Finding similar items from Twitter dataset

Anton Shchedro

Università di Studi di Milano, Milan, Italy
anton.shchedro@studenti.unimi.it
Matriculation number: 989297

Abstract. 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.

This paper describes an implementation on the problem of finding similar items from a large dataset using LSH (Locality Sensitive Hashing) method described in a book *Mining of Massive Datasets* by Jure Leskovec, Anand Rajaraman and Jeff Ullman [1].

Keywords: LSH

1 Introduction

Finding similar tweets from a large dataset of tweets using the native technique by simply comparing pairs of tweets is time consuming, making the process impossible in practice. LSH is a method that reduces the number of comparisons, but requires additional memory usage and with some probability of false negatives (when a similar item exists but is not found by the algorithm).

1.1 Potential applications

A possible application of this system could be to search for duplicates, whether they have identical text or have minor changes, such as different links or few different hashtags in the text, and/or similar context of tweets.

2 Dataset

This paper uses the *Ukraine Conflict Twitter Dataset* [2], a dataset of tweets that provides an overview of various people's views on Twitter about the *2022 Russian invasion of Ukraine*.

This dataset consists of several compressed *csv* files, each of which is a list of tweets made on the day specified in the prefix of the file name (in the format *MMDD_*, where MM is month, DD is day).[1].

Each file contains a list of tweet information, such as

- *userid*, *username* and other account information
- *tweetid*, *language*, *text*, *hashtags* and other information about a particular tweet
- other additional information

Due to the nature of the task, not all tweet information is needed, so only *tweetid*, *language* and *text* were used.

In this paper I used the version of the dataset as of 26/12/2002, so only tweets up to 25/12/2002 were used.

3 Pre-Processing

Files was first loaded as Pandas DataFrame. Due to the need for stop-words and for ease of experimentation, the DataFrame is filtered to contain only tweets in English. In case of necessity and availability of other language stop-words, other language can be used with small code modification[2]. The filtered DataFrame is then loaded into a Spark RDD as a list of pairs of *tweetid* and *text*.

3.1 Tokenize

To simplify the text, emoticons and URL links are removed. The resulting text is then tokenised by changing the text to lower case and splitting it into words, then removing any stop words.

3.2 Shingles

Tokenised text is then merged into a shingles that in the case of this work I make it as a group of 2 words.

For example, the list of words [*a,b,c,d*] is transformed into a list [*'a b','b c','c d'*].

However, this simple method of creating shingles has a problem: some pairs of shingles that contain the same words but in a different order may contain the same meaning, but will be seen by the algorithm as two different shingles. For example, "*Russia Ukraine*" and "*Ukraine Russia*" obviously have the meaning of the relationship between these two countries, but they are treated as two different shingles.

[1] All files from 1 January 2023 have a new prefix: YYYYMMDD_
All files from 27 February to 31 March have a format of UkraineCombinedTweets-Deduped_MMMDD, where MMM is the first 3 letters of the month.

[2] Technically it is only necessary to change the filter to another language and add the required stop words

This problem is not easy to solve, because it's not obvious which of a pair of shingles is primary and which is secondary and can be suppressed. Furthermore, not all of the pairs may have the same meaning.

At this point, the RDD record will look like a pair of *tweetid* and list of shingles.

4 LSH

The basic idea of LSH is to use multiple hash functions over data to sort it into multiple buckets, each of which may contain similar items.

For this work first Min-Hash was used to create a *signature* of a *text*, and then corresponding *twittid* is placed in a multiple bucket based on another hash functions on text *signature*.

4.1 Min-Hash

The general idea of min-hash is to create multiple permutation functions for each itemset and for each permutation find the first element that the itemset has in that permutation. The native solution could be to create n permutations of all possible items and store them in memory, but this will require a large amount of memory, so I used multiple hash functions, one per permutation.

These hash functions are a slight modification of the djb2 hash function by Daniel J. Bernstein:

1. Added mod operator p of a prime number, unique for each permutation
2. Added mod operator m, which limits the maximum possible value of the result.

The resulting algorithm looks like this:

```
def string_hash(text, p, n):
  h = np.uint32(5381)
  for i in range(len(text)):
    h = ((h*33) + ord(text[i])) % p
  return h % n
```

Additionally, I didn't use iteration over records for each hash function, but for each record I used iteration over hash functions. So for each record, the calculation of its signature via Min-Hash looks like this:

```
def signature(text, n, m, primes):
  # text - list of shingles in one record,
  # n - num of hash functins,
  # m - max value of hash
  # prines - list of prime numbers

  signature = [np.inf]*n
```

```
for i, p in enumerate(primes):
  for shingle in text:
    h = string_hash(shingle, p, m)
    if h < signature[i]:
      signature[i] = h
return signature
```

where primes is a list of the first n prime numbers greater than m.

