Data Science and Economics
Department of Economics, Management and Quantitative Methods
Department of Computer Science "Giovanni degli Antoni"
Università degli Studi di Milano

Market Basket Analysis

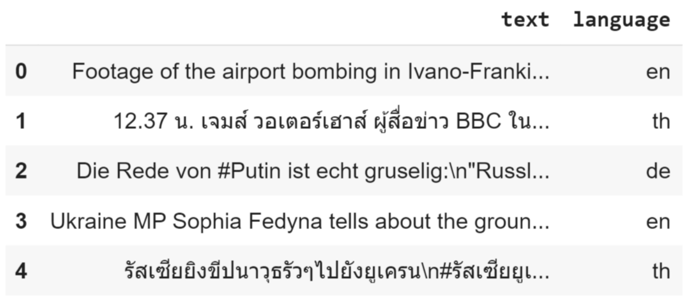
Algorithms for Massive Datasets
Shlyk Darya - 965162

a.y. 2021/2022

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 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.

1 Dataset

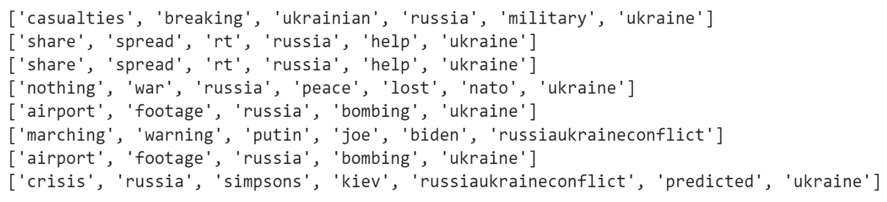
In the project we perform market basket analysis on the Ukraine Conflict Twitter dataset, published on Kaggle and released under the CC-BY-SA 4.0 license. This dataset contains 1.2M distinct tweets in different languages about the current ongoing Ukraine-Russia conflict. For the purpose of this project we selectively choose English to be the preferred language, which leaves us with a total of 720K tweets. In the framework of the MBA, the strings contained in the "text" column of the CSV file will be considered as baskets, while words will represent items.



	text	language
0	Footage of the airport bombing in Ivano-Franki...	en
1	12.37 น. เจมส์ วอเตอร์เฮาส์ ผู้สื่อข่าว BBC ใน...	th
2	Die Rede von #Putin ist echt gruselig:\n"Russl...	de
3	Ukraine MP Sophia Fedyna tells about the groun...	en
4	รัสเซียยิงขีปนาวุธรัวๆไปยังยูเครน\n#รัสเซียยู...	th

2 Data Handling

The processing phase is intended to transform the dataset into a format that would satisfy the algorithm requirements. In particular the algorithm expects to receive a list of tweets/baskets, each represented as a set of words/items. In the tokenization step, the raw tweet text is split into a set of atomic elements, or tokens. We import TweetTokenizer class from nltk library for natural language processing to perform the task. This tokenizer is designed to recognize tweet specific elements, such as emoji, hashtags, links and extract them as separate tokens. The cleaning phase follows the tokenization step, where we lower-case tokens, strip them of @ and #, filter out stopwords and keep strings made of alphabet characters only. The final result comprises a collection of around 720K baskets, with a small sample displayed below.



```
['casualties', 'breaking', 'ukrainian', 'russia', 'military', 'ukraine']
['share', 'spread', 'rt', 'russia', 'help', 'ukraine']
['share', 'spread', 'rt', 'russia', 'help', 'ukraine']
['nothing', 'war', 'russia', 'peace', 'lost', 'nato', 'ukraine']
['airport', 'footage', 'russia', 'bombing', 'ukraine']
['marching', 'warning', 'putin', 'joe', 'biden', 'russiaukraineconflict']
['airport', 'footage', 'russia', 'bombing', 'ukraine']
['crisis', 'russia', 'simpsons', 'kiev', 'russiaukraineconflict', 'predicted', 'ukraine']
```

3 Savasere, Omiecinski, and Navathe Algorithm

We implement the SON algorithm for mining frequent itemsets. This is an exact algorithm, that makes a total of two passes through the data. The idea is to distribute the computations, by splitting baskets file into chunks and process them in parallel in two stages. In the first phase, the algorithm generates a set of candidates by finding locally frequent itemsets in each partition. In the second stage, the support for all these candidates is computed at partition level and then aggregated, to determine which candidates are frequent in the whole dataset. The final output will comprise only itemsets whose global support exceeds the predefined threshold. The algorithm requires the user to specify the support threshold, expressed as a fraction of the total number of baskets.

