Market Basket Analysis

1 Dataset

The dataset "Ukraine-Russian-Crisis-Twitter " is taken from the public source Kaggle. The data contains following attributes:

userid, username, acctdesc, location, following, followers, totaltweets, usercreatedts, tweetid, tweetcreatedts, retweetcount, text, hashtags, language, coordinates, favourite_count, extractedts

The dataset contains texts from the tweets about Ukraine-Russia crisis in different languages generated by different users.

Attributes required for analysis:

- text: text of the tweet
- language: language code of the text language

Total languages in the dataset: 61

am	Amharic	fa	Persian	lt	Lithuanian	sl	Slovenian
ar	Arabic	fi	Finnish	lv	Latvian	sr	Serbian
bg	Bulgarian	fr	French	ml	Malayalam	sv	Swedish
bn	Bengali	gu	Gujarati	mr	Marathi	ta	Tamil
ca	Catalan	hi	Hindi	my	Burmese	te	Telugu
ckb	Persian	ht	Haitian	ne	Nepali	th	Thai
cs	Czech	hu	Hungarian	nl	Dutch	tl	Tagalog
cy	Welsh	hy	Armenian	no	Norwegian	tr	Turkish
da	Danish	in	Indonesian	or	Oriya	uk	Ukrainian
de	German	is	Icelandic	pa	Punjabi	und	Undetermined
dv	Dhivehi	it	Italian	pl	Polish	ur	Urdu
el	Greek	iw	Hebrew	ps	Pashto	vi	Vietnamese
en	English	ja	Japanese	pt	Portuguese	zh	Chinese
es	Spanish	ka	Georgian	ro	Romanian		
et	Estonian	kn	Kannada	ru	Russian		
eu	Basque	ko	Korean	si	Sinhala		

2 Data Organisation

Downloading Ukraine-Russian-Tweets Dataset from Kaggle:

```
!kaggle datasets download bwandowando/ukraine-russian-crisis-twitter-dataset-1-2-m-rows
-unzip
```

The dataset contains 118 files. All the files are downloaded in Google Colab path: /content/

```
import gzip
import shutil

with gzip.open(/content/0401_UkraineCombinedTweetsDeduped.csv.gzip, rb)
as f_in:
        with open(/content/tweets.csv, wb) as f_out:
        shutil.copyfileobj(f_in, f_out)
```

Downloading Stopwords Dataset from Kaggle:

```
!kaggle datasets download manasasmurthy/stopwords -unzip
```

Loading the Stopwords data file:
Contains stopwords for all languages except for language codes - ht, am, ml, und

```
import pickle
with open(/content/STOPWORDS.pk,rb) as f:
        stopwords=pickle.load(f)
```

3 General Methodology

The objective of implementing the frequent pattern mining algorithms is to find the words that occur simultaneously in the text. The general ML approach includes:

- Data collection: Data is taken from public source Kaggle
- Data Cleaning and Preprocessing: Using Python' s PySpark module to clean the data
- Data Transformation: Using Transaction Encoder to transform the data into required format
- Data Mining: Implementing Apriori and FP-Growth Algorithms in Python using mlxtend module
- Evaluation and Interpretation: Analysing the results from Data Mining step

4 Data Preprocessing

The data is grouped based on a particular language selected and the selected data is sampled for faster computation and execution of the code. The data is cleaned using PySpark dataframe functionalities.

Since the text is natural language data, the data can contain stop words, emoticons/emojis, various fonts in upper/lower/toggle case, punctuations, symbols, links, etc. It is necessary to remove these noise and incorrect formatted data from the dataset before implementing the machine learning models.

Stop words- stop words are any word in a stop list which are filtered out before or after processing of natural language data (text)

Steps for data cleaning:

- converting entire text to lower-case
- removing punctuations and extra spaces from the text
- removing hyper-links and emojis from the text
- removing the stop words from the text

Using PySpark library:

- Spark DataFrame is distributed and hence processing in the Spark DataFrame is faster for a large amount of data.

5 Implementation of Algorithms

Market basket analysis (MBA) is a data mining technique for identifying patterns in any environment. MBA analyses the purchases that usually occur together. For example, people who buy bread and nutella also buy jelly. Or people who buy shampoo may buy conditioner.

Our data consists of natural language text. Each text is considered as a transaction and the unique words in a single text are considered as items. So we are trying to find the words that usually occur together in the text.

