Market Basket Analysis - Ukraine Conflict Tweets

Roberto Staino - 967485

Matteo Biglioli - 938199

March 31, 2022

1 Introduction

This report describes an analysis performed on the Ukraine Conflict Twitter Dataset. The aim of our project is to leverage a distributed algorithm in order to highlight frequent sets of words over multiple tweets; in particular we will implement, with the aid of the Map Reduce programming model, the A Priori algorithm to perform a Market Basket Analysis.

The computing engine we will use is **Apache-Spark**, more specifically its interface designed for compatibility with the Python programming language which is called **PySpark**. In order to achieve a distributed but concise pre-processing we will also leverage **SparkNLP**, an open-source text processing library which seamlessly integrate with both Python and Spark.

We will start by providing a naive approach which strictly follows the algorithm presented during classes. In this first part we will be mostly interested in replicating literally all the steps of the algorithm, without worrying too much about the performance of it.

We will then offer a generalized approach, coded in a single Python function, which will compute at once all the frequent itemsets of different sizes based on different input parameters. In this phase we will provide two different implementations of one of the most relevant steps of the computation, which is the generation of all the, not necessarily frequent, itemsets of a given basket: one of them will be based on a Map-Reduce-like approach and should perform better on large datasets analysed on a cluster of different computing nodes, while the other will be more sequential-like and should perform better in our case, in which we have a small dataset and a single executor.

Finally we will analyze the different results we will get by analyzing datasets generated in different days with different parameters values. In this last phase we will actually find some correlation between our results and a few key events which occurred during the conflict.

2 Dataset and Ingestion phase

As mentioned in the introduction, the input data of this analysis is part of the Ukraine Conflict Twitter Dataset, which contains more than a 14 Millions different tweets, collected since the beginning of the Ukraine conflict and partitioned by day, for a total of more than 3GB of textual data. The data is collected every 15 minutes by monitoring (worldwide) hashtags pertaining to the ongoing Ukraine-Russia conflict.

Each partition, provided as a compressed coma-separated-value file (*csv.gzip*), include both the plain text of every scraped tweet and a few metadata fields, like the language of the text and some information about the author of the tweet. For the purposes of our analysis we will take into account only the plain text and its language while discarding everything else.

The dataset can be downloaded directly through the Kaggle API, which return us a compressed file containing all the different compressed partitions. After decompressing the whole dataset in a predefined folder, Spark allow us to read the gzipped CSVs in a distributed manner*, without the need of leveraging non-distributed libraries such as Pandas; in this way we can avoid a possible performance bottleneck which would generate by reading the whole dataset with a non-distributed approach†.

Once the dataset has been loaded, we filter out all the non-English tweets and select only the text column. The filter on the language is needed due to the fact that in the pre-processing pipeline we will implement different steps, such as stemming and stepwords removal, which are strongly dependent on the language of the text‡.

3 Pre-Processing

During the pre-processing phase we want to "normalize" the plain text of our dataset, more specifically our goal for this step is to eventually provide a complete and cleaned input to the Market Basket Analysis algorithm. While we obviously need to split our tweets into *shingles*, which in our case will be 1-Grams (single words), we also want to remove impurities contained in the plain text which could make our job harder and negatively influence the final outcome.

In this phase we will not directly work with **RDDs**§ but with **DataFrames**¶; the reason for this is that the library we will be using, SparkNLP, provides compact and efficent APIs which work directly with DataFrames.

*We need to highlight that this feature of Spark is impacted by a known issue for which we need first to change the extension of all the gzipped files from *.gzip* to *.gz*; this is done using a simple bash script.

†In our one-node case the outcome does not change, but if we had a cluster this small feature would allow each worker to directly read a dataset partition instead of having to pass through the driver.

‡As a next step we could implement a language-dependent pipeline which would allow us to avoid the language filter.

§A Resilient Distributed Dataset is the basic abstraction that the Spark framework provides in order to work with massive datasets.

¶A Spark DataFrame is merely a Dataset[Rows], and a Dataset is nothing more than a new interface, added in Spark 1.6, which provides both benefits of RDDs and Spark SQL's optimized execution engine.

