Data Science and Economics: Algorithms for Massive Datasets

Market-Basket analysis with Big Data

An application of the A-priori algorithm with Spark to find frequent combinations of items

Alessandro Mirone
966880

October 2022

UNIVERSITÀ DEGLI STUDI DI MILANO

Abstract

The following article presents an application of the A-priori algorithm to Market-Basket analysis in a distributed setting through PySpark package in Python. The objective of the analysis is to build association rules of the form $I \rightarrow j$ where I is a set of frequent items and j is a singular item; in this particluar application, items are words from tweets regarding the Ukrainian-Russian conflict. Rather than commenting the results, the focus of the article is to show how the A-priori algorithm can be implemented in a distributed setting, that is how to write the algorithm to allow for a scalable solution when applying this analysis to Big Data.

Dataset

The dataset, obtained from https://www.kaggle.com/datasets/bwandowando/ukraine-russian-crisis-twitter-dataset-1-2-m-rows (https://www.kaggle.com/datasets/bwandowando/ukraine-russian-crisis-twitter-dataset-1-2-m-rows) contains 50+ millions tweets in various languages, spanning a period of (currently) seven months since the beginning of the conflict. To show how to write the algorithm within a Spark context, is not necessary to use an actual distributed setting: Such situation can be mimicked by executing the Spark session locally; this allows to run the algorithm through the Spark shell, but doesn't use a real distributed storage system nor a distributed computing framework, so that nothing changes between this example and a real case usage of this technique, apart from the absence of such distributed framework. This means, however, that the analysis will use only a subsample of the whole dataset, to allow for storage and computing: in particular, the dataset used for this

article consists in 500 tweets in english language, collected during the fifth of september, 2022. In a distributed environment (Eg. Hadhoop) data have to be loaded into the distributed storage system. In this example, data are downloaded from Kaggle through the Kaggle API and stored in disk memory at runtime in zip format.

In [ ]:

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

In [ ]:

```
#downlaod the data
os.environ["KAGGLE_USERNAME"] = "" #parameters for the kaggle API
os.environ["KAGGLE_KEY"] = ""

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

After the data have been loaded, they are exported to the environment with `pd.read.csv` . To prepare them for computation using Spark, A Resilient Distributed Dataset (RDD) is created with the method `parallelize` . In this step, only english tweets are retained. RDDs are the Spark version of dataframes: this objects are automatically distributed in partitions for parallel computation over the cluster in an efficient and resilient manner; operation carried out on rdds propagate over the cluster so that using a RDD (and functions suitable to operate on them) is the only necessary condition to parallelize the computation in Spark. Functions performed over RDD are of two kinds: actions and transformations; because Spark uses lazy evaluation, transformations are carried out only after an action is invoked: this allows to streamline the operations on the RDD and save resources by optimizing the processing.

In [ ]:

```
#Load zipped file in environment
from zipfile import ZipFile
with ZipFile('0905_UkraineCombinedTweetsDeduped.csv.gzip.zip', 'r') as zip:
    zip.extractall()
filename = r'0905_UkraineCombinedTweetsDeduped.csv.gzip'
df = pd.read_csv(filename, compression='gzip', index_col=0,encoding='utf-8', quoting=csv.QU
```

In [ ]:

```
rdd = sc.parallelize(df.text[df.language == 'en'].sample(n=500,random_state=33))
```

Preprocessing

The RDD has the form of a list distributed between the nodes; each position in the list is currently occupied by a string (the text of each tweet from the dataset). Before proceding, tweets are tidied with the `cleantxt` function.

In [16]:

```
def cleantxt(x):
    res = x.lower() #Lowercase

    res = re.sub(r'<.*?>','',res) #remove html tags

    res = re.sub(r'https?://\S+|www\.\S+', '', res) #remove URLs

    res = re.sub(r'\d+', '', res) #remove numbers

    res = unidecode.unidecode(res) #convert accented characters to ascii characters

    emoji_char = re.compile("["
                            u"\U0001F600-\U0001F64F"  # emojis
                            u"\U0001F300-\U0001F5FF"  # symbols & pictographs
                            u"\U0001F680-\U0001F6FF"  # transport & map symbols
                            u"\U0001F1E0-\U0001F1FF"  # flags (iOS)
                            u"\U00002500-\U00002BEF"  # chinese char
                            u"\U00002702-\U000027B0"
                            u"\U00002702-\U000027B0"
                            u"\U000024C2-\U0001F251"
                            u"\U0001f926-\U0001f937"
                            u"\U00010000-\U0010ffff"
                            u"\u2640-\u2642"
                            u"\u2600-\u2B55"
                            u"\u200d"
                            u"\u23cf"
                            u"\u23e9"
                            u"\u231a"
                            u"\ufe0f"  # dingbats
                            u"\u3030"
                            "]+", flags=re.UNICODE)
    res = emoji_char.sub(r'',res) #remove emojis, pictographs and other unusual characters

    res = re.sub("&amp", "and", res).replace("\n", " ") #remove refuses due to encoding

    res = res.translate(str.maketrans('', '', punctuation)) #remove punctuation

    pattern = re.compile(r'\b(' + r'|'.join(stopwords.words('english')) + r')\b\s*') #remov
    res = pattern.sub('', res)

    res = re.sub('\s+'," ", res) #remove whitespaces

    spell_corrector = Speller(lang='en') #corrector NOTE: this is an english corrector: as
    res = spell_corrector(res)


    return res
```

In [ ]:

