AMD PROJECT

MARKET BASKET ANALYSIS-UKRAINE CONFLICT

ANJALI RAJAGOPAL 963277

Declaration

"I declare that this material, which I/We now submit for assessment, is entirely my/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 my/our work. I/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 me/us or any other person for assessment on this or any other course of study"

Contents

1 Data Setup

1.1 Data Download

The first step is to obtain the dataset for "Ukraine Conflict Twitter". The direct download option through the command line is available on Kaggle's website. The command Kaggle datasets download requires the precise name of the item to be downloaded and the person's login credentials.

```
# The below code can be used to download data from Kaggle into the Direcotry ad /content/Kaggle:


path = '/content/Kaggle'

if not os.path.exists(path):
  # Create a new directory because it does not exist
  os.makedirs(path)

%cd /content/Kaggle/


os.environ['KAGGLE_USERNAME'] = "anjalirajagopal" # username from the json file
os.environ['KAGGLE_KEY'] = "dd4b25d5edbb7a9dbffd6713ddde2808" # key from the json file
os.environ['KAGGLE_CONFIG_DIR'] = "/content/Kaggle"


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

1.2 Data File Extension Change

In order to read the downloaded data directly into PySpark, we needed to change the file extension of the downloaded files from ".gzip" to ".gz". The below code performs this operation.

```
#files = glob('./*.gzip')
files = glob('/content/Kaggle/UkraineWar/UkraineWar/*.gzip')
# files = glob('./Kaggle/UkraineWar/UkraineWar/*.gzip')
for file in files:
  os.rename(file, '.'.join(file.split('.')[0:2]+['gz']))
```

1.3 Data Description

The dataset "Ukraine Conflict Twitter Dataset" originates from a publicly accessible source, specifically Kaggle It is the raw tweets extracted from Twitter. Each row in this dataset corresponds to a tweet and its corresponding attributes. The given dataset contains 17 columns and over 100 million rows, which is increasing day-by-day. I typically use five columns that are certain to be relevant.

- Userid: A unique identifier
- Retweetcount: How many times that tweet has been retweeted by others
- Text: Actual raw tweets that people have posted

- Language: The language in which the tweet is expressed.

The "Ukraine Conflict Twitter Dataset" contains over 100 million tweets arranged in the manner indicated above; this indicates that the outcome of any operation involving a combination of sentences cannot be stored in the memory of an average-sized machine. This difficulty can be solved by employing scalable algorithms designed for large datasets.

```
# Selecting file and sampling data (using only 0.5% of one day's data)
df = (
    spark
    .read
    .option("multiLine", 'true')
    .option("escape", "\"")
    .csv("/content/Kaggle/UkraineWar/UkraineWar/0401_UkraineCombinedTweetsDeduped.csv.gz", header=True)
    .select(['userid','language','retweetcount','text']))

df.show(5)
```

```
+------------------+--------+------------+--------------------+
|            userid|language|retweetcount|                text|
+------------------+--------+------------+--------------------+
|          16882774|      en|        3412|⚡The Ukrainian Ai...|
|        3205296069|      en|         100|Chernihiv oblast....|
|1235940869812809728|      en|           9|America 🇺🇸 is p...|
|1347985375566966784|      en|         573|JUST IN: #Anonymo...|
|1505394816636846083|      en|         190|***PUBLIC MINT NO...|
+------------------+--------+------------+--------------------+
only showing top 5 rows

time: 9.28 s (started: 2022-09-05 18:03:01 +00:00)
```

1.4 Data Filtering

Inorder to get useful insights out of the data, I set up additional filters on the dataset.

1. All tweets that were not in English were removed from the dataset.
2. Tweets having a retweetcount" below the median "retweetcount" of the dataset were eliminated.
3. All duplicate tweets were eliminated based on duplicate "userid" and "text" columns.
4. All records where the "text" or the "language" was missing were eliminated.
5. To test any code that performs operations on this dataset while circumventing the memory issues for an average machine (in an acceptable amount of time) and to allow for iterative experimentation, a subset of *0.5 %* of one day's data of tweets after applying the above filters is chosen and processed as if it were a large dataset. However, the code is written with scalability in mind.