Our pre-processing pipeline will be composed of the following different steps:

1. **Document Assembler**: This first annotator is the entry point for every SparkNLP pipeline due to the fact that prepares data into a format that is processable by SparkNLP.
2. **Tokenizer**: In this step we split each text content (tweet in our case) into smaller shingles, called tokens, which corresponds to single words or punctuation marks.
3. **Link Remover**: Leveraging a Normalizer annotator from sparkNLP which filters out tokens matching a defined regex, we filter out all the links present in our text.
4. **Punctuation Remover**: Leveraging a Normalizer annotator from sparkNLP which filters out tokens matching a defined regex, we filter out all the punctuation present in our text.
5. **StopWords Cleaner**: In this step we remove English stopwords, such as conjunctions or articles, which could negatively impact our results. It would be highly probable indeed to find stopwords as frequent itemsets, due to the fact that they are frequently used in every written text.
6. **Stemmer**: This annotator returns hard-stems out of words with the objective of retrieving the meaningful part of the word. In this way nouns and adjectives get pinned together and we can generate more meaningful results.

This pre-processing pipeline return a detailed DataFrame with different columns containing the output of each step and some additional metadata. For the purpose of this analysis we will extract only the processed text of each tweet.

4 Market Basket Analysis Algorithm

After having pre-processed the input data we can start to work on the actual Market Basket Analysis.

First of all we recall that we will approach the problem using the Map-Reduce programming model; for this reason our input DataFrame will be mapped to a Spark RDD (thanks to the *.rdd* method) in which data will be organized as **Key-Value** pairs.
More specifically we will set each key to 1 while the value will contain list of stemmed words of a tweet.

We will also remove possible duplicated words through a *set* function because we are not interested in processing multiple identical items which are in the same basket.

```
# Because from now on we will define the algorithm using
# the Map and Reduce functions,
# which are only defined over rdd in pyspark, we will convert
# our preprocessed dataframe into an rdd
# with the same structure that we saw in the hadoop framework: (key, value)
# + we will map each tweet in a list(set(tweet))
# in order to remove duplicate words which are not useful in our analysis
input_rdd = preprocessed_df.rdd.map(lambda x: (1, list(set(x[0]))))

# This is our input rdd
```

```
input_rdd.take(1)

#[(1, ['moscow', 'vladimir', 'putin', 'visit', 'nation', 'space', 'centr'])]
```

Before diving in the details of the implementation we would need to define a threshold which would determine weather a set of words is considered frequent or not. While for implementational reason we would need a specific number, we prefer to define it with respect to the total number of baskets (tweets) in our dataset; in order to achieve this we just define the threshold as the number corresponding to 20% of our baskets.

```
# We start by defining a proper threshold in order to understand
# if an itemset is frequent.
# To do that we count the number of baskets (tweets) in our
# whole dataset and we define an itemset "frequent"
# if it appears in over x% of the baskets

# Note that the countByKey function is basically equivalent
# to the map-reduce structure
# .map(lambda x: (1, 1)).reduceByKey(lambda x,y: x + y)
# But in our test it performed at twice the speed!

THRESHOLD_PERCENTAGE = 0.15
n_of_baskets = input_rdd.countByKey()[1]

THRESHOLD = math.ceil(n_of_baskets * THRESHOLD_PERCENTAGE)
```

4.1 A Priori Algorithm and Monotonicity Property

As stated in the introduction, we will implement an algorithm which is called A Priori Algorithm. The underling idea of this approach is that of avoiding having to compute the frequency of all the possible itemsets generated from our dataset by generating a set candidates whose frequency will be checked in order to avoid false positives.

More specifically we will start by evaluating frequent singletons (words) with a naive approach (i.e. by counting the occurrences of each word in our corpus) but we will then continue by checking only itemsets which are generated as a combination of these frequent singletons.

This approach is very convenient thanks to a logical implication called **anti-monotone property of support** or **monotonicity**. This property implies that if an item drops out from an itemset, the support value of the new generated itemset will either be the same or will increase. In other words, if a set of words is frequent, all its subsets must be frequent as well. It cannot exist a frequent itemset generated by items which are not frequent themselves.