At this point, the RDD record looks like a pair of *tweetid* and a list of hash values that are minimum for each permutation.

4.2 LSH

To create LSH from signature, its possible to divide signature into a b bands, each containing n/b Min-Hash values of the signature. Each band can be mapped to one of k buckets, with total number of buckets for all bands of $k*b$.

I use hash function on each band of the record to find index of a bucket of corresponding band. To do this, for each band, the sum of that part of the signature is taken and then, using mod k operation, mapped to corresponding bucket:

```
def lsh(text, b, k):
  l = [None]*b
  r = round(len(text)/b)
  for i in range(b):
    l[i] = (np.sum(text[r*i:r*(i+1)]) % k) + i*k
  return l
```

At this point, RDD is a pair of *tweetid* and a list of bucket indexes. Then RDD is transformed into a pair (bucket index, *tweetid*) and grouped by bucket index.

The resulting RDD can be used for searching and optionally saved to disk.

5 Search

Search of duplicates or similar items for a tweet (in following called *target*) is made by calculating its LSH bucket indexes and for each its index a list of *tweetid* received and then number of occurrences if *tweetid* is counted. Similar tweets are received by taking only thous *tweetid* that have number of occurrences grater or equal to needed level of similarity.

The search for duplicates or similar items for a requested tweet (hereafter called *target*) is done by calculating its LSH bucket indexes, and for each its index a list of *tweetid* is received and then number of occurrences of *tweetid* is counted. Only those *tweetid* whose number of occurrences is greater than or equal to the required similarity level are taken as a similar tweets.

6 Additional (optional) modifications

6.1 Saving LSH to disk

The proposed implementation can take a long time to compute LSH from the entire dataset, and as a result, searching for similar tweets will also take a long time. To speed this up, LSH needs to be cached in memory or stored on disk.

It's possible to divide a dataset into multiple subsets, calculate LSH of each subset and then save it to disk. This will require a large amount of time to save all LSH of each subset, but at next run of a program only loading of saved LSH is needed (with loading LSH from files to a multiple RDDs and union of these RDDs).

6.2 Saving list of records

In case if LSH was loaded from files (read "**Save LSH to disk**"), and if we need to keep not only LSH, we can save a list of *tweetid* and file containing it. If we need to read additional information about tweets returned by a search, we can extract the information not by searching all files for the needed record, but by loading it directly from an appropriate file.

7 Scalability

The scalability of this method can be broken down into several parts:

1. Calculating the LSH table
2. Search

7.1 Calculation of LSH

To properly calculate LSH, the algorithm needs to iterate through the entire dataset (or subset, depending on where we want to search) to collect the list of *tweetid* for each bucket. The time needed for this increases linearly based on the number of records to process. For example [3]:

- 1 file *0820* [4] contain 20103 records and are processed for a total of 224,041 s (0.011s/record)
- 2 files *0820* and *0821* contain 40345 records and take a total of 443.232 s (0.011s/record) to process.
- 3 files *0820*, *0821* and *0822* contain 62513 records and take a total of 665.231 s (0.011s/record) to process.

[3] Computer specifications on witch was run this test:
Intel(R) Core(TM) i7-6700 CPU @ 3.40GHz (8 core)
32GB RAM
SSD up to 550MB/s Read

[4] Full name of file: 0820_UkraineCombinedTweetsDeduped.csv.gzip
This and following file names was reduces up to month and days

7.2 Search

Searching for similar records requires calculating LSH buckets of target record, which is done in constant time, and then list of *tweetid* is taken from corresponding buckets of LSH and number of occurrences is calculated via reduce method. The calculation time of these operations may be different depending on the option chosen to store LSH:

- If LSH is stored in cache, sometimes, if needed memory is more than system can provide, some values of LSH have to be recalculated, which can take time, in worst case to recalculate whole LSH (same happens if LSH is not stored in cache, not on disk).
  The amount of memory needed is proportional to number of bands and number of buckets. For example, storing the LSH cache of the file *0820* requires:
  1. for 20 bands and 1000 buckets: 173'016 bytes
  2. for 20 bands and 100 Buckets: 16'184 bytes
  3. for 10 bands and 1000 Buckets: 85'176 bytes
  4. for 25 bands and 1000 Buckets: 219'064 bytes

- If LSH is stored on disk, only additional time is required to read LSH files from disk.