```
# Filtering data which is only in English language, lies in the above 50% percentile range of retweet count, should

df = df.withColumn("retweetcount", df["retweetcount"].cast(FloatType()))
only_text = (
            df
            .dropna(subset = ['text','language'])
            .filter((df.language == "en") & (df.retweetcount >= df.approxQuantile("retweetcount", [0.5], 0.25)[0]))
            .dropDuplicates(["userid","text"])
            .select("text")
            .sample(fraction = 0.005,seed = 2022)) # Change the Fraction Argument to take different sample sizes

only_text.show(5)
```

```
+--------------------+
|                text|
+--------------------+
|#Russia #Ukraine ...|
|#Mariupol April 1...|
|⚡The Ukrainian Ai...|
|My government dec...|
|Courage, strength...|
+--------------------+
only showing top 5 rows

time: 12.7 s (started: 2022-09-05 18:03:10 +00:00)
```

2 Text Pre-processing

To be able to work on the dataset in an efficient and precise manner, proper pre-processing is essential In this project, in particular, it is required to operate and make judgments as if the dataset were a big dataset.

When dealing with unstructured text data, pre-processing becomes even more important as text data comes with inherent noise and unwanted information that can obscure the actual information of value. In order to preserve and enrich the value of the data, a plethora of pre-processing techniques are applied to the text data. Below I will describe the ones I used.

2.1 Tokenisation

Tokenisation means splitting a whole sentence into words. Every word is tokenised here and is treated as such in subsequent pre-processing. The following are some special features provided by tokenisation that cannot be performed by standard word separator(s):
Splitting the abbreviations with separators like:

- Understand words of various languages, such as German or French
- Understands compound words in different languages
- Tokenisation has the added benefit of creating the baskets of items (tokens) that we will require for generating frequent itemsets.

2.2 Normalisation

Text normalisation is the process of transforming a text into a canonical (standard) form. For example, the words "gooood" and "gud" can be transformed into "good," its canonical form. In our code, we also convert all tokens into lowercase during normalisation.

2.3 Lemmatization

As derived from the name "lemma", lemmatisation normalises the word to its original, dictionary form. Since lemmatisation uses a pre-defined dictionary, it is slower than stemming. Stemming is another type of normalisation technique, with an inferior quality of performance when compared to lemmatization.
Hence, results from lemmatisation make much more sense and, hence, are more useful.

2.4 StopWords Cleaner

In any language, just as English there are some helping words that don't carry much meaning in itself, but are used in sentence formation. For example, "a", "our", "for", "in", etc. By eliminating these words the computer can focus on more important words that carry weight, making the analysis and therefore the insights more useful. I have also allowed a placeholder in the code where custom stop words can be added as per the need.

2.5 nGram Generation

A single word by itself often does not carry much meaning. By taking combinations of words together, we can often extract more semantic information out of the text. This process is called "nGram Generation." While I only used monograms for our analysis, the code I have written will allow for defining the number of tokens which can be considered together as ngrams.

2.6 Final Data Filtering

I then filtered the output to get only unique tokens in each tweet. Unique data is then filtered again based on the unique token counts. The data was distributed according to the unique token count into a normal bell-shaped curve and the tweets having only 1 unique token or above 75 percentiles were taken for further computation.

2.7 Summary

In summary, by implementing the above text pre-processing techniques, we are now set up to execute our market basket analysis in order to find frequent itemsets and association rules.

Tokenisation converted the tweets in the dataset into baskets of tokens, while the other techniques removed unwanted noise from the dataset while also enriching it with more semantic information.

