An Apriori Algorithm implementation for Market Basket Analysis

Sofia Introzzi, 967164

December 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 Data

The project aims to find frequent itemsets inside a collection of baskets. This study is also known as Market Basket Analysis (MBA).

The dataset chosen for this purpose is the "Ukraine Conflict Twitter Dataset" published on the open-source website Kaggle, and it has been retrieved via Kaggle API.

Considering the size of the data, for a time consuming purpose, only a sample has been considered for the application: the file concerning tweets of the date of 28Th of September of 2022.
The reason behind this choice has been related to the break out of news of the rupture of the Nord Stream gas pipeline of two days earlier.

The language in which tweets have been retrieved is German, considering the country an interested part of the event.
Data consists of several information related to tweets publication, user's profile, comment, language. Although, our interest is focused on the content of the tweets, thus the column 'text' has been extracted. For the Market Basket

Analysis, tweets represents our baskets and words our items.

Although in this setting only a sample of data is examined, the processing has been implemented to be suitable for any size of data, i.e. large-scale data.

As first step then, we have created the framework that enables us to perform parallel processing: the SparkContex.

Our data has been converted into a Resilient Distributed Dataset (RDD) through the method `.parallelize(data)`.
Data, if not specified, is partitioned automatically; in this case into two chunks.

```
[ ]  from pyspark.sql import SparkSession
     spark = SparkSession.builder.appName("test").master("local[*]").getOrCreate()
     sc = spark.sparkContext
     spark
```

```
[ ]  rdd = sc.parallelize(data)
     rdd.getNumPartitions()
```

Figure 1: PySpark

As previously stated, the task is finding frequent itemsets. Our interest is then finding the occurrences of n - items set in our file that satisfy a certain threshold.

The processing is made possible by the MapReduce implementation.

2 Pre-processing

In order to conduct a reasonable analysis, some pre-processing needed to be implemented. Indeed, as we are concerning text-type data, the collection of tweets had to be subject to some cleaning.

The NLP Natural Language Process (NLP) has been for this reason performed by taking into account that German had been selected as target language. The library we relied on has been the NLTK.

First, words have been tokenized by specifying German as language. Then, stop words have been removed, to avoid spoiling our search. Finally, punctuation has been removed and words have been flatted to lower case form.

```
[ ]  def pre_process(tweets):
       text = nltk.word_tokenize(tweets, language = 'german')
       text = [w for w in text if not w in stopwords]
       text = [lemmatizer.lemmatize(word) for word in text]
       text = " ".join([w for w in text if w not in puncts])
       text = text.lower()
       text = text.split()
       return text
```

Figure 2: Data cleaning

Those steps have been merged in a function that has been applied through a map function to the rdd.

3 Apriori algorithm implementation

As previously stated, the MapReduce processing is the core computation that allows us to process the distributed dataset.

Before implementing the algorithm, each basket need to store only unique values. Thus, the `ReduceByKey` is applied.
The function maps strings to one, afterwards, values with same key are merged - reduced -.

```
[ ]  plain_rdd = (preprocessed_rdd.flatMap(lambda tw: tw)
                                  .map(lambda w: (w, 1))
                                  .reduceByKey(lambda tw1, tw2: 1)
                                  .map(lambda x: x[0])
                                  .collect())
```

Figure 3: Unique words

As we want to run our computation with integers rather than strings, we created a dictionary that takes as value the unique words present in the distributed file and return an associated integer as value.

This passage enabled writing the function `convert(testo)` that, by scanning each word in basket, and comparing them to whose in the dictionary, it returns the value associated to the key and thus converts the collection of

```
[ ]  dizionario = dict(zip(plain_rdd, range(len(plain_rdd))))
```

Figure 4: Dictionary

strings into a collection of integers.

```
[ ]  def convert(testo):
       listaint = [None] * len(testo)

       for index in range(len(testo)):
         word = testo[index]
         for k, v in dizionario.items():
           if k == word:
             listaint[index] = v

       return listaint
```

Figure 5: Conversion function

We passed thus the function to the file that will now store a new distributed file containing integers such that for each occurrence of a certain word its corresponding integer value was printed. The new file is the `rdd_int`.

Afterward, we counted the occurrences of the singletons by using the file of integers as index.

