UNIVERSITÀ DEGLI STUDI DI MILANO
Facoltà di Scienze Politiche, Economiche e Sociali
Master's degree in Data Science and Economics

Finding similar items - Ukraine Conflict

Gabriele Fusar Bassini - ID 974407

A. Y. 2021-2022

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.

Abstract

This paper studies how to apply an algorithm in order to find similar items in a large dataset. The case study is a dataset of Tweets concerning the Ukraine Conflict, while similar items are Tweets that, apart from users and embedded links, are actually identical. The algorithm exploits a Spark RDD (Resilient Distribuited Dataset) in order to be scalable for massive datasets. The analysis focuses on a sample of 10k English speaking Tweets from the Kaggle dataset mentioned before.

Contents

1 Data

The dataset used to perform this analysis is called "Ukraine Conflict Twitter Dataset". At the time of writing, it is available on Kaggle as CC0 (free to use) at this url: https://www.kaggle.com/datasets/bwandowando/ukraine-russian-crisis-twitter-dataset-1-2-m-rows . It contains over 400.000 Tweets in 61 different languages concerning the conflict. It reports the text of the Tweet, which is the interesting part for this paper, as well as a series of other informations such as the author, hashtags and location. For the porpuse of this project it is useful to select only one language. Moreover, for simplicity we consider only a sample of 10000 items to perform the algorithm. The dataset can be easily downloaded via Kaggle API and after unzipping it you are ready to transform it into a Pandas dataframe.

2 Preprocessing

Once we have selected the textual part into a Pandas series, we need to get rid of every useless information or possibile slight difference that could trick us into not recognizing plagiarism between Tweets. We use the function **cleaner** to remove references to other users, hashtags, links and emojis. We eventually lowercase the entire corpus of texts. This is also going to be useful in order to have less k-grams to deal with later on.

```
def cleaner(text):
  #remove @nickname, hashtags, links, emojis
  text = re.sub("(@[A-Za-z0-9]+)","",text)
  text = re.sub("#","",text)
  text = re.sub(r'http\S+', '', text)
  text = re.sub('\w+:\/\/\S+', '', text)
  text = remove_emojis(text)
  return text
```

The function **remove_emojis** expolits re.UNICODE and removes a variety of symbols from a given text. It was made available by the Japanese GitHub user **1n9-jp** on his repository: https://gist.github.com/n1n9-jp/5857d7725f3b14cbc8ec3e878e4307ce .

```
#from https://gist.github.com/n1n9-jp/5857d7725f3b14cbc8ec3e878e4307ce
def remove_emojis(data):
    emoj = re.compile("["
        u"\U00002700-\U000027BF"  # Dingbats
        u"\U0001F600-\U0001F64F"  # Emoticons
        u"\U00002600-\U000026FF"  # Miscellaneous Symbols
        u"\U0001F300-\U0001F5FF"  # Miscellaneous Symbols And Pictographs
        u"\U0001F900-\U0001F9FF"  # Supplemental Symbols and Pictographs
        u"\U0001FA70-\U0001FAFF"  # Symbols and Pictographs Extended-A
```

```
        u"\U0001F680-\U0001F6FF"  # Transport and Map Symbols
                          "]+", re.UNICODE)
    return re.sub(emoj, '', data)
```

After restricting the dataset as mentioned before and applying **cleaner** to all the Tweets in the dataset, we are ready to parallelize it into an RDD.

```
#select a portion of the english speaking tweets only
num_docs = 10000

text = df[df.language == "en"]

#select the textual column, the only one of interest
text = text.text[:num_docs]
text = text.map(lambda x: cleaner(x))
```

2.1 Tokenization

Between preprocessing and the actual topic of this paper stands **kgram**: this function simply creates k-grams, in this case 5-grams, from a text and returns the set of them, in order to avoid recurrent k-grams. Since this is the first operation of the algorithm, it will be applied only after the parallelization phase.

```
#create a function to split into 5-grams
K = 5

def kgram(sentence,k):
  ngr = set(ngrams(sentence, k))
  return ngr
```

3 Algorithm implementation

3.1 Characteristic Matrix

The first step to find similar items is to represent the corpus as a matrix, named characteristic matrix, where every row corresponds to a shingle, in our specific case to a k-gram, and every column corresponds to a document. The function **char_matrix** transforms the documents into sets of k-grams mapping them to the same RDD and then performs two operations. The first one aims to create a set consisting of all the distinct shingles contained in the dataset. This is going to be the row of the characteristic matrix. In this case, after using flatMap since we don't need to distinguish between elements coming from different documents, we use distinct to have a set of non-repetitive kgrams. Only after those operations we can be confident that storing the rows in the main memory by collecting them won't affect drammatically the need

for space in the memory. The following operation builds the column of the matrix one by one, expoliting another function, **col_cm**. In this case, a transformation is enough to have the desired result. Every column is a binary list: if the shingle is contained in the document, it will generate a 1; otherwise, the column returned will have a 0 in the position of the very same k-gram.

```
def char_matrix(rdd):

  #trasform tweets into sets of k-grams
  rdd = rdd.map(lambda gram: kgram(gram,K))

  #create the rows of the matrix, i.e. the set of all kgrams
  rows = rdd.flatMap(lambda gram: gram).distinct().collect()

  #create the columns of the characteristic matrix
  columns = rdd.map(lambda gram: col_cm(gram, rows))

  return len(rows),columns
```

```
#characteristic matrix column
#if a shingle is present in the tweet, the cell is equal to 1, 0 otherwise
def col_cm(gram, rows):
  col = np.zeros((len(rows),), dtype=int)
  for shingle in gram:
    col[rows.index(shingle)] = 1

  return col
```