We will be using mlxtend python library for imlementing the algorithms. MLxtend (Machine Learning extensions) consists of many functions for data analysis and machine learning tasks.

Transaction Encoder converts item lists into transaction data for frequent itemset mining. It transforms the dataset into an array format suitable for typical machine learning APIs.

```
text_list=list(spark_df.select('text').toPandas()['text'])

te=TransactionEncoder()
te_array=te.fit(text_list).transform(text_list)
txn=pd.DataFrame(te_array,columns=te.columns_)
```

Terminology:

support:

- support metric is defined for itemsets
- support is used to measure frequency of an itemset in a database

confidence:

- confidence of a rule A->C is the probability of seeing the consequent in a transaction given that it also contains the antecedent

lift:

- lift metric is commonly used to measure how much more often the antecedent and consequent of a rule A->C occur together than we would expect if they were statistically independent
- if A and C are independent, the Lift score will be exactly 1.

leverage:

- leverage computes the difference between the observed frequency of A and C appearing together and the frequency that would be expected if A and C were independent
- leverage value of 0 indicates independence.

5.1 Apriori Algorithm

Apriori generates the frequent patterns by making the itemsets using pairing such as single item set, double itemset, triple itemset. Apriori uses candidate generation where frequent subsets are extended one item at a time.

It is a bottom-up and breadth first approach. Apriori's principle: If an itemset is frequent, then all of its subset must also be frequent. The support of an itemset never exceeds o of its subset support. This is known as anti-monotype property of support. The main idea of this algorithm is, it generates k-th candidate itemsets from the (k-1)-th frequent itemsets. It also finds the k-th frequent itemset from the k-th candidate itemsets. The algorithm gets terminated when the frequent itemsets cannot be extended further. The advantage is that multiple scans are generated for candidate sets. The disadvantage is that the execution time is more as wasted in producing candidates everytime, it also needs more search space and computational cost is too high.

Steps To Mine The Frequent Elements

1. Generate And Test:
   First find the 1-itemset frequent elements L1 by scanning the database and remove all the elements from C which don't satisfy the minimum support criteria.

2. Join Step:
   To attain the next level elements Ck joins the previous frequent elements by self join i.e Lk-1*Lk-1 known as Cartesian product of Lk-1 i.e this step generates new candidate k-itemsets based on joining Lk-1 with itself which is found in the previous iteration. Let Ck denote candidate k-itemsets and Lk be the frequent k-itemset.

3. Prune Test:
   Pruning eliminates some of the candidate kitemsets using the Apriori's principle. A scan of the database is used to determine the count of each candidate in Ck would result in the determination of Lk (i.e all the candidates having a count less than the minimum support count). Step 2 and 3 is repeated until no new candidate set is generated.

```
from mlxtend.frequent_patterns import apriori
from mlxtend.frequent_patterns import association_rules

freq_ap=apriori(txn, min_support=supp,max_len=2, use_colnames=True)
rules_apriori=association_rules(freq_ap,metric="confidence",
                                min_threshold=conf)
```

5.2 FP-Growth Algorithm

FP Growth generates an FP-Tree for making frequent patterns. FP-growth generates conditional FP-Tree for every item in the data.

FP Growth algorithm discovers the frequent itemset without the candidate generation. It follows two steps such as: In step one it builds a compact data structure called the FP-Tree, in step two it directly extracs the frequent itemsets from the FP-Tree. The advantage is that it constructs conditional pattern base from database which satisfies minimum support, due to compact structure and no candidate generation it requires less memory. The disadvantage is that it performs badly with long pattern data sets.

FP-Tree:
FP-Tree represents all the relevant frequent information from a data set due to its compact structure. Each and every path of FP-Tree represents a frequent itemset and nodes in the path are arranged in a decreasing order of the frequency. The great advantage of FP-Tree is that all the overlapping itemsets share the same prefix path. Because of this the information of the data set is highly compressed. It scans the database only twice and it does not need any candidate generation. FP-Tree is constructed using 2 passes over the data set.

Pass1:
It first scans the data and then find support for each item. Then it discards the infrequent itemsets and sort the frequent itemsets in decreasing order based on their support.

Pass2:
Nodes corresponds to itemset and have a counter.