We have thus created two tables: one with the integers that identify the items, and one for counting their frequency.

```
[ ]  count_singletons = (rdd_int.map(lambda index: (index, 1))
                                .reduceByKey(lambda x1, x2: x1+x2))
```

Figure 6: Singletons count

A key point in the Market Basket Analysis is given by the choice of the support threshold s. This latter indeed identifies whose items to be considered frequent or not. Therefore, before we moved to the second pass, we defined s. Here a support of 1% has been chosen.

Therefore, words in basket those occurrences will be higher than the threshold will be considered.

Here we filtered the singletons by keeping only whose that satisfied the support threshold.

Then, we defined a function `count_freq` such that, by passing it to our RDD, through a map-reduce process, it returns the counts of occurrences of whose elements (unique, pairs, or triples..) that satisfies the support threshold and thus are frequent.

```
[ ]  def count_freq(rdd):
       return (rdd.flatMap(lambda w: w)
                  .map(lambda w: (w, 1))
                  .reduceByKey(lambda x1, x2: x1+x2)
                  .filter(lambda x: x[1] >= s))
```

Figure 7: Counts of frequent itemsets

Thus, we could identify the frequent singletons.

```
[ ]  frequent_singletons = count_freq(preprocessed_rdd).map(lambda x: x[0]).collect()
```

Figure 8: Frequent single itemsets

Now, a relevant assumption for following reasoning is the monotonicity property: it states that if a set of items I is frequent, so it is every subset of it. This also induces that no itemset can be frequent unless its subsets are not frequent as too.

For this reason, if we want to find frequent pairs, each singleton must be frequent. This latter would be indeed candidates for being items of the frequent pairs.

In order to create pairs, we first scanned the baskets in the distributed file and we kept only whose elements present in `frequent_singletons` that we are interested in.

```
[ ]  def scan(basket):
       basket = [w for w in basket if w in frequent_singletons]
       return basket
```

Figure 9: Only frequent items

Finally, we were able to detect the frequent pairs: here the triangular method was used to return pairs of itemsets and their occurrences.

We have generated all the possible pairs for each item in the baskets. Afterwards - as we did for the singletons - we count the occurrences of the pairs in the dataset and we retain only those that are frequent.

```
[ ]  count_pairs = count_freq(rdd_unique_in_basket.map(generate_set2)).sortBy(lambda x: -x[1])
```

Figure 10: Counts of frequent pairs

The same processing is later performed to find frequent triples.

```
[ ]  def generate_set3(basket, r=3):
       basket = [comb for comb in combinations(basket, r)]
       return basket
```

```
[ ]  count_triples = count_freq(rdd_unique_in_basket.map(generate_set3)).sortBy(lambda x: -x[1])
```

Figure 11: Find frequent triples

4 Association rules

Finally, after having obtain the frequency of the sets, we were able to compute two interesting measures that define the association rules: confidence and interest.
First, we consider how likely is a word to compare in tweet given a subset of words. This is known as confidence. It is calculated as the ratio between the support of that subset with the target word and the support of the support of that subset without the target word.
Although, for a more accurate meaning, we can compute the interest: the

difference between the confidence of a word and the support of that word. This measure suggests us how much actually is possible to find a word in tweet given a subset.

```
[ ]  def make_confidence_interest(target):
       if target in triple_supp.keys():
         supp_IuJ = triple_supp[target]
         if target[:-1] in double_supp.keys():
           supp_I = double_supp[target[:-1]]
           confidence = supp_IuJ/supp_I
           if target[2] in single_supp.keys():
             supp_j = single_supp[target[2]]
           interest = confidence - supp_j/len(data)
           return confidence, interest
```

Figure 12: Confidence and Interest

5 Considerations and conclusions

In conclusion, although only a portion of dataset has been selected for this project, the implementation has been performed considering a large-scale data.

Processing has been implemented inside the Spark environment executing Map and Reduce functions. Data has been stored as Resilient Distributed Dataset that enables parallel computation.

Concerning the results, outcomes have been in line with expectations. We recall that a support threshold of 1% was selected. It is not surprising indeed that only small subsets of itemsets has been found to be frequent. Only 32 itemsets have been found to be frequent pairs. While, when passing to the search of frequent triples, results dropped to only three solutions.

Finally, for a Market Basket Analysis model, association rules represent meaningful measures to perform too. In one case we obtained a confidence of 0.33 with an interest of 0.17, while in another a confidence of 0.45 and interest of -0.007.