3.2 Minhashing

Dealing with big data raises two main problems: time and space. In fact, comparing each row for every pair of documents would be highly expensive, thus we need a method to perform the same operation way faster. The one used, minhashing, happens to solve also the problem of dealing with a highly sparse matrix such as the characteristic matrix. It essentially consists in shuffling the rows of the characteristic matrix and building another matrix, called signature matrix, where each row reports for every document the position of the first 1 in the shuffled characteristic matrix. Since permuting the rows isn't implementable, we simulate it by a random hash function that maps row numbers to the same number of buckets. This preliminary step is contained in the function **shuffler** and repeated in **hash_matrix**.

```
#create the sequence of new indexes to reorder the characteristic matrix
def shuffler(len_r, rnd):
  shuffler = np.array([(i + rnd) * rnd % len_r for i in range(len_r)])
```

```
  return shuffler
```

```
#create a matrix of new sequences: each one works as hash function
def hash_matrix(len_r):
  hash_matrix = []
  rnd_lst = random.sample(range(0, len_r), H)
  for rnd in rnd_lst:
    hash_matrix.append(shuffler(len_r,rnd))
  return hash_matrix
```

The following step consists in creating the signature matrix. To keep a consistent approach, the matrix is compiled column-wise.

```
#for every text and every hash function, generate the signature
def hash_column(col,len_r,hash_matrix):
  sig_col = np.full(H,len_r)
  for i in range(len_r):
    if(col[i] == 1):
      for j in range(H):
        if(sig_col[j] > hash_matrix[j][i]):
          sig_col[j] = hash_matrix[j][i]
  return sig_col
```

```
#wrap up the signature of each text
def sig_matrix(hash_matrix,len_r,columns):

  columns = columns.map(lambda x : hash_column(x,len_r,hash_matrix))
  return columns
```

After the number of hash functions (hence rows of the signature matrix) is set to 100, we are ready to parallelize the data and create the matrix as an intermediate step.

```
#number of hash functions
global H
H = 100
```

```
#Eventually, start the process by parallelizing the tweets
rdd = sc.parallelize(text).cache()

len_r,columns = char_matrix(rdd)
sig_matrix = sig_matrix(hash_matrix(len_r),len_r,columns)
```

3.3 Locality-Sensitive Hashing

To address even better the space problem, a further level of hashing subdivides the signature matrix into bands, namely group of rows of equal length. Once they are hashed (in this case, exploiting the embedded Python function), we are left with a small number of rows to scan in seek of candidate similar items.

```
#define the number of rows per band
global r
r = 5
global b
b = H/r
b

global t
t = round((1/b)**(1/r),2)
t
```

Considering s the probability that two columns denote a candidate pair, the probability that the couple agree in all bands is $1 - (1 - s^r)^b$. Crucial is the choice of a threshold t. We approximate it as $(1/b)^{(1/r)}$, without enhancing neither the avoidance of false negatives by lowering to a smaller value, nor the velocity by performing the opposite operation. Hence, we keep $t = 0.55$.

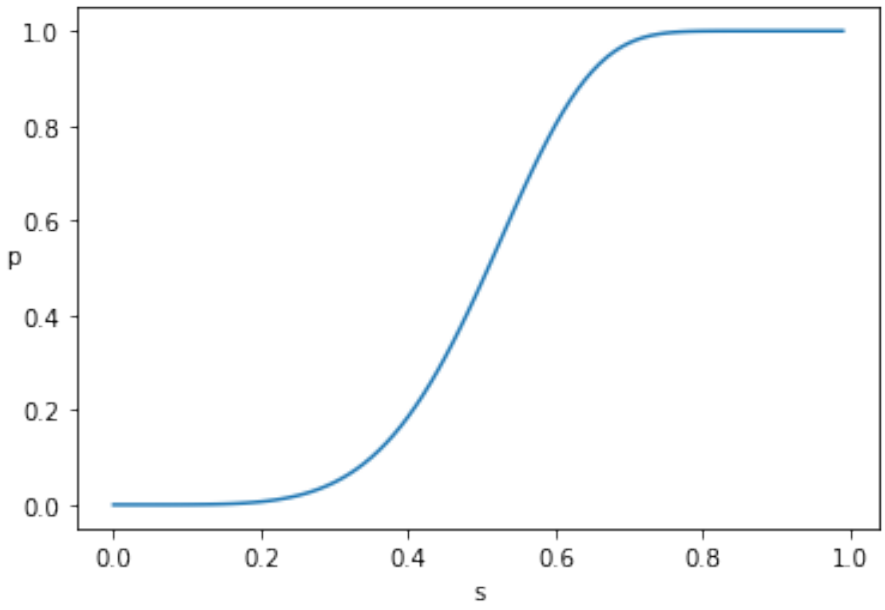

Figure 1: To choose a t value of s, we approximate the point of maximum steepness