4 Implementation

To find local frequent itemsets in stage one, we run the Apriori algorithm within each partition. A suitably written function implements the algorithm, making use of monotonicity property while generating candidate itemsets of higher cardinality. The function takes the minimum support as input and adjusts the chunk threshold to account for the size of the partition. The algorithm runs passes over baskets for each set-size, k, until no frequent itemset of a certain size is found. In the first pass, it counts all the items, and those items whose count is at least the adjusted support threshold form the set of frequent singletons. To find frequent itemsets of a given size k, as we read the baskets we only consider items that are frequent as singletons. From these items, the algorithm constructs k-size sets and only counts an itemset if all of its immediate subsets form the set of frequent itemsets of size k-1. After each pass, we update the set of frequent itemsets with candidate itemsets whose support exceeds the adjusted support threshold.

To execute the second phase of the SON algorithm, we design a different function, which takes a list of local candidates at input and simply counts their support at partition level. Later, the reduce task will gather the output from different partitions and will compare individual support with the threshold.

We execute the SON algorithm within MapReduce framework, relying on pyspark Resilient Distributed Dataset and its mapPartition() method, that takes a higher order function as an argument and applies it partition-wise. Our implementation of the SON algorithm makes use of two map-reduce jobs, one for each stage of the algorithm. In the first stage, mapPartition() is used to invoke A-priori to find frequent itemsets in each partition. In the latter job, mapPartition() triggers a validation pipeline to detect truly frequent itemsets among the local candidates. The final output of the algorithm consists of frequent itemsets and the corresponding support.

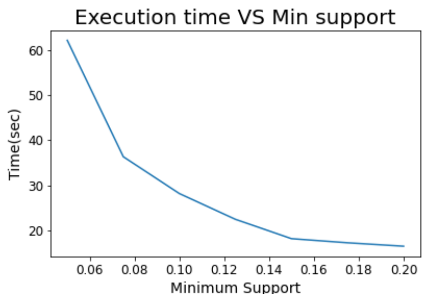
5 Experimental results

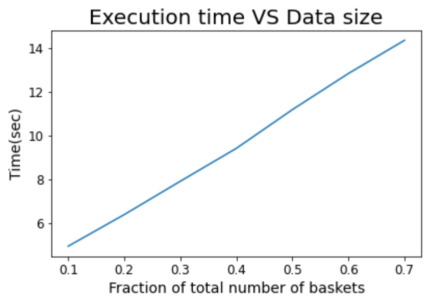
To execute the SON algorihtm on entire dataset, consisting of around 722 thousand baskets, we choose to distribute the workload over 4 partitions. It takes the algorithm around thirty seconds to find all the frequent itemsets when the support threshold is set to 10% of the total number of baskets. The final output produced by the SON algorithm is displayed below.

```
[(['russia', 'russian', 'ukraine'], 75580),
 (['russia', 'russian'], 82385),
 (['russia', 'ukraine'], 233309),
 (['russian', 'ukraine'], 157804),
 (['ukraine', 'war'], 79136),
 (['putin', 'ukraine'], 96320),
 (['people', 'ukraine'], 77476),
 (['ukraine', 'ukrainian'], 94407),
 (['russia'], 290464),
 (['russian'], 184662),
 (['ukraine'], 548232),
 (['war'], 101830),
 (['kyiv'], 91054),
 (['people'], 89893),
 (['ukrainian'], 110930),
 (['putin'], 137373)]
```

Next, we study the runtime of the algorithm as we let the support threshold vary from 0.05 to 0.2. (Figure: Left) As the threshold increases the number of candidate itemsets that SON generates at the first stage goes down. Thus, it takes the algorithm less time to terminate for higher values of the parameter.

Finally, we measure the execution time of the SON algorithm for increasing number of baskets, in order to understand how it scales up with the data size. We run the algorithm several times, starting with 10% of the basket file, and increasing the input size with additional 10% at each run, while keeping the support threshold fixed at 0.15. We can observe from the figure on the right, that the execution time grows proportionally with the size of the input data.


Execution time VS Min support
Time(sec)
Minimum Support


Execution time VS Data size
Time(sec)
Fraction of total number of baskets