- It first reads 1 transaction at a time and maps it to a path.
- Fixed order is used so paths can overlap when transaction shares items (when they have same prefix) I this case, counters are incremented.
- Pointers are maintained between nodes containing the same item, resulting a linked list (dotted lines). The compression will be high based on the more paths that overlap. FP-Tree may fit in the memory.
- Frequent itemsets are extracted from the FP-Tree.

```
from mlxtend.frequent_patterns import fpgrowth
from mlxtend.frequent_patterns import association_rules

freq_fp=fpgrowth(txn, min_support=supp,max_len=2, use_colnames=True)
rules_fp=association_rules(freq_fp,metric="confidence",
                                  min_threshold=conf)
```

6 Results

6.1 Word Frequency

Frequency of each word in the data:

Language=Italian:

word	count
putin	1461
russia	860
ucraina	792
guerra	666
biden	507
gas	352
mariupol	320
russo	265
draghi	256
santoro	255
zelensky	243
rubli	238
ukrainerussiawar	201
russi	197
ucraino	189
guerraucraina	172
mosca	165
aprile	159
drittoerovescio	158
piazzapulitala	149

only showing top 20 rows

Language=Russian:

word	count
ukraine	1236
russianukrainianwar	780
ukrainewar	738
украина	690
ukrainerussianwar	634
stoprussia	599
вторжениероссии	369
войнасукраиной	369
агрессияроссии	362
всу	342
войнапутина	340
украине	315
russia	290
белгород	289
russiaukrainewar	252
оккупантов	235
standwithukraine	218
российские	197
ukrainerussiawar	187
news	184

only showing top 20 rows

6.2 Frequent Itemset

Hyperparameters for the algorithms: [language=Italian]

min support=0.1
min confidence=0.3

	support	itemsets
0	0.1720	(biden)
1	0.1290	(gas)
2	0.2490	(guerra)
3	0.1160	(mariupol)
4	0.4800	(putin)
5	0.3140	(russia)
6	0.2820	(ucraina)
7	0.1160	(biden, putin)
8	0.1415	(guerra, putin)
9	0.1285	(guerra, russia)
10	0.1380	(guerra, ucraina)
11	0.1355	(putin, russia)
12	0.1300	(putin, ucraina)
13	0.1495	(ucraina, russia)

An itemset is considered as ”frequent” if it meets a user-specified support threshold. For instance, if the support threshold is set to 0.5 (50%), a frequent itemset is defined as a set of items that occur together in at least 50% of all transactions in the database

Here,

(biden) occurs in at least 17.20% of the data

itemset	support
(biden)	0.1720

(biden and putin) occurs together in at least 11.60% of the data

itemset	support
(biden,putin)	0.1160

6.3 Association Rules

	antecedents	consequents	antecedent support	consequent support	support	confidence	lift	leverage	conviction
0	(biden)	(putin)	0.172	0.480	0.1160	0.674419	1.405039	0.033440	1.597143
1	(guerra)	(putin)	0.249	0.480	0.1415	0.568273	1.183902	0.021980	1.204465
2	(guerra)	(russia)	0.249	0.314	0.1285	0.516064	1.643517	0.050314	1.417544
3	(russia)	(guerra)	0.314	0.249	0.1285	0.409236	1.643517	0.050314	1.271235
4	(guerra)	(ucraina)	0.249	0.282	0.1380	0.554217	1.965308	0.067782	1.610649
5	(ucraina)	(guerra)	0.282	0.249	0.1380	0.489362	1.965308	0.067782	1.470708
6	(russia)	(putin)	0.314	0.480	0.1355	0.431529	0.899018	-0.015220	0.914734
7	(ucraina)	(putin)	0.282	0.480	0.1300	0.460993	0.960402	-0.005360	0.964737
8	(ucraina)	(russia)	0.282	0.314	0.1495	0.530142	1.688350	0.060952	1.460015
9	(russia)	(ucraina)	0.314	0.282	0.1495	0.476115	1.688350	0.060952	1.370529

Rule:

antecedents	(biden)
consequents	(putin)
support	0.1160
confidence	0.6744
lift	1.405

Interpretation:

support = 0.1160 means - 11.6% of the text contain antecedents and consequents combination

confidence = 0.6744 means - of the text containing (biden), 67.44% of the text is likely to contain the word (putin) as well

