Market Basket Analysis On Ukranie-Russia Conflict Tweets

UNIVERSITÀ DEGLI STUDI DI MILANO

LA STATALE

Deparment of Economics,Management
and Quantitative Methods
Data Science and Economics

Project of Algorithms For Massive Data

Murat Aydın

March 2022

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

Introduction

This project focuses on the implementation of a market basket analysis on the tweets about Ukraine-Russia conflict dataset from Kaggle repository that contains information about tweets.

The market basket analysis consists in finding the frequent itemsets, which are the items appearing together in the same baskets a sufficiently large number of times defined in advance. For this analysis, the whole textual content are considered as baskets and the single words as items. This concept is generally exemplified by customers and their purchases. For example, the information about the purchased items of a customer can be analysed to derive association rules which is basically the regularities or recurrences in the baskets. In this case the baskets is the total number of products bought at a single transaction while the items are what is in the basket such as like bread or milk.

Dataset

In Kaggle Repository,all the tweets tweeted about Ukraine-Russian conflict from 27 February 2022 to 8 March 2022 is published. Total number of tweets is 6.361.444 millions. Each day from 27 February to 8 March has its own csv file with the same structure. Dataset has 17 features in total however we will use only 3 of them for this analysis. In particular, our dataset will have the following features:

-*userid*: a unique identifier assigned to each user.
-*text* : the textual content of the tweet.
-*language* : the language of the text

From this dataset, we build one additional feature that is tokenized, cleaned form of textual content in a suitable form for the algorithms we will use.

Pre-Processing

For this project, to be computationally faster we use 500.000 examples from the original dataset. To get such tweets, we generated random integers in range(0, len(dataset) and sliced the original dataset to a get reduced form of it. This might sound too little however not all the tweets in the dataset serve for our aims. In particular, after removing duplicate tweets (re-tweeted tweets) and filtering them by the English language, what we are left with is only 1.138.499 millions of tweets. Thus, we can say that we use nearly %50 of the clean dataset.

Recall that our basket is the textual content of a given tweet, while the words in that tweet are our items. Therefore, to get a reasonable result from our algorithms, we need to remove white spaces, clean from stop-words, punctuations and repetition of the words since each basket needs to be composed of unique items. To reach this end, we first remove punctuations and then tokenize the whole **text** column to do the remaining steps mentioned above. This is how our original, reduced-form data looks like :

userid	text	language
1181058490774642688	#PopeFrancis expr...	en
867630410	@carolecadwalla M...	en
1179528759034236929	A very long trave...	en
1500756161167228930	The pub is open\n...	en
1266213324481593347	🚨 Borodyanka des...	en
262832017	Heartbreaking! #K...	en
1369033038148100098	They wanted to ba...	en
1329959992708001793	KREMLIN CONDEMNS ...	en
44596502	@AmaduwurrieJa11 ...	en
3398202539	It’s too early to...	en

only showing top 10 rows

Figure 1: Raw data

Where we have *userid, textual content of the tweet and the language it is tweeted in.*

After the pre-processing, we will have something of the following form:

userid	text	words_clean
1181058490774642688	popefrancis expre...	[popefrancis, fas...]
867630410	carolecadwalla mu...	[invasion, ikea, ...]
1179528759034236929	a very long trave...	[trip, place, gri...]
1500756161167228930	the pub is open\n...	[pfizer, 🤘🤘🤘🤘...
1266213324481593347	🚨 borodyanka des...	[httpstcobucmapsm...
262832017	heartbreaking kyi...	[ukraine, httpstc...
1369033038148100098	they wanted to ba...	[tampons, smh, lo...
1329959992708001793	kremlin condemns ...	[paralympic, inte...
44596502	amaduwurrieja11 u...	[working, loveukr...
3398202539	it's too early to...	[control, it's, w...

only showing top 10 rows

Figure 2: Final data

Where the **text** column is transformed into a basket of items where each item is unique, cleaned of punctuations, white-spaces and stop-words.

Algorithms

In order to find the frequent item sets, two algorithms have been considered in this project. In particular, we will use **FP growth** and **Apriori** algorithms.

FP growth

FP-growth algorithm takes as input the items in a basket, in this case the **words-clean** column in the Final data shown in figure 2 along with a user-defined minimum threshold value.In the first step, it calculates the frequency of items and checks them against the minimum threshold. In the second step, it creates a data structure called **frequent pattern tree** where only the frequent singletons and their count is stored. Every time a pattern is found, the corresponding item's count is increased. After the second step, the frequent items can be extracted from the tree.

