Market Basket Analysis
Apriori algorithm implementation

Project for Algorithm for Massive Data course
Data Science and Economics

Marco Cazzola (matr: 964573)

UNIVERSITÀ DEGLI STUDI DI MILANO

FACOLTÀ DI SCIENZE POLITICHE,
ECONOMICHE E SOCIALI

Abstract

In this paper, we are going to implement the apriori algorithm, an algorithm which is particularly popular in the market basket analysis setting, since it allows to detect frequent itemsets over a set of baskets. In our setting, the "baskets" will be tweets (properly preprocessed) about the Ukrainian war (available on Kaggle), and the items will be words. What we are interested in is to find words appearing often together, from which we will derive association rules whose strength will be tested by considering their confidence and interest. The peculiarity of this work resides in the fact that the code has been written in order to be scalable, i.e., in order to be applied efficiently to Big Data datasets.

Contents

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

The dataset we will use to test the apriori algorithm we are going to implement is the Ukraine Conflict Twitter Dataset, available on Kaggle and consisting in a collection of daily tweets about the ongoing conflict between Russia and Ukraine. In this paper, we are going to focus on a sample of 5.000 English tweets from the 4^{th} of April, that is the day when incontrovertible photographic and video evidence about the atrocities committed by the Russian soldiers in Bucha started to be spread across the Internet.

2 Data import and preprocessing

After importing the data through the Kaggle API, we focus on the tweets marked as "English" (`en`) and we extract a random sample of 5.000 unique documents out of the corresponding column.

```
df = pd.read_csv("/content/0404_UkraineCombinedTweetsDeduped.csv.gzip",
                 compression='gzip', index_col=0, encoding='utf-8',
                 quoting=csv.QUOTE_ALL)

df = df[df.language == 'en']

tweets = pd.Series(df.text.unique())\
.sample(n = 5000, random_state=42)
```

We then need to define a function to transform the obtained tweets in something that can be given as input to our algorithm. This is precisely the goal of the `cleaning_txt` function that, given a tweet (string of multiple words) as input, it:

1. Lowercases every word in the string;
2. Removes URLs, since they do not carry any particular meaning for the market basket analysis we need to carry out;
3. Removes punctuation (# and @ included);
4. Removes emoticons and deals with potential UTF-8 codes erroneously displayed;

5. Tokenizes the tweet, in the sense that it extracts a list of words out of each tweet;
6. Removes stopwords from the list (e.g., conjunctions and articles);
7. Lemmatizes the words in the list, so to reduce the variability of the words and also decrease the number of unique words in the corpus;
8. Returns just the unique words inside the list representing the original tweet.

```
def cleaning_txt(tweet):

  #1. Lowercase everything
  res = tweet.lower()
  #2. Remove URLs
  res = re.sub("https?:\/\/.*[\r\n]*", "", res)\
  #3. Remove punctuation
  res = "".join([ch for ch in res if ch not in punct])
  #4. After point 3, "&amp" (standing for "&") has become "amp",
  #so let us substitute any " amp " with " and "
  #   Moreover, let us also remove any "strange" utf code and emoticons
  res = re.sub(" amp ", " and ", res).encode("ascii", "ignore")\
  .decode().replace("\n", " ")
  #5. Tokenize
  res = res.split()
  #6. Remove stopwords
  res = [w for w in res if w not in stop_words]
  #7. Lemmatization
  res = [lemmatizer.lemmatize(w) for w in res]
  #8. Just unique words inside a sentence
  res = list(set(res))

  return(res)
```

We then create a Spark context in order to distribute our sample of 5.000 English tweets through the `parallelize` Spark method, so to be able to apply the `cleaning_txt` function we have just described above in a distributed manner.

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

rdd = sc.parallelize(tweets)

rdd = rdd.map(cleaning_txt)
```

Lastly, we create a series of object that will be useful later during the algorithm implementation:

1. `n`: the number of tweets in the corpus;
2. `unique_ws`: a list of the unique words contained in the corpus;
3. `unique_wsk`: a dictionary building a one-to-one correspondence between the unique words in the corpus and integer numbers. In particular, this dictionary will have words as keys and integers as values;
4. `unique_wsv`: a dictionary building a one-to-one correspondence between the unique words in the corpus and integer numbers. In particular, this dictionary will have integers as keys and words as values.