This implication grants us the possibility to build an efficient algorithm that has no false negatives. In fact, it is possible to build candidate itemsets by joining frequent subsets found in the previous step. Starting from frequent singletons, we can build candidate pairs that could be frequent, starting from frequent pairs we can build candidate triplets that could be frequent, and so on and so forth. In this way, only the candidate itemsets could be potentially frequent, hence their frequency is evaluated, while every other itemset is necessarly not frequent and hence discarded.

4.2 Naive approach

Our first naive approach strictly follows the algorithm presented during classes. For this reason we compute, in different steps, frequent singletons, pairs and triplets. As stated in the Introduction of this report, in this first part we will exclusively focus on the theoretical algorithm without worrying too much about possible enhancements which could be implemented in order to improve the overall performance.

4.2.1 Frequent Singletons

We start off by computing frequent singletons; the underling idea in this part is to compute the frequency of each word in our dataset and extract only the ones whose frequency exceeds our predefined threshold.

The steps to obtain this result are the following:

1. First of all we map each word of every tweet to a key-value pair in which the key is the word itself and the value is 1. We then flatten our result so that the output of this first step is a flattened list of all the words in our dataset.

```
# Extract all singletons
singleton_itemsets_rdd = input_rdd \
                         .flatMap(lambda x: x[1]) \
                         .map(lambda x: (x,1))

singleton_itemsets_rdd.take(2)
# [('moscow', 1), ('vladimir', 1)]
```

2. We then perform a Reduce (by key) operation in which we sum all the values with the same key (in our case, a unique word). This provides us with the frequency of each single word in our dataset encoded in key-value pairs in which the key is a unique word and the value is the number of times that particular word is found in the whole corpus.

```
# Compute frequencies using a reduce operation
singleton_itemsets_w_frequencies_rdd = singleton_itemsets_rdd \
                                       .reduceByKey(lambda x,y: x + y)

singleton_itemsets_w_frequencies_rdd.take(2)
# [('moscow', 77), ('vladimir', 28)]
```

3. Finally we filter out all the words whose frequency is under the predefined threshold, obtaining as output the list of frequent singletons.

```
# Filter singletons with a frequency higher than THRESHOLD
frequent_singleton_itemsets_rdd = singleton_itemsets_w_frequencies_rdd
                                  .filter(lambda x: x[1] > THRESHOLD)

frequent_singleton_itemsets_rdd.take(2)
# [('putin', 733), ('amp', 449)]
```

4.2.2 Frequent Pairs

We then continue our analysis by extracting frequent pairs; the underling idea in here is to generate candidate pairs of words which could be frequent in order to avoid checking the frequency of all the possible pairs. The candidate pairs are generated by computing all the possible combinations of two elements among the frequent singletons.

While we can be sure that this approach will not generate false positives because we will eventually check the actual frequencies of all the candidate pairs, the eventuality that we could generate false negatives (i.e. frequent pairs which will not show up as candidates) is handled by the correct interpretation of the Monotonicity Property explained above.

The steps to obtain this result are the following:

1. First of all we generate all the possible combinations of two frequent singletons by performing a join operation in which the joining key is equal to 1 for all the values.

```
# Extract all frequent singleton without frequency
freq_singleton_itemsets_wout_freq_rdd = frequent_singleton_itemsets_rdd \
                                            .map(lambda x: (1, x[0]))

# Generate all candidate pairs
# (note that we need to remove from the join all possible duplicates)
# This is not the fastest approach as we could have leveraged
# specific functions (like .distinct()) but it is the most
# academic and hadoop like approach
candidate_pairs_rdd = frequent_singleton_itemsets_wout_freq_rdd \
                      .join(frequent_singleton_itemsets_wout_freq_rdd) \
                      .filter(lambda x: len(set(x[1])) == len(x[1])) \
                      .map(lambda x: (tuple(sorted(x[1])), 1)) \
                      .reduceByKey(lambda x,y: x)

candidate_pairs_rdd.take(2)
# [(('putin', 'ukrainian'), 1), (('amp', 'ukrain'), 1)]
```