```
rdd = rdd.map(cleantxt)
```

This RDD is then transformed to obtain a list of lists split between the nodes, with positions in the array occupied by lists containing the texts of each tweet (*i.e.* baskets): a map function will recursively take as inputs each string in the initial array and a lambda function, stacking each text content in the positions of the list. Each list t contains n_t elements (all the words in that tweet) and represents a basket in the analysis.

In [ ]:

```
rdd = rdd.map(lambda x: x.split(' ')) #"tokenization" (basket creation)
```

The `lem` function is then applied to the RDD in the same way: the result is again an array of lists (baskets) each containing the tokens present in the relative basket as single string elements. The A-priori algorithm can be directly applied to this array.

In [19]:

```
def lem(res): #Lemmatization
    from nltk.corpus import wordnet
    for i in range(len(res)):
        res[i] = (nltk.pos_tag(res))[i]
        wntag = get_wordnet_pos(res[i][1])
        if wntag is None:  # don't supply tag in case of None
            res[i] = lemma.lemmatize(res[i][0])
        else:
            res[i] = lemma.lemmatize(res[i][0], pos = wntag)

    res = list(filter(None,res)) #remove NA elements still in the baskets
    return res
```

In [ ]:

```
rdd = rdd.map(lem) #apply lemmatization
```

Algorithm

The A-priori algorithm is composed of k subsequent steps:

- In the first step, $g = 1$, a full scan of the baskets file is performed: all items are assigned an integer and the occurences of each item are counted and stored in an array
- The second step, $g = 2$, considers only items (combinations in the successive steps) which occurences are greater than a given threshold s (frequent items); each frequent item/combination is recursively paired with all other frequent items in the baskets file. A second full scan is then performed over the baskets file: each newly formed combination is assigned an integer and the occurences of each combination are again counted and stored.
- For $g \leq k$, repeat the second step and increase g by one for each iteration.

The algorithm either terminates when reaching k steps, or when it can no longer finds combinations which occurences are above the threshold s. Note how g marks the cardinality of the combination in the g-th step: The first step identifies frequent singletons, the second one identifies frequent pairs, the third one identifies frequent triplets, and so on.

The following section comments the code of the reproducible example attached to this report. To view the full code, including loading and preprocessing, just open the mark2.ipynb file.

In [21]:

```
# a priori implementation; phase 1
# count frequency of each item (word) #WARNING: this function is extremly slow in a local s

def count_freq(rdd):
    return (rdd.flatMap(lambda x: x)
            .map(lambda word: (word, 1)).
            reduceByKey(lambda a,b: a+b))
```

In [22]:

```
freq = count_freq(rdd) #all words frequencies
freq.take(10)
```

Out[22]:

```
[('ukrainian', 59),
 ('volodymyr', 2),
 ('zelenskyy', 6),
 ('praise', 2),
 ('work', 19),
 ('down', 2),
 ('missile', 7),
 ('helicopter', 2),
 ('vow', 1),
 ('defend', 8)]
```

In the first step, the full scan over the basket file is performed; the `flatMap` function is used to merge the baskets into a single list, the inverse of the operation carried out when tokenizing. This is necessary because the subsequent `map` function assigns to each word in the basket the integer 1, forming key-value pairs of the form (word,1). The final `reduceByKey` sums all values (the integers) grouping them by key, so that the result of the first phase is a list of key-value pairs, each composed by a unique word and relative counter.

In [24]:

```
def threshold(x):
    return True if x[1] >= 25 else False #we define the threshold at 5% (25 for 500 tweets)
tab = freq.filter(threshold)
tab.take(10) #most frequent words and corresponding frequences
```

Out[24]:

```
[('ukrainian', 59),
 ('russiaisaterroriststate', 25),
 ('person', 35),
 ('nato', 29),
 ('say', 47),
 ('putin', 66),
 ('ukrainerussiawar', 43),
 ('europe', 28),
 ('standwithukraine', 30),
 ('russian', 139)]
```

The second step is initialized by retaining only the words occurying more times than the threshold s throughout the basket file.

In [ ]:

```
unique_words = tab.map(lambda x : x[0])
```

During this phase a different RDD containing all the frequent words in the corpus is created for later use.

In [28]:

```
b=2
c=2
d=1
tab2 = tab
tab2.cache()
while(tab2.isEmpty() == False):

    tab_words = tab2.map(lambda x : x[0]) #collect combinations of frequent words of recurr
    while b >= 3:
        tab_words = tab_words.map(lambda x : [x[i] for i in range(len(x))]) #convert previo
        break
    tab_words.cache()
    combined = unique_words.cartesian(tab_words) #create all possible combinations with wor
    while b >= 3:
        combined = combined.map(unpack) #add the next word to the combination
        break
    combined.cache()
    combined = combined.map(removeReplica) #remove duplicates and permutations
    combined = combined.distinct()
    combined.cache()
    combined = combined.filter(lambda x: len(x) == c)
    combined.cache()
    combined_2 = rdd.cartesian(combined) # build the confront table: here each of the combi
    combined_2.cache()
    combined_2 = combined_2.map(count_comb) #retain combinations if they appear in at least
    combined_2.cache()
    combined_2 = combined_2.filter(lambda x : x is not None) #clean output of previous step
    combined_2.cache()
    combined_2 = combined_2.map(lambda x: (x , 1)) #assign the counters to the combinations
    combined_2.cache()
    combined_2 = combined_2.reduceByKey(lambda a,b: a+b) #count the occurences of the combi
    combined_2.cache()
    combined_2 = combined_2.filter(lambda x: x[1] >= 25) #retain only frequent combinations
    combined_2.cache()

    while d == 1:
        result = combined_2
        d +=1
    result = result.union(combined_2).distinct() #create the output table
    result.cache()
    tab2 = combined_2 #initialize the next table
    tab2.cache()
    c += 1
    b += 1
```

This block contains the core of the algorithm: the `while` loop encloses the iterative procedure to construct the results. The input of the first `map` function is a RDD containing the list of frequent words identified in the first step; each key (the single word) from that list is isolated and stored in the `tab_words` RDD. In the second step, the inner `while` loops are inactive, so that the next operation is the `cartesian` transformation: this

function computes the cartesian product of `unique_words`, the list of all frequent words in the basket file, with `tab_words` itself. the result is the RDD `combined` which stores a list of tuples of the form (unique_word, tab_word), computed for all words in `unique_words` and in `tab_words`; this is the input of the second phase of the second step. In the second phase the tuples are again combined, but this time with the baskets: `combined_2 = rdd.cartesian(combined)` is another RDD, composed of lists of baskets and tuples formed in the previous step: is the heaviest object in the algorithm, containing $N \times J \times F$ occurences, where N is the number of baskets, J is the number of words in `unique_words` and F is the number of words in `tab_words` (so that, in the first loop of the algorithm, $J = F$). This RDD is the input to the `count_comb` transformation:

In [ ]:

```
def count_comb(x): #define function to identify combinations'presence in the baskets
    y=list("a"*len(x[1]))
    for i in range(len(x[1])):
        if x[1][i] not in x[0]:
            continue
        else:
            y[i]=x[1][i]

    if "a" not in y:
        return tuple(y)
    else:
        return None
```

this function retains a combination of words (in the second step, combinations are pairs) if the combination is present in at least one basket, and returns it each time it appears. It's used in pair with the `filter` transformation below, as the function produces some NA in the RDD.

In [ ]:

```
combined_2 = combined_2.filter(lambda x : x is not None)
```

The next `map` and `reduceByKey` functions perform what was already seen in the first step: they respectively assign an integer to the combinations appearing in the baskets and then count the occurences of those combinations. The next `filter` transformation retains only the combinations appearing at least s times.

The final part of the `while` loop stores the results and intialize the next step; the cycle terminates when no more frequent combinations are found.

Note that, since the third step, the two innermost `while` conditions become active: the earliest one is used to allow the first `cartesian` function to create combinations of words in the form of *lists* of two elements, wich are the previously formed combination created in the previous step and a word from `unique_words`; one such list is created for each combination and for each word in `unique_words`. To reconvert the combination in the form shown before, that is, a tuple which elements are the words in the combinations, the function `unpack`, shown below, is also applied through a `map` transformation.

In [ ]:

```
def unpack(tup): #define function to unpack previously formed combinations for processing
    t=tuple()
    for x in tup:
        if not type(x) == list:
            t += (x,)
        else:
            t +=(*x,)
    return t
```

Finally, the function `removeReplica`, in pair with the `distinct` transformation, is used to strike permutation of the same combination from the list of possible combinations.

In [ ]:

```
def removeReplica(record): #define function to remove permutations of a combination
    lt = []
    for i in record:
        if i not in lt:
            lt.append(i)
    lt.sort() #sort tuples for later distinct
    result = tuple(lt)
    return result
```

To collect the results, a final `for` loop is performed: for each combination and for each word in that combination, an association rule of the form $I \rightarrow j$ is output and confidence and interest are calculated; for details on association rules, see the "Results" section.

In [29]:

```
result_list = result.collect()
result_list
```

Out[29]:

```
[(('ukraine', 'ukrainian'), 31),
 (('russia', 'russian'), 40),
 (('putin', 'russia'), 29),
 (('russia', 'ukraine'), 59),
 (('ukraine', 'war'), 32),
 (('russian', 'ukraine'), 52)]
```

In [30]:

```
interest=[]
for tupl in result_list:
    support_I_j = tupl[1] #get support of I U j
    for y in tupl[0]:
        j = y
        support_j = tab.filter(lambda x: (x[0] == "{}".format(j))).collect()[0][1] #get sup
        full = [word for word in tupl[0]]
        full.remove("{}".format(y)) #get support of I
        I = full
        if len(I) == 1:
            I = I[0]
            support_I = tab.filter(lambda x: (x[0] == "{}".format(I))).collect()[0][1]
        else:
            I = tuple(I)
            support_I = result.filter(lambda x: x[0] == I).collect()[0][1]

        confidence = (support_I_j / support_I)
        number = confidence - (support_j / len(rdd.collect()))
        temp = [I,j,confidence,number]
        interest.append(tuple(temp))
```

Scalability

As just shown, the algorithm is entirely written as a sequence of `map` and `reduce` functions over RDDs; this means that the analysis can be used for massive collections of data without any change to the algorithm. The only `collect` action invoked is used for computing confidence and interest of the combinations found by the algorithm, so it's not part of it but it's rather used to retrieve the analytics. However, even when considering it as a part of the method, the result of the A-priori algorithm (the frequent combinations) is a very small subset of the total data set, and the RDD `result` can be safely collected in all practical application without any problem, as long as the threshold scales with the number of records.

Results

The aim of the market-basket analysis is to find ***association rules***: an association rule is a function of the form $I \rightarrow j$, where I is a set of frequent items and j is an item whose presence or abscence in the basket can be inferred by the strenght of the association rule, expressed through *confidence* and *interest* of the rule. In particular:

- to build an association rule, select combinations of items, I, which are *frequent* in the basket file. A combination I is frequent if it occurs in more than s baskets. Then, for each $j \in I$, build an association rule of the form: $I \setminus \{j\} \rightarrow j$
- the ***confidence*** of the association rule is defined as:

$$\frac{Support(I \cup \{j\})}{Support(I)}$$

- the ***interest*** of the association rule is defined as:

$$Confidence - \frac{Support(\{j\})}{N}$$

where N is the number of baskets.

Ideally, a good association rule has high confidence and high interest.

The results of this analysis, summarised by the output below show, is unsurprisingly not very interesting; with only 500 records, the only two rules with some interest are "putin" → "russia" , with a confidence of 0.43, and "war" → "ukraine", with a confidence of 0.52. However, both rules have low interest, respectively 0.08 and 0.09, meaning that the j , in this two cases "russia" and "ukraine" is far from being univocally associated with the item in the set of one element I. Moreover, only item sets of cardinality one were retained, characterising again the scarcity of meaningful results with so little data.

In [31]:

```
interest #first term (or first tuple) is the I, second term is the j, first number is the c
```

Out[31]:

```
[('ukrainian', 'ukraine', 0.5254237288135594, 0.09742372881355937),
 ('ukraine', 'ukrainian', 0.14485981308411214, 0.026859813084112144),
 ('russian', 'russia', 0.28776978417266186, -0.06823021582733813),
 ('russia', 'russian', 0.2247191011235955, -0.053280898876404525),
 ('russia', 'putin', 0.16292134831460675, 0.030921348314606745),
 ('putin', 'russia', 0.4393939393939394, 0.08339393939393941),
 ('ukraine', 'russia', 0.2757009345794392, -0.08029906542056076),
 ('russia', 'ukraine', 0.33146067415730335, -0.09653932584269664),
 ('war', 'ukraine', 0.5245901639344263, 0.09659016393442627),
 ('ukraine', 'war', 0.14953271028037382, 0.027532710280373823),
 ('ukraine', 'russian', 0.24299065420560748, -0.03500934579439255),
 ('russian', 'ukraine', 0.37410071942446044, -0.053899280575539554)]
```

Conclusions

In this particular example, the lack of solid results is due to the low cardinality of the data; this was an unavoidable measure, since the algorithm is designed to work on a distributed system: when executed locally, PySpark `reduceByKey` functions render the computation extremely slow, and thus the problem was simplified at the expense of the results of the analysis. Nonetheless, the solution proposed for applying the A-priori algorithm to big data is scalable and effectively able to return insights for a market-basket type of analysis.