```
+--------------------+--------------------+--------------------+----------+
|                text|     finished_ngrams|              unique|word_count|
+--------------------+--------------------+--------------------+----------+
|#Russia #Ukraine ...|[recent, photo, f...|[recent, photo, f...|        23|
|#Mariupol April 1...|[mariupol, april,...|[mariupol, april,...|         5|
|⚡The Ukrainian Ai...|[air, force, like...|[air, force, like...|        19|
|My government dec...|[government, deci...|[government, deci...|        19|
|Courage, strength...|[courage, strengt...|[courage, strengt...|         9|
+--------------------+--------------------+--------------------+----------+
only showing top 5 rows

time: 40 s (started: 2022-09-05 18:03:23 +00:00)
```

2.8 Apriori algorithm

The Apriori approach is used to discover patterns of associations between one or more items in a dataset, and it assumes that any subset of a frequent itemset must also be frequent. The below three parameters can be used to determine the significance of an association rule:

1. Support: The percentage of transactions containing a specific combination of items compared to the overall number of transactions in the database

2. Confidence: A conditional probability measure. If the item on the left-hand side (antecedent) is bought, the item on the right-hand side (consequent) will be bought as well.
   Confidence(antecedent ⇒ consequent) = (Transactions involving both antecedent and consequent) / (Transactions involving only antecedent)

3. Lift: The chance of all items occurring together divided by the product of antecedent and consequent occurring as if they were independent of one another is known as lift.
   Lift(antecedent ⇒ consequent) = Confidence(antecedent, consequent) / Support(consequent)
   As a result, the likelihood of a consumer acquiring both the antecedent and the consequent together is 'lift-value' times greater than the likelihood of purchasing both separately.

   - Lift (antecedent ⇒ consequent) = 1 means that there is no correlation within the itemset.
   - Lift (antecedent ⇒ consequent) > 1 means that there is a positive correlation within the itemset, i.e., products in the itemset, antecedent, and consequent, are more likely to be bought together.
   - Lift (antecedent ⇒ consequent) < 1 means that there is a negative correlation within the itemset, i.e., products in itemset, antecedent, and consequent, are unlikely to be bought together

2.9 Helper Functions

2.9.1 Generate Dictionary from Items

This function creates a numeric encoding for all unique items present in the provided list of transactions. This numeric encoding is primarily used for efficient operations as it reduces the memory footprint of the collections.

```
def generate_dictionary_from_items(transactionList):
    uniqueTokens = transactionList.flatMap(lambda x: x).distinct()
    dictionary = dict(zip(uniqueTokens.collect(), range(len(uniqueTokens.collect()))))
    reverse_dictionary = {v: k for k, v in dictionary.items()}
    return dictionary, reverse_dictionary
```

2.9.2 Encode Items

This function encodes a collection of items to their numeric counterparts as defined in the dictionary created using the function (generate_dictionary_from_items)

```
def encode_items(items, dictionary):

    return [dictionary[item] for item in items]
```

2.9.3 Decode Items

This function decodes a collection of encoded items to their string counterparts as defined in the dictionary created using the function (generate_dictionary_from_items)

```
def decode_items(encoded_items, dictionary):

   return [dictionary[encoded_item] for encoded_item in encoded_items]
```

2.9.4 Filter By Min Count

This function calculates frequency counts for items in a PySpark RDD and filters the RDD using a minimum count threshold

```
def filter_by_min_count(items, minCountThresh):
    filtered_items = (
        items.map(lambda x: (x, 1))
        .reduceByKey(lambda a, b: a + b)
        .filter(lambda item: item[1] >= minCountThresh)
    )
    return filtered_items
```

2.9.5 Update Frequent Items Table

This function creates a 1-dimensional numpy array with a size equal to the number of unique items (based on the index dictionary created using (generate_dictionary_from_items)) with item indexes (based on the index dictionary created using (generate_dictionary_from_items)) as the index and assigns a numeric value as the value. The numeric value is set to 0 for an item if the item is not considered to be frequent.

```
def update_frequent_items_table(frequent_items, index_dictionary_size):
    frequent_itemsets = np.zeros(index_dictionary_size)
    i = 0
    for itemset in frequent_items.collect():
        if isinstance(itemset[0], tuple):
            for item in itemset[0]:
                i = i + 1
                frequent_itemsets[item] = i
        else:
            i = i + 1
            frequent_itemsets[itemset[0]] = i
    return frequent_itemsets
```

2.9.6 Get Frequent Item Sets

This function finds all the frequent items and then checks if the count if the value is less than the threshold and then filters with it to find the frequent items