   As stated in the comment we highlight that we cannot just easily join the frequent singleton set with himself but we need to discard the possible duplicates, which are both pairs containing the same value (i.e. ('putin', 'putin')) and pairs sorted in opposite ways (i.e. ('putin', 'ukrainian') and ('ukrainian', 'putin')). This is achieved by respectively leveraging a filter and a Map-Reduce operation:

```
    .filter(lambda x: len(set(x)) == len(x)) \
    .map(lambda x: (tuple(sorted(x[1])), 1)) \
    .reduceByKey(lambda x,y: x)
```

2. We can then broadcast the list of candidate pairs[‖] in order to make it available locally to each executor, which would definetly improve the overall performance.

```
# We can then generate a list of candidate pairs
# which can be broadcasted to all executors
# due to the fact that we expect this to be small
# (it could be even part of our final output)
candidate_pairs_list = candidate_pairs_rdd.map(lambda x: x[0]).collect()
broadcasted_candidate_pairs_list = sc.broadcast(candidate_pairs_list)
```

3. Now we need generate every possible pair of items (words) of each basket (tweet) in our dataset. In order to achieve this we could easily perform a join operation on the dataset as we did in the step above, but this time we need to have different unique keys for each tweet, in order to avoid spurious pairs between words of different tweets.

```
# Compute all possible pairs on our whole dataset
input_w_unique_key_rdd = input_rdd \
                         .map(lambda x: x[1]) \
                         .zipWithUniqueId() \
                         .flatMap(lambda x:
                             [(x[1], word) for word in x[0]])

pair_itemsets_rdd = input_w_unique_key_rdd \
                    .join(input_w_unique_key_rdd)

pair_itemsets_rdd.take(2)
# [(0, ('moscow', 'moscow')), (0, ('moscow', 'vladimir'))]
```

4. Finally we can filter out every pair of words which does not appear in our candidate's list. We then compute the frequency of each remaining pair and filter out all the itemsets whose frequency is under the predefined threshold, obtaining as output the list of frequent pairs.

```
# Filter only pairs that are in the candidate
# frequent pairs list and compute their freq
freq_pairs_itemsets_rdd = pair_itemsets_rdd \
        .filter(lambda x: x[1] in broadcasted_candidate_pairs_lst.value) \
        .map(lambda x: (x[1],1)) \
        .reduceByKey(lambda x,y: x + y) \
        .filter(lambda x: x[1] > THRESHOLD)

# Frequent pairs itemsets w/ the relative frequency
frequent_pairs_itemsets_rdd.take(2)
# [(('russia', 'ukrain'), 1241), (('russian', 'ukrain'), 493)]
```

In order to generate a unique id for each tweet in our dataset we used the Spark function zipWithUniqueId which is already optimized to work with a distributed dataset.
We know that we could have leveraged an hash function over each text in order to generate a unique identifier. The problem with this different approach, which will be reported in the code below, is that in order to avoid collision for identical tweets[**] we would have needed to insert as input for the hash function also a random number for each tweet.
Unfortunately when we implemented this kind of approach combined with a FlatMap operation

[‖]Note that we expect this list to be small enough to be broadcasted due to the fact that it could be a part of our final output.

[**]That could happen more frequently that expected (i.e. *Stop War*).

Spark was not able to properly "order" these two operations, and it would end up generating a random number for each word of each tweet which would obviously result in an empty join.

```
# BONUS: We know that a more appropriate approach with respect to zipWithUniqueId
# would have been to generate a unique index using an hash function
# like the one reported below
# but it seems that (maybe due to the lazy not linear
# computational approach of spark)
# this approach did not manage to generate a proper
# joined rdd (even if we used caching)

def unique_key(basket):
  unique_value = f"{random()}-{basket}"
  return hashlib.sha256(unique_value).encode("utf-8")).hexdigest()

input_w_unique_key_rdd = input_rdd \
                         .map(lambda x: (unique_key(x[1]), x[1])) \
                         .flatMap(lambda x: [(x[0], word) for word in x[1]])
```

4.2.3 Frequent triples

In this last step of the Naive approach phase we will extract frequent triplets. Because the process is very similar to the one described before, we will present thoroughly only the few differences.