Apriori

Apriori algorithm has two phases. Similarly to FP growth, it takes as input the basket of items along with a minimum support threshold. This algorithm exploits the monotonicity property that says the support of an itemset cannot be greater than then the support of its subsets. This implies that all the subsets in a item-set should be frequent as well. This clear approach allows the Apriori to reduce number of possible items to be evaluated. The algorithm uses the following metric to decide whether an itemset is frequent or not :

$$supp(j) = \frac{freq(j)}{totaln.baskets}$$

First Pass

1. Each item is initialized with a counter equal to one. *(item,1)*
2. Sum the occurrences of each item collected in the first step. *(item,numberofoccurrences)*
3. Keep only the items where numberofoccurrences > minimum support.

Second Pass

1. Those items whose occurrences are greater than then minimum support are combined to create new itemsets. These are called **candidate pairs.**
2. For each new pair created in the first step, Apriori checks if such pairs are included in the basket, if so, the corresponding counter is increased.
3. If the resulted counter of a given candidate pair is > minimum support, they are appended to the frequent item list.

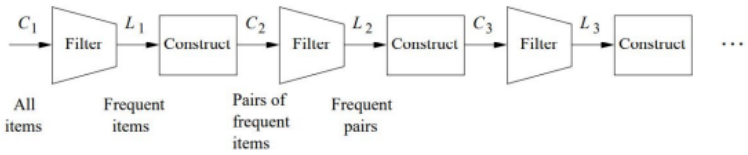
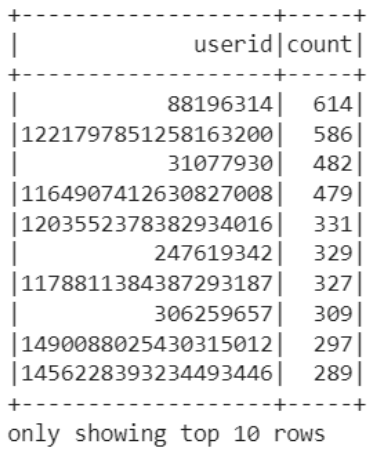


Figure 3: Apriori Process

Data Inspection

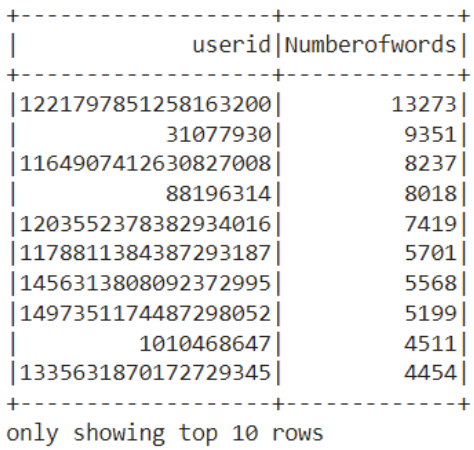
Before going into the experiment results, we would like to go a little deeper in the dataset.In particular, this small analysis will show us why we used only %50 of the original dataset.



```
+------------------+-----+
|            userid|count|
+------------------+-----+
|          88196314|  614|
|1221797851258163200|  586|
|          31077930|  482|
|1164907412630827008|  479|
|1203552378382934016|  331|
|         247619342|  329|
|1178811384387293187|  327|
|         306259657|  309|
|1490088025430315012|  297|
|1456228393234493446|  289|
+------------------+-----+
only showing top 10 rows
```

Figure 4: Number of total baskets per user

In figure 4, we see total number of baskets per user. This means that the dataset contains many different tweets of the same user.



```
+------------------+-------------+
|            userid|Numberofwords|
+------------------+-------------+
|1221797851258163200|        13273|
|          31077930|         9351|
|1164907412630827008|         8237|
|          88196314|         8018|
|1203552378382934016|         7419|
|1178811384387293187|         5701|
|1456313808092372995|         5568|
|1497351174487298052|         5199|
|         1010468647|         4511|
|1335631870172729345|         4454|
+------------------+-------------+
only showing top 10 rows
```

Figure 5: Number of total items per user

In figure 5, we are looking at the number of total items per user. An item here represents a single word. For example, the first user in

Figure 5 which is the second user in Figure 4 in terms of the highest number of tweets has the highest number of unique words used. So the textual content of that user is richer in terms of different words with respect to the other. Why we are interested in such numbers? Because more unique words means larger baskets which means more storage. In particular, the total number of unique words, in other words, total number of unique items in our reduced-form dataset is 8.406.952 which is fairly sufficient for our analysis even if we use only %50 of the whole data.