```
#Nr of baskets
n = len(tweets)

#List of unique words
unique_ws = rdd.flatMap(lambda txt: txt)\
.map(lambda w: (w, 1))\
.reduceByKey(lambda w1, w2: 1)\
.map(lambda x: x[0])\
.collect()

#Dict of the type(word : int)
unique_wsk = dict((w, i) for i,w in enumerate(unique_ws, 1))
#Dict of the type(int : word)
unique_wsv = dict((i, w) for i,w in enumerate(unique_ws, 1))
```

3 Algorithm implementation

Before the actual algorithm implementation, we need to define a couple of "auxiliary" functions, that are `toWords` and `candidateFrequentSets`:

```
def toWords(x, w_dict):
  if type(x) == int:
    return( w_dict[x] )
  else:
    return( (tuple([w_dict[i] for i in x])) )

def candidateFrequentSets(sent, freq_only):

  if type(freq_only[0]) == int:
    cand_set = [tuple(sorted(t)) for t in list(combinations(sent, 2))]
    res = []
    for cand in cand_set:
      filtr = [w for w in cand]
      if all( [sub in freq_only for sub in filtr] ):
        res.append((cand, 1))
      else:
        continue

  else:
    km1 = len(freq_only[0])
    cand_set = [tuple(sorted(t)) for t in list(combinations(sent, km1 + 1))]
    res = []
    for cand in cand_set:
      filtr = [tuple(sorted(t)) for t in list(combinations(cand, km1))]
      if all( [sub in freq_only for sub in filtr] ):
        res.append((cand, 1))
      else:
        continue

  return(res)
```

`toWords` is a function that, given as input an integer number (or a list of integers), returns the corresponding word (or list of words).

`candidateFrequentSets` is the core of apriori algorithm: it takes as input a tweet (`sent`) and, given a list of frequent itemsets of size k (`freq_only`), it returns a list of key-value pairs of the type $(I, 1)$, where I is an itemset of size $k+1$ whose all possible subsets of size k appear in `freq_only`, and 1 is just a value to ease the task of counting the occurrences of I in the whole corpus. This allows us to significantly reduce the amount of candidates to be considered as potential frequent itemsets. Suppose for example we are interested in frequent pairs of words: rather than considering each possible pair of words in the corpus and count their occurrences, we can just consider the pairs composed by words that have already been found to be frequent. This set of pairs composed of frequent words only is precisely what is output by the `candidateFrequentSets` function we have just discussed. The `if-else` structure of the function is just to properly deal with the particular case in which we need to output candidate frequent pairs against the more general case where we output k-sized candidates, with $k > 2$.

We are now ready to present the full-algorithm implementation. The `apriori` function takes as input the RDD of tweets (properly preprocessed), the dictionary for words-to-integers translation, the number of baskets `n`, the size `k` of the frequent itemsets we are interested in and the threshold that allows us to define what *frequent* means (in this case, we consider to be frequent any item appearing in at least 2% of the baskets). As output, the function returns a list of k RDD, each containing the frequency of the frequent itemsets of different cardinality, from 1 to k. We do so because, in order to compute the confidence and interest of the association rules we are going to test, it is advisable to have at hand not only the frequencies of frequent itemsets of size k, but also the frequencies of singletons and of frequent itemsets of size $k-1$.

```
def apriori(rdd, i_dict, n, k=2, s=0.02):

  counts = [None] * k
  freq_only = [None] * k

  sent_int = rdd.map(lambda x: [i_dict[w] for w in x])
```

```
counts[0] = sent_int.flatMap(lambda sent: sent)\
.map(lambda w: (w, 1))\
.reduceByKey(lambda w1, w2: w1+w2)\
.filter(lambda x: x[1] > s*n)

freq_only[0] = counts[0]\
.map(lambda x: x[0])\
.collect()

if k == 1:
  return (counts)

else:
  curr_k = 2

  while curr_k <= k:
    counts[curr_k - 1] = sent_int.filter(lambda x: len(x) >= curr_k)\
    .flatMap(lambda x: candidateFrequentSets(x, freq_only[curr_k - 2]))\
    .reduceByKey(lambda t1, t2: t1+t2)\
    .filter(lambda x: x[1] > s*n)

    freq_only[curr_k - 1] = counts[curr_k - 1]\
    .map(lambda x: x[0])\
    .collect()

    if len(freq_only[curr_k - 1]) == 0:
      return("No frequent itemsets of such size.")
    elif curr_k == k:
      return (counts)
    else:
      curr_k += 1
```

The output of `apriori` function is a list of RDD containing key-value pairs of the type (I, f), where I is a frequent itemset of any size between 1 and k, and f is the frequency of I. However, it should be noted that items inside I are still encoded as integers; in order to access words and their frequency, we need to pass the output of `apriori` function to the `getWordsAndFreq`

function that, taking as input the list of frequencies, the size of the itemsets we are interested in and the dictionary for words translation, it is able to return us a dictionary with the word(s) as keys and their frequency as values.