The following steps are creating the bands and hashing them through the function **hash_array**:

```
#group by splitting every hashed document into b bands
def hash_array(x):
  return abs(hash(x.tostring())).to_bytes(8,"big").hex()
```

Defining a simple Jaccard similarity to count the rows in which two items agree over the ones they differ (in this case, it is extended to avoid considering the so called "z-case" where the rows are both zero in case one would want to compare the original characteristic matrix):

```
#Jaccard similarity between two documents
def j_sim(l1,l2,i,j,n_parts):
  jacc = sum([1 for ii, jj in zip(l1, l2) if ii == jj and ii != 0])/n_parts
  if jacc != 0:
    return (i,j)
```

And eventually hash the signature matrix dividing the entries into b bands through a map transofrmation. Moreover, we wrap everything into a function that takes as input the hashed matrix (the only element collected to the main memory except from the rows of the characteristic matrix), that is finally compressed enough to work with in a single computer. We compute the Jaccard similarity for every couple and save the ones the matches assessed into a list. In this analysis, since the main goal is to enable working with a massive dataset, we limit ourselves to calculate the percent of matching candidates by dividing the number of similar couples by the total number of them, without considering deeper insights such as groups of identical Tweets and the dimensions of those. Note that in order to save time we compare each couple only once: j_sim is in fact performed in loading *jc* with a nested for loop. This operation enables us to look for similarities only with documents denoted with a higher index. Therefore, the first element will be compared with all the others, the second only with them all apart from the first, and so on.

```
#hash every column to a bucket (an integer) in each band
res = sig_matrix.map(lambda x: [hash_array(arr)
        for arr in (np.array_split(x, b))]).collect()
```

```
#the final wrapper takes the lsh-hashed documents and compares them
def results(res):
  global ls_sim
  global jc
  len_res = len(res)
  #list of candidates
  jc = [(j_sim(res[i],res[j],i,j,b))
        for i in range(len_res-1) for j in range(i+1,len_res)]
  jc = [i for i in jc if i is not None]

  #get the list of items involved in similarities
  ls_sim = list(itertools.chain(*jc))
  ls_sim = set(ls_sim)

  #the percentage of similar couples is calculated
  #by dividing the number of similar couples by the number of total couples
  sim_perc = len(jc)/binom(num_docs, 2)*100
  sim_perc = round(sim_perc, 3)
  print(f'The percentage of similar items is: {sim_perc}%')
```

3.4 Optional: final check

One may be interested in confirming at the document level that the elements detected effectively correspond. In order to check it, it is sufficient to slightly rework the functions above, taking as input also the list of items involved in similarities. Clearly, this operation is possible only if the number of items in this list is small enough to collect them in the main memory, as well as to compare them in an affordable time. If it were not, it would make no sense to even perform the whole algorithm above. Considering the sample collected, where most of the items are involved in several similarities - you can check it by taking a glimpse of the variables *jc* and *ls_sim* - we are not in a case where it is possible to proceed in the following steps.

```
def j_sim2(l1,l2,i,j,n_parts):
  jacc = sum([1 for ii, jj in zip(l1, l2) if ii == jj and ii != 0])
    /(n_parts - sum([1 for ii, jj in zip(l1, l2) if ii == 0 and jj == 0]))
  if jacc >= t:
    return (i,j)
```

```
def final_results(res):
  global jc

  res = [(sub[1], sub[0]) for sub in res]
  res = dict(res)

  jc2 = [(j_sim2(res[jc_tup[0]],
        res[jc_tup[0]],jc_tup[0],jc_tup[1],len_r)) for jc_tup in jc]

  sim_perc = len(jc2)/binom(num_docs, 2)*100
  sim_perc = round(sim_perc, 3)
  print(f'The percentage of similar items is: {sim_perc}%')
```

Since the columns are stored as an RDD, the only way to filter them using their indexes is to use zipWithIndex() before filtering. The chosen columns are subsequently gathered in the main memory.

```
res1 = columns.zipWithIndex()
res1 = res1.filter(lambda x : x[1] in ls_sim).collect()

#check specifically if the documents are identical
final_results(res1)
```

3.5 Results

The percentage of similar items, namely in this case couples of similar Tweets over all the possibile combinations, is around 0.7%. Thus, the analysis assesses the presence of a significant number of duplicate Tweets, altough this paper does not have

interest in scouting whether the causes be plagiarism, malicious purposes or other reasons. Checking the similarity also at the document level has given no significant contribution.

4 Conclusions

This project implements an algorithm to find similar items on a corpus of Tweets concerning the Ukraine Conflict exploiting a series of techniques to deal with massive datasets using a Spark context. The analysis, performed on a sample of 10000 English speaking Tweets, highlighted that comparing two random different Tweets from the dataset will end up in finding out they share the same textual part in about 0.7% of the cases, a considerable sum.