```
def get_frequent_item_sets(
    transactionList, itemset_size, minSupport, frequent_tokens_table
):
    frequent_itemsets = (
        transactionList.map(lambda x: filter_frequent(x, frequent_tokens_table))
        .filter(lambda x: len(x) > 1)
        .flatMap(lambda x: combinations(x, itemset_size))
        .map(lambda x: (x, 1))
        .reduceByKey(lambda x1, x2: x1 + x2)
        .filter(lambda x: x[1] > minSupport)
    )
    return frequent_itemsets
```

2.9.7 Get Frequent Item Sets

This function filters an iterable, keeping only the items that are considered frequent, as per the array created using (update_frequent_items_table)

```
def filter_frequent(items, frequent_items_table):
    return [item for item in items if frequent_items_table[item] != 0]
```

2.9.8 Generate rules from itemset

It returns a list of dictionaries containing the association rule by generating the rules for antecedent and precedent for the itemset pairs.

```
def generate_rules_from_itemset(itemset, frequent_items, min_conf, transaction_count):
    rules = []
    k = len(itemset[0]) # for itemset ((1,2),10) k = 2
    if k >= 2:
        subsets = list(combinations(itemset[0], k - 1)) # combinations((1,2),1) = [(1),(2)]
        support = itemset[1] # for itemset ((1,2),10) support = 10
        for antecedent in subsets:
            antecedent_tuple = [items for items in frequent_items if items[0] == antecedent][0]
            antecedent_support = antecedent_tuple[1]
            confidence = float("{0:.2f}".format(support / antecedent_support))
            if confidence >= min_conf:
                consequent = tuple(
                    filter(lambda x: x not in antecedent, itemset[0])
                )
                rule = {
                    "antecedent": antecedent,
                    "consequent": consequent,
                    "confidence": confidence,
                    "support": float("{0:.3f}".format(support / transaction_count)),
                }
                rules.append(rule)

    return rules
```

2.10 FP Growth

2.10.1 Introduction

FP GROWTH (Frequent Pattern Growth): The FP growth algorithm represents the database as a tree known as a frequent pattern tree or FP tree.

The association between the itemsets will be maintained by this tree structure. The database is fragmented by a common item. This fragmented part is referred to as a "pattern fragment." These fragmented patterns' itemsets are examined.

As a result of this method, the search for frequent itemsets is significantly reduced.

2.10.2 FP TREE

The Frequent Pattern Tree is a tree-like structure created from the database's first itemsets. The FP tree's objective is to find the most common pattern. Each item in the itemset is represented by a node in the FP tree.

Null is represented by the root node, while itemsets are represented by the lower nodes. While forming the tree, the association of the nodes with the lower nodes, that is, the itemsets with the other itemsets, is maintained.

We can use the frequent pattern growth method to locate the frequent pattern

without having to generate candidates. Let's have a look at the stages involved in mining a frequent pattern using the frequent pattern growth algorithm:

1. The first step is to search the database for instances of the itemsets. Support count or frequency of 1-itemset refers to the number of 1-itemsets in the database.

2. The FP tree is built in the second stage. To do so, start by making the tree's root. Null is used to represent the root.

3. Scanning the database and examining the transactions is the next stage. Examine the first transaction to determine the itemset included therein. The highest-counting itemset is at the top, followed by the next-lowest-counting itemset, and so on. It signifies that the tree's branch is made up of transaction itemsets arranged in descending order of count.

4. The database's next transaction is reviewed. The itemsets are sorted by count in ascending order. This transaction branch would share a common prefix to the root if any itemset of this transaction is already present in another branch (for example, in the first transaction). This means that in this transaction, the common itemset is linked to the new node of another itemset.

5. The count of the itemset is also incremented as transactions occur. As nodes are formed and linked according to transactions, the count of both the common node and new node increases by one.

6. The next step is to mine the FP Tree that has been generated. The lowest node, as well as the links between the lowest nodes, are evaluated first. The frequency pattern length 1 is represented by the lowest node. From there, follow the FP Tree's course. A conditional pattern base is the name given to this path or paths. The conditional pattern base is a sub-database that contains prefix pathways in the FP tree that start at the lowest node (suffix).

7. Create a Conditional FP Tree based on the number of itemsets in the path. The Conditional FP Tree considers the itemsets that meet the threshold support.

8. The Conditional FP Tree generates Frequent Patterns.

2.11 Experiments

In this section,I applied the Apriori Algorithm implementation I have developed to the dataset as described in the sections before. The code tests out the algorithm in a variety of parameter combinations. It takes 3 separate minimum support values, 3 separate minimum confidence values, and 2 different max item sizes to evaluate the drivers of performance. Subsequently, I also ran a comparison between my implementation of the Apriori algorithm and the highly scalable FP-Growth implementation included with PySpark.

2.12 Apriori Algorithm with key performance drivers