FP-growth results and Scalability

FP Growth algorithm implementation chosen for this work is the pyspark implementation. After trying many different thresholds, we decided to use a minimum support of 0.02 since lower values resulted in irrelevant items while the higher values returned a few items.

```
+------------------+------+
|items             |freq  |
+------------------+------+
|[ukraine]         |297247|
|[russia]          |161398|
|[russia, ukraine] |104651|
|[putin]           |103436|
|[russian]         |84491 |
|[war]             |67018 |
|[putin, ukraine]  |56576 |
|[russian, ukraine]|54312 |
|[nato]            |46351 |
|[people]          |42933 |
|[war, ukraine]    |42487 |
|[ukrainian]       |38610 |
|[ukrainerussiawar]|37183 |
|[amp]             |35778 |
|[stop]            |35172 |
+------------------+------+
only showing top 15 rows
```

Figure 6: Frequent items found by FP growth with threshold 0.02

The result is somehow expected. The most frequent words turned out to be **Ukraine, Russia, Putin, War.** FP-growth with threshold 0.02 returns 254 frequent items.

antecedent	consequent	confidence	lift	support
[close, ukraine]	[sky]	0.8516299308081889	28.32158067203821	0.023878
[defend, humanitarian, weapons]	[needs]	0.982999213836478	30.545000740677335	0.020006
[assistance, humanitarian]	[weapons]	0.9384169703766003	27.059312871297585	0.020084
[assistance, humanitarian]	[needs]	0.9421549387907672	29.27583552267625	0.020164
[assistance, humanitarian]	[ukraine]	0.9595364919166433	1.6142453490921258	0.020536
[assistance, humanitarian]	[defend]	0.9460798056256424	32.46447757963223	0.020248
[provide, stop]	[putin]	0.9353955652888037	4.5220523141608675	0.020502
[provide, stop]	[ukraine]	0.9428779998175016	1.5862204708092649	0.020666
[provide, stop]	[innocent]	0.9162332329592116	29.470351655169235	0.020082
[stopputin]	[ukraine]	0.8016170838330303	1.3485747131362615	0.041442
[potus, putin]	[ukraine]	0.8482391287065745	1.4270078105080506	0.02134
[defend]	[ukraine]	0.903653344320835	1.5202321334832307	0.026318
[potus, stopputin]	[ukraine]	0.9169593577521324	1.542617077127766	0.02193
[provide, putin]	[weapons]	0.909932584269663	26.23796379093607	0.020246
[provide, putin]	[ukraine]	0.9168539325842696	1.5424397184881171	0.0204

only showing top 15 rows

Figure 7: Association rules with threshold 0.02

The association rule, figure 7, results are similar to what one would have expected. Note that supports higher than 0.02 did not result in any association rule.

Threshold	Time Elapsed(seconds)
0.01	96.22094345092773
0.02	83.52431321144104
0.06	81.16606903076172
0.09	81.67123103141785
0.1	85.01623153686523

Figure 8: Elapsed Time in seconds vs different threshold

Here we tried different minimum thresholds. Lower thresholds requires more time therefore there is a lowering trend in time with respect to the thresholds used. As seen from the figure 8, smaller minimum support values needs more time to be processed and obviously returns more frequent item sets. We tried even smaller thresh-

olds but that leads Google Colab to crash.

To study the scalability of the algorithms proposed, we used different dataset sizes. In particular, we extracted %10, %30, %50, %70, %90 of the dataset and compared the time required for operation to be completed.

Fraction	Time Elapsed(seconds)
0.1	30.303032159805298
0.3	31.29839015007019
0.5	36.52366089820862
0.7	45.83070635795593
0.9	53.735288858413696

Figure 9: Elapsed time with threshold 0.02 vs fractions

In figure 9, we present the elapsed time with respect to the different fractions of the dataset used. As the dataset gets larger, the algorithm takes more time to process it.