```
def getWordsAndFreq(counts, w_dict, k):

  res = counts[k-1].map(lambda x: (toWords(x[0], w_dict), x[1]))\
  .sortBy(lambda x: -x[1])\
  .collect()

  return(dict(res))
```

Finally, let us consider the function that we use to compute confidence and interest. Starting from a set of frequent itemsets, we iteratively compute those two metrics for all the possible association rules. In other words, starting from a frequent itemset of size k, we consider all the possible subsets I of size $k-1$ and compute the metrics for the rule $I \setminus \{j\} \Rightarrow j$ for all possible I and j.

```
def confAndIntr(counts, n, w_dict, k):

  big_sets = counts[k-1].map(lambda x: x[0]).collect()
  conf = {}
  intr = {}

  for big_set in big_sets:
    if k == 2:
      Is = [i for i in big_set]
    else:
      Is = [tuple(sorted(t)) for t in list(combinations(big_set, k-1))]

    js = [int(np.setdiff1d(big_set, t)[0]) for t in Is]

    freq_big_set = counts[k-1].filter(lambda x: x[0] == big_set)\
    .map(lambda x: x[1])\
    .collect()[0]
```

```
    for idx in range(len(Is)):
      I = Is[idx]
      j = js[idx]

      freq_I = counts[k-2].filter(lambda x: x[0] == I)\
      .map(lambda x: x[1])\
      .collect()[0]

      freq_j = counts[0].filter(lambda x: x[0] == j)\
      .map(lambda x: x[1])\
      .collect()[0]

      conf[(toWords(I, w_dict), toWords(j, w_dict))] = freq_big_set / freq_I
      intr[(toWords(I, w_dict), toWords(j, w_dict))] = (freq_big_set / freq_I)-\
                                                       (freq_j / n)

  conf = dict(sorted(conf.items(), key= lambda x: -x[1]))
  intr = dict(sorted(intr.items(), key= lambda x: -x[1]))

  return(conf, intr)
```

4 Scalability

It should be noted how the proposed solution makes an extensive use of Spark methods to ensure high scalability: starting from the preprocessing phase, everything is applied in a distributed manner, with the only exception of the list of frequent itemsets inside the `apriori` function, upon which a `collect()` action is called. However, it should be noted that this is simply a list of integers, thus needing very little space to be stored in main memory. Moreover, as we increase the size of frequent itemsets, their cardinality gets smaller and smaller, thus making this issue even more negligible.

5 Results

Now that we have defined all the needed functions, we can run them on the RDD we have created in Section 2 and see what are the results.

```
#Generate the counts
counts = apriori(rdd, unique_wsk, n, k=3)
freq_itemsets_k = getWordsAndFreq(counts, unique_wsv, k=3)
```

The frequent triplets (since we set `k=3`) found by the algorithm are the following:

```
{('russia', 'ukraine', 'russian'): 197,
 ('putin', 'russia', 'ukraine'): 197,
 ('russia', 'ukraine', 'war'): 190,
 ('putin', 'ukraine', 'war'): 148,
 ('ukraine', 'russian', 'war'): 128,
 ('ukraine', 'crime', 'war'): 120,
 ('bucha', 'russia', 'ukraine'): 120,
 ('bucha', 'ukraine', 'russian'): 103,
 ('putin', 'ukraine', 'russian'): 103,
 ('ukrainian', 'ukraine', 'russian'): 101}
```

From what can be seen at first glance, lots of frequent triplets contain the word `'ukraine'` and or `'russia'`. Also `'putin'`, `'war'` and `'bucha'` are quite frequent. Nonetheless, what we are interested in are association rules, so we run the related function over these results, to see which words have the strongest correspondence, as measured by confidence and interest.

```
conf, intr = confAndIntr(counts, n, unique_wsv, k=3)
```

The strongest rule both in terms of confidence and interest is {'ukraine', 'crime'} $\Rightarrow$ 'war', showing a confidence equal to 0.82 and an interest of 0.67. On the contrary, the rule {'russian', 'war'} $\Rightarrow$ 'ukraine' shows a confidence of 0.68 with an interest of 0.18, meaning that, while the rule is often true (confidence is high), the association between the specific words is not so high, probably because 'ukraine' is a very frequent word throughout the whole corpus.

6 Conclusion

In this paper, we implemented the apriori algorithm in order for the code to be run on massive datasets. Indeed, the original data structure has been changed to a Resilient and Distributed Dataset (RDD) in order to distribute much of the computations and therefore achieve scalability. The output of the apriori algorithm (a list of frequent itemsets of different size with their frequency) has been used to test several association rules, among which we have find lots of rules with high confidence, but very few with high interest.