```
supports_list = [0.01,0.025,0.05]
conf_list = [0.05,0.1,0.2]
itemset_size_list = [2,3]

times = []
n_itemsets = []
n_rules = []
params = list(product(supports_list, conf_list, itemset_size_list))

def run_experiment(param,transactionList):
  result = namedtuple('result','params time n_itemsets n_rules')
  start = time.time()
  rules, itemsets = applyApriori(transactionList = transactionList,
                                 max_itemset_size = param[2],
                                 minSupport = param[0],
                                 min_conf = param[1])
  end = time.time()
  times = end-start
  n_itemsets = len(itemsets.collect())
  n_rules = len(rules)
  return result(param, times, n_itemsets, n_rules)


results = Parallel(n_jobs=cpu_count() - 1)(delayed(run_experiment)(param, transactionList) for param in params)

df = pd.DataFrame.from_records(
   results,
   columns=['params','time','n_itemsets', 'n_rules']
)
```

	minSupport	minConfidence	maxItemSets	time	n_itemsets	n_rules
0	0.010	0.05	2	19.759132	2434	2043
1	0.010	0.05	3	49.128168	12539	12148
2	0.010	0.10	2	11.240982	2434	2042
3	0.010	0.10	3	54.488224	12539	12147
4	0.010	0.20	2	10.169199	2434	2032
5	0.010	0.20	3	48.280488	12539	12137
6	0.025	0.05	2	7.780003	360	232
7	0.025	0.05	3	13.044376	1419	1291
8	0.025	0.10	2	10.054004	360	232
9	0.025	0.10	3	10.340884	1419	1291
10	0.025	0.20	2	7.005145	360	232
11	0.025	0.20	3	10.175245	1419	1291
12	0.050	0.05	2	7.580012	202	171
13	0.050	0.05	3	9.477848	1171	1140
14	0.050	0.10	2	7.389149	202	171
15	0.050	0.10	3	9.493550	1171	1140
16	0.050	0.20	2	7.574097	202	171
17	0.050	0.20	3	9.046215	1171	1140

```
time: 5min 3s (started: 2022-09-03 12:46:32 +00:00)
```

2.13 Apriori vs FP Growth Performance

In this experiment I run both the Apriori and FP Growth algorithms with *Maximum Itemset Size = 2*, *Minimum Support = 0.05* and *Minimum Confidence = 0.05* and compare their performances. Its found that Apriori takes less time as compared to FP growth

Apriori

```
apriori_experiment_1_rules, apriori_experiment_1_itemsets,  = applyApriori(transactionList = transactionList,
                                                                           max_itemset_size = 2,
                                                                           minSupport = 0.05,
                                                                           min_conf = 0.05)
apriori_experiment_1_rules = pd.DataFrame(list(chain.from_iterable(apriori_experiment_1_rules)))
apriori_experiment_1_rules.to_csv('apriori_experiment_1_rules.csv',index = False)
```