The steps are the following:

1. First of all we need to generate the list of candidate triplets. In order to do so we will join the frequent pairs found previously, obtaining quadruplets. (i.e. ('putin', 'nato', 'ukrain', 'russia'), ('putin', 'nato', 'nato', 'russia')).
   We will then extract only the unique values of each quadruplet and filter out all the results whose size differ from 3[††].
   Finally as we did before we will filter out duplicates generated by a different sorting of the tuple.

   ```
   # Generate all candidate triplets
   # NOTE that we need to remove from the join all possible duplicates
   # NOTE that we generate candidate triplets by joining frequent pairs,
   # in this way we generate the lowest possible number of candidates
   # by leveraging the monotonicity property
   candidate_triplets_rdd = frequent_pairs_itemsets_wout_freq_rdd \
                .join(frequent_pairs_itemsets_wout_freq_rdd) \
                .map(lambda x: (1, sum(x[1], ()))) \
                .map(lambda x: (1, tuple(set(x[1]))) ) \
                .filter(lambda x: len(x[1]) == 3) \
                .map(lambda x: (tuple(sorted(x[1])), 1)) \
                .reduceByKey(lambda x,y: x)

   candidate_triplets_rdd.take(2)
   # [(('russia', 'ukrain', 'ukrainian'), 1),
   #  (('putin', 'russia', 'ukrain'), 1)]
   ```

[††]Note that we expect both cases in which the size of the set is greater than 3 (i.e. ('putin', 'nato', 'ukrain', 'russia')) and cases in which it is lower (i.e. ('nato', 'ukrain', 'nato', 'ukrain'))

2. We then broadcast the list of candidates as we did in the previous iteration:

```
# We can then generate a list of candidate triplets
# which can be broadcasted to all executors
# due to the fact that we expect this to be small
#(it could be even part of our final output)
candidate_triplets_list = candidate_triplets_rdd.map(lambda x: x[0]).
                                                collect()
broadcasted_candidate_triplets_list= sc.broadcast(candidate_triplets_list)
```

3. Now we need to generate all the possible triplets of each tweet in order to filter out all of the non-candidates. As we reported in the comment, this is an approach which, while being highly scalable, is definitely extremely expensive to be performed on a single node. In our specific case we could have just saved all the frequent pairs and join only those or, even better, directly check, for each tweet, the presence of a candidate triplet with a for-comprehension approach. We will discuss this different approach later on in the report.

```
# Compute all triplets on our whole dataset
# Note that this is definetly an extremely expensive approach,
# we could have just saved all the frequent pairs and join only
# those or (better) avoid having to compute all triplets
# and directly check, for each tweet, the presence of a candidate triplet
# (this approach will be used in the generalization phase, for now we
                                                stick with the naive approach)
triplets_itemsets_rdd = input_w_unique_key_rdd \
                        .map(lambda x: (x[0], (x[1], ))) \
                        .join(pair_itemsets_rdd)  \
                        .map(lambda x: (1, sum(x[1], ())))

triplets_itemsets_rdd.take(2)
# [(1, ('construct', 'construct', 'construct')),
#   (1, ('construct', 'construct', 'nation'))]
```

4. Finally we can extract the frequent triplets in the same way we did for the frequent pairs.

```
# Filter only triplets that are in the candidate frequent triples list
# and compute their freq
frequent_triplets_itemsets_rdd = triplets_itemsets_rdd \
    .filter(lambda x: x[1] in broadcasted_candidate_triplets_list.value) \
    .map(lambda x: (x[1],1)) \
    .reduceByKey(lambda x,y: x + y) \
    .filter(lambda x: x[1] > THRESHOLD)

# These are examples of frequent pairs itemsets with the relative
                                                frequency
frequent_triplets_itemsets_rdd.take(2)
# [(('putin', 'ukrain', 'ukrainian'), 112),
#  (('russia', 'ukrain', 'ukrainian'), 140)]
```

4.3 Generalized Approach

In this second approach we defined a function able to perform the analysis at once. The underling idea is that it keeps looking for larger and larger frequent sets of words, and it stops when it can't find any new ones.

Because most of the code reported in the function is the same that we used in the naive approach we will present only the key differences.