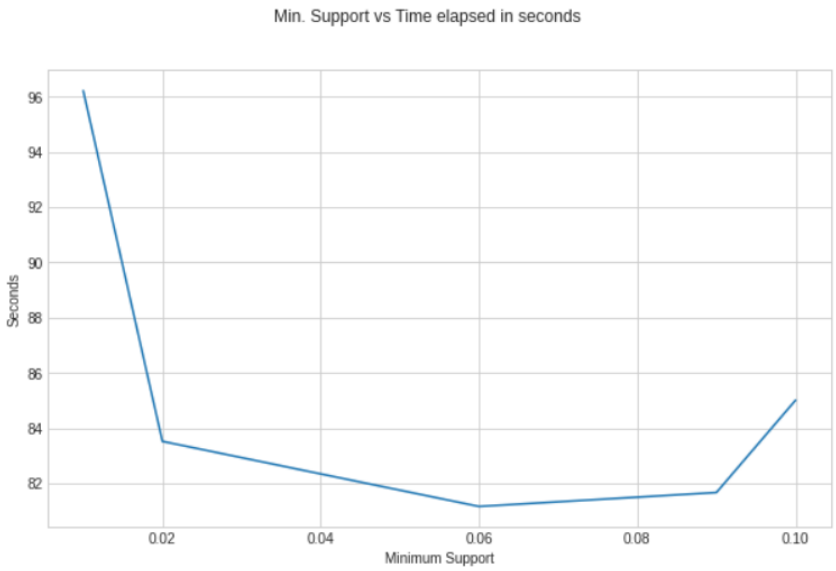


Figure 10: Python Plot of Figure 6

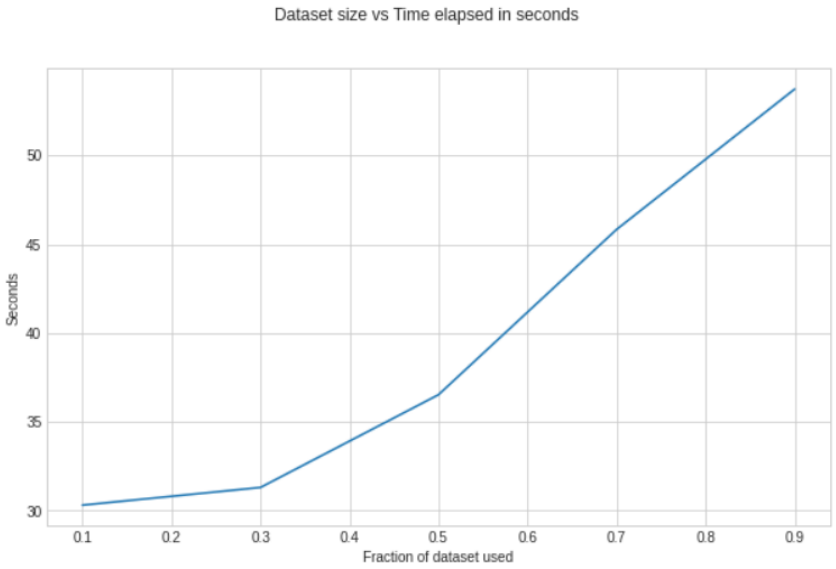


Figure 11: Python Plot of Figure 7

Both plots in Figure 10 and 11 clearly shows how FP-growth nicely scales up with the larger operations and larger datasets. Given any fractions of the dataset, the algorithm was able to process them without the consumption of all the system RAM.

A-priori results and Scalability

For the implementation of A-priori algorithm, we used mlxtend python library. Since we are not relying on Pyspark, the algorithm is not able to process 500.000 tweets. We were able to use only 10.000 tweets for the A-priori implementation. Therefore, we cannot really make a comparison between FP-growth and A-priori results in this project.

A-priori algorithm scales up by filtering the data thanks to its monotonicity property which states that if a basket of items are frequent, then it should be frequent any subset of baskets in that basket. This is highly related to the **maximality** concept. A set is maximal if there is no superset that is frequent. Therefore, if we collect all the maximal sets, then we know all of their subsets are frequent thanks to the **monotonicity property.** On the other

hand, if they are not maximal, then their subsets cannot be frequent. Given this nature of the algorithm, it makes sense that the algorithm discards all the subsets failing to reach the minimum support. That is why we say the algorithm scales up with the dataset size.

	support	itemsets
16	0.5980	(ukraine)
7	0.3225	(russia)
32	0.2124	(russia, ukraine)
6	0.2035	(putin)
8	0.1681	(russian)
23	0.1307	(war)
30	0.1099	(putin, ukraine)
34	0.1065	(russian, ukraine)
3	0.0894	(nato)
4	0.0878	(people)

Figure 12: Frequent items found by A-priori with threshold 0.04

In figure 12, we present the frequent items found by A-priori algorithm. Even if we cannot make a real comparison between FP-growth and A-priori, we can see that the results are highly similar. This threshold (0.04) returns 42 frequent items while with 0.02 threshold A-priori returns 321 frequent items that is slightly larger than the FP-growth that returned 254 items with threshold 0.02.

	Threshold	Time Elapsed
0	0.01	24.431689
1	0.02	13.683434
2	0.06	15.196359
3	0.09	13.404383
4	0.10	12.053977

Figure 13: Min.Support vs Time Elapsed

In figure 13, we present the table showing the thresholds used and their respective elapsed time values.