```
time: 7.09 s (started: 2022-05-03 12:51:53 +00:00)
```

	antecedent	consequent	confidence	support
0	(air,)	(address,)	0.49	0.052
1	(address,)	(air,)	0.89	0.052
2	(air,)	(misinformation,)	0.49	0.052
3	(misinformation,)	(air,)	1.00	0.052
4	(air,)	(mullgirl,)	0.49	0.052
...	...	...	...	...
337	(protectuesky,)	(silly,)	1.00	0.052
338	(protectuesky,)	(stoprussia,)	1.00	0.052
339	(stoprussia,)	(protectuesky,)	0.62	0.052
340	(protectuesky,)	(ukraineunderattack,)	1.00	0.052
341	(ukraineunderattack,)	(protectuesky,)	1.00	0.052

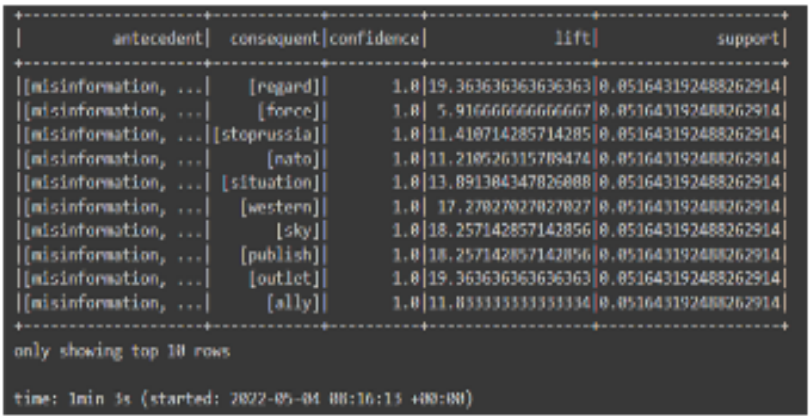
342 rows × 4 columns

time: 7.15 s (started: 2022-05-04 08:16:05 +00:00)

FP Growth

```
fpGrowth = FPGrowth(itemsCol="unique", minSupport=0.05, minConfidence=0.05)
model = fpGrowth.fit(result)