- First of all, the function starts by computing the frequent singletons and generating candidate pairs in the same way as we did in the Naive approach. This step is different from the others because we do not yet have candidate itemsets to check, and this is also the reason why it is performed prior to the while loop.
- Before the while loop the function generates a unique identifier for each tweet in the same way we saw before. This step is computed only if the chosen approach is "hadoop-wise" (we will se later what this means). We only highlight that we will leverage a technique called **Caching**[‡‡] on this particular RDD, that is because we will use it repeatedly over our computation.
- Then, as we mentioned above, we introduced a while loop which helps us to look for every possible itemset with frequency over the predefined threshold. In order to to this we implemented a condition which check if the step before ended with the generation of new candidate itemsets:

```
# Loop until we have candidates for frequent itemsets
while candidate_itemsets_rdd.count() != 0:
```

- As most of the code contained in the while loop is basically identical to the one explained during the Naive approach we will only focus on the way we generate frequent itemsets.
  We implemented two different approaches:

  1. The first approach is called "hadoop-wise" and it is similar to the one described previously, the only difference here is that we define the generation of itemsets from a tweet using a variable which is updated at each iteration of the loop. This allow us to compute, at each iteration, itemsets of increasing size.
  2. The second approach, called "single_machine-wise", is definitely more interesting and it allow us to drastically reduce the computational time. The idea is to check, for each candidate itemset of words, if it is a contained in the set of words of every single tweet, introducing a sequential process.

```
# single_machine-wise: Generate all itemsets of length
# "candidate_itemsets_length" and filter them based on THRESHOLD
if approach == "single_machine-wise":
  tmp_frequent_itemsets_rdd = input_rdd \
    .map(lambda x : [(candidate_itemset, 1)
                     for candidate_itemset in
                     broadcasted_candidate_itemsets_list.value
                     if set(candidate_itemset).issubset(x[1])]) \
    .flatMap(lambda x : x) \
```

[‡‡] Caching an RDD means persisting it in the local memory of the executors.

```
            .reduceByKey(lambda x,y: x + y) \
            .filter(lambda x: x[1] > THRESHOLD) \
            .map(lambda x: (1, x[0]))
```

While this approach makes the code more intensive on each single worker, which now has to go through a list comprehension step as long as the list of candidates, it also require less memory.
We speculate that this kind of balance makes the hadoop-wise approach faster if deployed on a cluster with a lot of small nodes, while the sinle_machine-wise approach should be a better fit for a smaller cluster with stronger nodes in terms of computation[§§].
To conclude, we highly suggest using the "single machine-wise" approach for non-distributed or small distributed systems, that have a fairly good computational capacity. For large distributed systems, with decent computational capacity, we suggest the "hadoop-wise" approach, as it maybe be slightly slower but nicely convenient.

5 Results

In this section, the result of the generalized analysis are reported[¶¶].

For our academic purposes the analysis is limited to twenty thousand tweets per day.

We start by proposing a trivial illustration of the results achieved on tweets dated 28th February 2022:

```
# 2022-02-28
# threshold_percentage = 0.2, limit = 20000
['putin',
 'ukrain',
 'russia',
 'ukrainian',
 'russian',
 'ukrainerussiawar',
 ('russia', 'ukrain'),
 ('ukrain', 'ukrainian'),
 ('russian', 'ukrain')]
```

It stands out immediately that the longest itemsets are pairs, probably because of the limited number of tweets we choose to take into consideration with respect to the 20% threshold chosen for the analysis.
This is also consistent with the small number of singletons highlighted, of course the more the singletons the higher the probability to form larger itemsets.
Moreover, there are very similar words such as "russia" and "russian" or "ukrain" and "ukrainian", that are nouns and adjectives with the same root. This means that the stemming phase could be improved and the **SparkNLP** tools don't really help us with that specific pre-processing task.

[§§]Note that this is an out-of-topic discussion in this situation, in which the length of the candidate list is definitely not a problem even for weaker machines.

[¶¶]We highlight that in order to easily generate results for different days and with different parameters we built, on top of the A-Priori algorithm function a small generalization layer. Because this part is merely a computer science exercise, we will not discuss in this paper how it was implemented but we accurately commented the code in order to make it easier to understand.