	Fraction	Time Elapsed
0	0.1	17.114436
1	0.3	16.745912
2	0.5	16.811858
3	0.7	16.490144
4	0.9	17.582630

Figure 14: Fraction of dataset vs Time Elapsed

In figure 14, we present the elapsed time with respect to the dataset size.

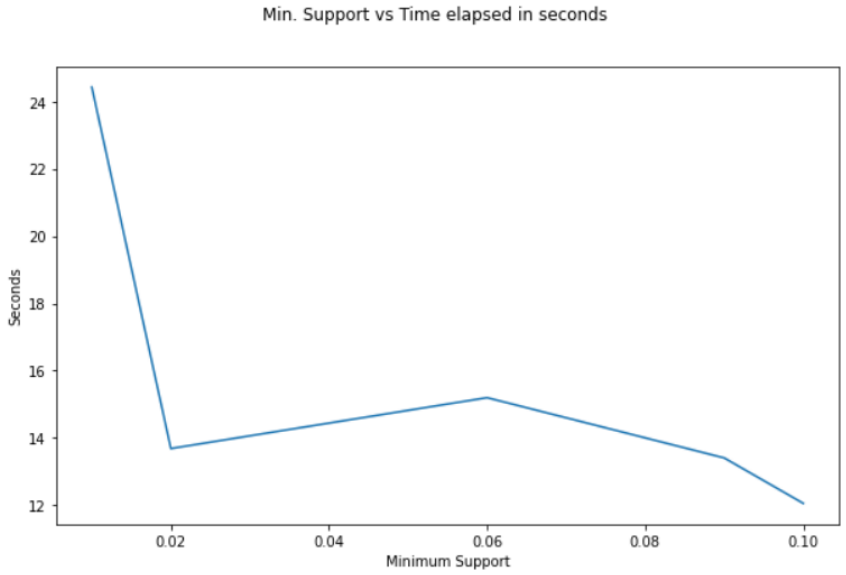


Figure 15: Python plot of figure 13

In Figure 15, we can see the plot of figure 13. As minimum support increases, the elapsed time decreases.

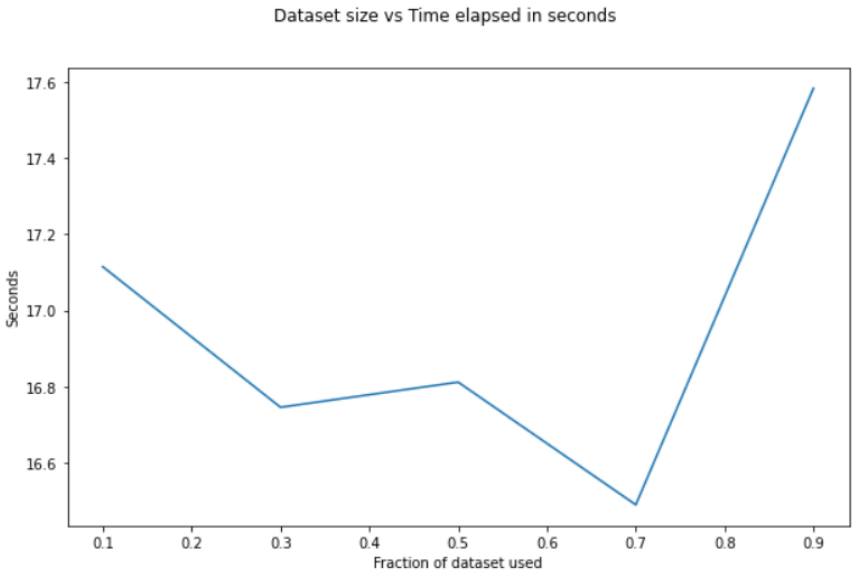


Figure 16: Python plot of figure 14

In figure 16, we see the plot of figure 14. As dataset gets larger, the time elapsed increases however the results are not stable with respect to the FP-growth. In FP-growth, both with respect to the minimum support and dataset size, the trend was clear and intuitive while in A-priori algorithm, in particular in figure 16 the result is not stable.

Conclusion

In this project two different algorithms have been used to find the frequent itemsets. In particular, we used PySpark implementation of FP-growth and mlxtend implementation of A-priori algortihm. Even though we can not really compare them because the dataset sizes were in different in two cases, the resulting frequent item tables were almost identical. A-priori algorithm was unstable with respect to the time it requires while FP-growth returned a more consistent result when it comes to scalability.