items = model.freqItemsets
rules = model.associationRules
rules.show(10)
```



antecedent	consequent	confidence	lift	support
[misinformation, ...]	[regard]	1.0	19.363636363636363	0.051643192488262914
[misinformation, ...]	[force]	1.0	5.916666666666667	0.051643192488262914
[misinformation, ...]	[stoprussia]	1.0	11.410714285714285	0.051643192488262914
[misinformation, ...]	[nato]	1.0	11.210526315789474	0.051643192488262914
[misinformation, ...]	[situation]	1.0	13.891304347826088	0.051643192488262914
[misinformation, ...]	[western]	1.0	17.270270270270272	0.051643192488262914
[misinformation, ...]	[sky]	1.0	18.257142857142856	0.051643192488262914
[misinformation, ...]	[publish]	1.0	18.257142857142856	0.051643192488262914
[misinformation, ...]	[outlet]	1.0	19.363636363636363	0.051643192488262914
[misinformation, ...]	[ally]	1.0	11.833333333333334	0.051643192488262914

only showing top 10 rows

time: 1min 1s (started: 2022-05-04 08:16:15 +00:00)

3 Scalability

Big data analysis and machine learning frameworks, as well as applications that need to analyse very massive and real-time data from data repositories, social media, sensor networks, smartphones, and the Web, all require scalability. Today, scalable big data analysis can be accomplished through parallel implementations that take advantage of the computing and storage capabilities of high-performance computing (HPC) systems and clouds, whereas Exascale systems will be used to implement extreme-scale data analysis in the near future.

The primary issues to be addressed and overcome for deploying innovative data analysis applications on Exascale systems are identified and investigated here, as well as how clouds currently enable the development of scalable data mining solutions.

The code used in this project was written specifically with scalability in mind. The code uses the highly efficient and scalable functionalities provided by the PySpark Framework to achieve this. All data involved in large scale operations is stored in PySpark data structures like RDDs and Spark Dataframes, which allow for storing the data partitioned across the multiple nodes of the cluster and also allow them to do processing in parallel.

I have also made use of the PySpark functions like map, flatMap, and reduceByKey, amongst others, that perform these operations on RDDs and DataFrames.

Another big factor that increases the scalability of my solution is text preprocessing. During this phase, I removed a lot of irrelevant information from the data, such as stopwords, non-standard word forms, and other noise from the data, thereby reducing the memory footprint of the data while preserving the information value. For text pre-processing, I have utilised the functionality offered by the SparkNLP library, which again works on the RDDs and DataFrames to perform text pre-processing in a highly efficient manner.

Wherever the implementation does not use PySpark data structures, the code

utilises parallel processing techniques to maintain scalability.

Lastly, Apriori as an algorithm works iteratively and for each iteration eliminates non-frequent itemsets. This reduces the memory footprint and allows for large-scale analysis. I also used the pre-implemented FP_Growth algorithm from the PySpark library for comparison, and I discovered that Apriori is faster than the pre-implemented one on my dataset. One limitation with FP-Growth is that the FP Tree has to be stored entirely in memory during the entire process. This leads to severe memory issues and can thus be a bottleneck. No such issues were observed with the Apriori algorithm.

My experiments showed that the maximum item size being considered is the primary driver of performance. Calculating frequent itemsets with large item sizes is computationally expensive, but even more expensive is mining rules from them as the number of distinct itemsets explodes.

Through clever use of Spark functionalities, text pre-processing, and parallel processing, along with the nature of Apriori, I believe that my code is highly scalable and should not have trouble analysing large datasets on suitably capable machines.

4 Discussion

The major goal of this research is to apply market basket analysis to a textual dataset. I have applied both the "Apriori algorithm" and the "FP Growth algorithm" to the "Ukraine Conflict Twitter" data set and have used both self-written functions and already installed libraries to compare the performance.

I have selected the "Ukraine Conflict Twitter" dataset to perform my analysis, which can be downloaded directly from Kaggle using an individual API and can be stored both on a personal drive and on Google Collab.

"Ukraine Conflict Twitter" is a very large dataset and consists of tweets from all regions, written in many different languages. To perform the analysis, I have selected only 1 day's data and to make it understandable, I have chosen data that was written only in English language and have used many filters to further simplify the data. Finally, I have taken a sample comprising of *0.5%* of filtered data and run the algorithm.

Most important part of any data analysis is text cleaning and pre-processing. This step involved running text through many pre-processing steps including tokenization, lemmatization, normalization and stopword cleanser. This is crucial as this step removes all unwanted noises and only keep the most relevant data.

Post-text pre-processing I ran the experiments that are mentioned above and recorded the results. Below are some of my observations:- formalised paraphrase On reducing the minimum support threshold for the algorithm, we observe an increase in the number of itemsets considered. This in turn leads to an increase in the number of rules mined from the dataset. Obviously, for mining this increased number of itemsets and association rules, the computation time involved

also increases.

- On increasing the maximum itemset size, we observe an increase in the number of itemsets considered. This in turn leads to an increase in the number of rules mined from the dataset. Obviously, for mining this increased number of itemsets and association rules, the computation time involved also increases.
- Modifying the minimum confidence threshold should not affect the computation time or the number of itemsets generated. It should, however, affect the number of rules generated. In this case, I did not observe a significant change in the number of rules mined even after increasing the confidence threshold. This suggests that most of the rules mined are of high confidence and thus are not affected by a small confidence threshold increase.
- The Apriori algorithm performed faster as compared to the pre-implemented FP_Growth algorithm, even though FP_Growth is supposed to be faster and more efficient. It might be because of the small sample size taken in this project and FP_Growth may work better with bigger sample size.
- The FP Growth algorithm quickly ran out of memory, even for small data sizes. A standard machine usually has 8–16 GB of RAM. Because FP Growth has to store the entire FP tree in memory and the memory footprint of the tree can be very large, even for smaller datasets, this was expected.

Scalability is an important consideration when it comes to coding decisions, as it considers the input data size as if it were large enough to be deemed "big data." When the conditions and nature of the data structure to be managed are suitable for distributed and parallel computing, map and reduce functions are used.

Experiments are carried out to assess the algorithm's performance as a whole, as well as to see how well it performs when used to compute confidence metrics for market-basket analysis. Apriori and the FP Growth algorithm both behave as expected in certain circumstances. In summary, a highly scalable and efficient implementation of the Apriori algorithm to identify frequent itemsets and subsequently mine association rules has been presented in this project.

For future work on this, better text pre-processing techniques can be used to further remove irrelevant information from the dataset. For example, a POS tagger can be used to only keep noun phrases or adjective noun combinations. Furthermore, a more sophisticated and high-spec machine can be used to perform the analysis at a larger scale to get better insights.