We can then see how the result changes when we lower a bit the threshold:

```
# 2022-02-28
# threshold_percentage = 0.08, limit = 20000
['putin',
 'ukrain',
 'russia',
 'countri',
 'amp',
 'ukrainian',
 'kyiv',
 'russian',
 'peopl',
 'russiaukrainewar',
 'ukrainerussiawar',
 'war',
 ('russia', 'ukrain'),
 ('putin', 'ukrain'),
 ('ukrain', 'ukrainian'),
 ('putin', 'russia'),
 ('russian', 'ukrain'),
 ('russia', 'russian'),
 ('ukrain', 'ukrainerussiawar'),
 ('ukrain', 'war'),
 ('russia', 'russian', 'ukrain')]
```

As expected by lowering the threshold we immediately see an higher number of frequent itemsets returned and even a triplet.

For a better overview, we decided to run the analysis throughout several days and to point out the main differences which can be spotted.
In order to do so, the better option was to save the output on a *.json* file.
Here's an interesting result that stood out:

```
#    "2022-03-04":
#    singletons
     "ukrain", "russian", "plant", "power", "russia",
     "nuclear", "zaporizhzhia", "fire", "putin"
#    pairs
     ["russian", "ukrain"], ["plant", "power"], ["plant", "ukrain"],
     ["power", "ukrain"], ["russia", "ukrain"], ["nuclear", "plant"],
     ["plant", "zaporizhzhia"], ["nuclear", "power"], ["power", "zaporizhzhia"],
     ["nuclear", "zaporizhzhia"], ["nuclear", "ukrain"], ["ukrain", "zaporizhzhia"]
#    triplets
     ["plant", "power", "ukrain"], ["nuclear", "plant", "ukrain"],
     ["nuclear", "power", "ukrain"], ["nuclear", "plant", "power"],
     ["plant", "power", "zaporizhzhia"], ["nuclear", "plant", "zaporizhzhia"],
     ["nuclear", "power", "zaporizhzhia"]
#    quadruplets
     ["nuclear", "plant", "power", "zaporizhzhia"],
     ["nuclear", "plant", "power", "ukrain"]
```

We assessed that the 4th of March 2022, the Russian forces captured a Ukrainian power plant, the largest in Europe. After a massive fire was extinguished, Ukrainian workers continued to operate the facility and n release of radioactive material had been reported (Source).

This result proves not only that the algorithm works as expected, but also that we can clearly

see how the events happening in the war affected the way people are talking about it on social media .
In fact, among the frequent singletons we can find ”power”, ”plant”, ”fire”, ”nuclear” and ”zaporizhzhia”, the city involved in the attack.
Also, the quadruplets generated are another proof that the public eye was focused on the nuclear concern that the assault on a that specific power plant could generate.

We included different results in the GitHub repository containing the code of this project.

6 Conclusion

In this project we implemented the A Priori algorithm to perform a Market Basket Analysis over a dataset of tweets. We wrote each line of this code (ingestion included) with scalability in mind; in fact this code could be deployed in a cluster and it would be able to leverage all the advantages that a distributed system could provide.

We also highlighted during the discussion a few next steps that could be taken in order to improve the results obtained:

- We could provide a multi-language implementation of the pre-processing pipeline, this could allow the user to perform an analysis also for tweet which are not written in English. Note that this analysis should be isolated for different languages, due to the fact that we do not expect to find itemsets containing words from different languages.
- We could improve the stemming step of the pre-processing pipeline, in particular for what it concerns the words Russia/Russian and Ukraine/Ukranian. This could help us in understanding better the actual results because it would remove a lot of spurious similar results.
- We could implement a function which compares the results of different days, in order to be able to analyze the evolution of the behaviour of users without the need of mechanically looking at each result.

We declare that this material, which We now submit for assessment, is entirely our own work and has not been taken from the work of others, save and to the extent that such work has been cited and acknowledged within the text of our work. We understand that plagiarism, collusion, and copying are grave and serious offences in the university and accept the penalties that would be imposed should I engage in plagiarism, collusion or copying. This assignment, or any part of it, has not been previously submitted by us or any other person for assessment on this or any other course of study.