6.4 Apriori v/s FP-Growth

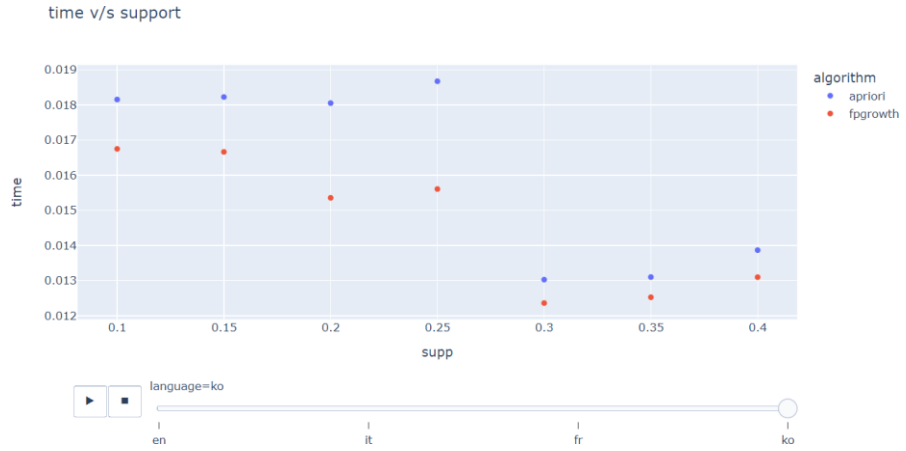
Both Apriori and FP Growth algorithm are used to mine the frequent patterns from database. Both the algorithm uses some technique to discover the frequent patterns. Apriori algorithm works well with large database but FP Growth algorithm works badly with large database. The comparisons between both the algorithms based on technique, memory utilization, number of scans and time consumed are given below:

- Technique:
  Apriori algorithm uses Apriori property and join, pure property for mining frequent patterns. FP Growth algorithm constructs conditional pattern free and conditional pattern base from the database which satisfies the minimum support.
- Search Type:
  Apriori uses breadth first search method and FP Growth uses divide and conquer method.
- Memory Utilization:
  Apriori algorithm requires large memory space as they deal with large number of candidate itemset generation. FP Growth algorithm requires less memory due to its compact structure they discover the frequent itemsets without candidate itemset generation.
- Number of Scans: Apriori algorithm performs multiple scans for generating candidate set. FP Growth algorithm scans the database only twice.
- Time: In Apriori algorithm execution time is more wasted in producing candidates every time. FP Growth' s execution time is less when compared to Apriori. But FP Growth is slower when the pattern length is long.

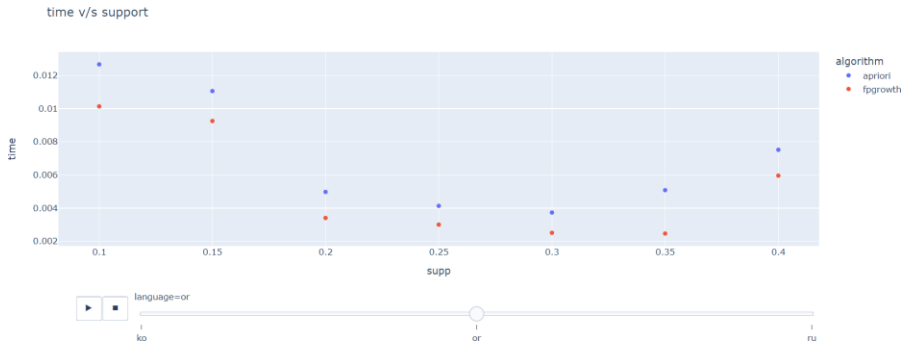
Case 1: Short Pattern Datasets

FP-Growth takes less execution time than Apriori algorithm

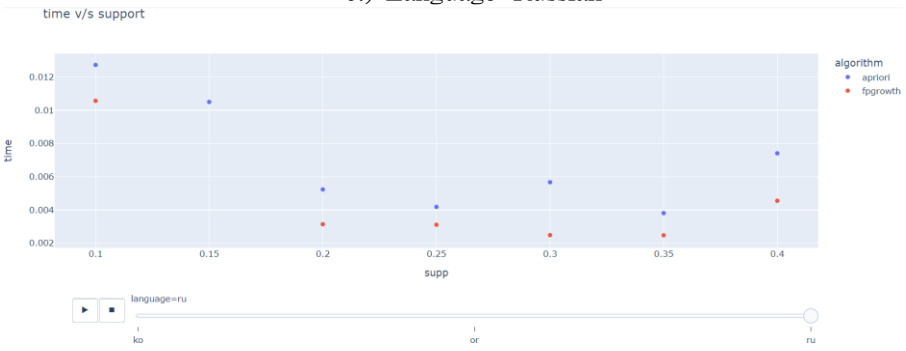
a.) Language=Korean



b.) Language=Oriya



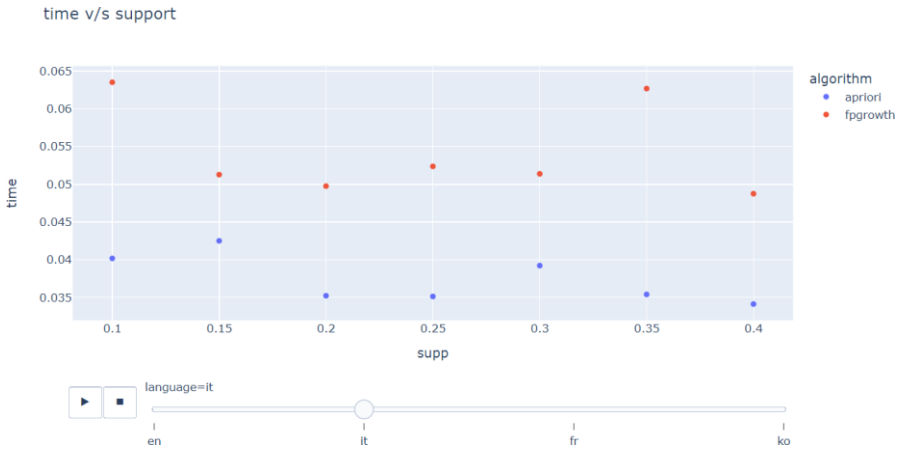
c.) Language=Russian



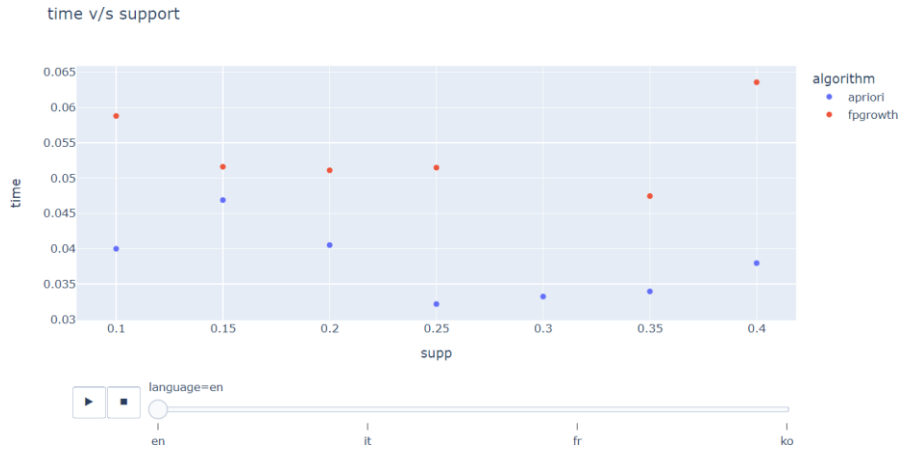
Case 2: Long Pattern Datasets

FP-Growth takes more execution time than Apriori algorithm

a.) Language=Italian



b.) Language=English



7 Conclusion

The purpose of this project is to apply Frequrnt Pattern Mining algorithms (Apriori and FP-Growth) on textual dataset "Ukraine-Russian-Crisis-Tweets" dataset taken from Kaggle. The dataset contains tweets generated by different users in multiple languages. Since the data is in textual format, removing the stopwords and other noise from the data is crucial step. After the cleaning of the data, the data is transformed into TransactionEncoder dataframe and the models are applied. Both the models generate frequent itemsets and association rules for different threshold values.

Scalability and Replicability of the approach:

The pre-processing is done using PySpark module. Spark DataFrame is distributed and hence processing in the Spark DataFrame is faster for a large amount of data The algorithms implemented can work with large datasets. The adopted approach will produce the same outcome if the process is repeated.

Important Result:

- FP-Growth algorithm is generally faster than apriori algorithm because Apriori algorithm scans the database in each of its steps it becomes time-consuming for data where the number of items is larger whereas FP-tree requires only one scan of the database in its beginning steps so it consumes less time. But FP-Growth runs poorly when the pattern length is too long.