Then, an additional filter on the number of occurrences is applied (so that only records that meet the required similarity level are kept), and the Jaccard similarity between the target and each tweet can be calculated from the resulting list.

Search time with LSH stored on disk (without Jaccard similarity calculation), computed from files starting from *0819* to *1225* (129 files in total) and grouped by 9 files, takes on average 1.3 min. [5] [6]

8 Results

The test of the method described in this paper focuses on computation time rather than accuracy, due to the lack of a ground truth list of similar tweets and a large dataset. The test was run on several combinations of values of the variables used in the method and on a small subset made from the *0820* file (chosen randomly).

Combinations of values are given below:

[5] Using the same computer specifications as in **Calculating LSH**
[6] Time of LSH calculation was not included, read **Calculation of LSH**

Num. of combination	Num. of Bands	Num. of Buckets	Num. of Tokens	Num. of hash functions
1	20	1000	2	100
2	20	100	2	100
3	20	1000	2	200
4	10	1000	2	100
5	25	1000	2	100
6	20	1000	1	100
7	20	1000	3	100

For each combination of values, LSH was computed once and 100 searches were performed and measured using random target tweets. The following table shows the time to compute LSH for a subset and the mean time to compute *Target signature* and the mean time to search for similar tweets based on the target signature (*Target search time*).

Num. of combination	LSH computation time	Target signature time	Target search time
1	235.452s	0.040s	12.781s
2	218.301s	0.039s	12.752s
3	414.574s	0.087s	12.642s
4	224.258s	0.042s	12.693s
5	216.654s	0.040s	12.692s
6	140.104s	0.025s	12.857s
7	280.835s	0.051s	12.850s

Additionally, some random tweets were chosen to compare the search results of the method using different combinations of values, taking all tweets that are in at least 3 same buckets as the target tweet (2 buckets give too many FP results). Jaccard similarity of these tweets was done by comparing their list of tokens. Checking the answers of the search, we can see the following properties:

- 1st combination: Shows mostly TP where the text is similar to the target text, but may have some grammatical errors such as missing spaces between words. They have a Jaccard similarity between 1.0 and 0.9 and a number of shared buckets between 9 and 20 (out of 20). Additionally, some tweets can be seen as TP while they have small modifications, such as different user mentions (@user) or hashtags, but the same text, with Jaccard similarity from 0.8 to 0.5 and number of same buckets between 3 and 6. The result may have a small number of FP tweets (Jaccard similarity less than 0.5, but number of same buckets between 3 and 6) and some number of FN tweets (similar tweets that are missed in this result but are present in other combinations).
- 2nd combination: It has all the tweets from the 1st combination, plus a lot of FP tweets and additional TP tweets that were FN in the 1st combination.
- 3rd combination: keeps a portion of tweets from 1st combination with strong relationships (with number of same buckets between 9 and 20 in 1st com-

bination). Number of same buckets is lower for same pair of tweets in compression with 1st combination for each pair except tweets with maximum possible value of same buckets.
- 4th combination: keeps only part of tweets from 1st combination with strong relationships (with number of same buckets between 11 and 20 in 1st combination). It's much more string than 1st combination because of smaller number of bands, and as a result number of same buckets in compression is lower than in 1st combination with maximum possible value 10.
- 5th combination: Due to larger number of bands, this combination may find more tweets that were FN in the 1st combination. This combination have greater number of same buckets of similar tweets than in 1st combination, but sometimes may miss some tweets that in 1st combination have low number of same buckets. The maximum possible number of same buckets is 25.
- 6th combination: Due to the small number of tokens in a shingle, it returns many FP results, some of them with very low Jaccard similarity, while having the maximum number of matching buckets.
- 7th combination: Because of the larger number of tokens in a shingle, it gives a more severe result, which mostly has a high Jaccard similarity, but can still find some FP result, even with less probability.

Depending on the required Jaccard similarity, we can adjust the method variables to suit our needs:

- If we need to look for identical tweets, we can take the 1st combination and filter the answer with a higher number of similar buckets, or increase the number of tokens, or take a higher number of hash functions, but a higher number of hash functions will make the LSH calculation time longer.
- If it is necessary to find not only identical but also similar tweets, we can take the first combination and filter the answer with the number of similar buckets equal by 3, or additionally lower the number of buckets in the algorithm. However, in this case it is strongly recommended to check the Jaccard similarity, because there may be a certain number of FP tweets.

As a result, with the right combination of method variables and Jaccard similarity value, LSH, based on this implementation, can provide a list of similar tweets in a short time.

References

1. Jure Leskovec, Anand Rajaraman, Jeff Ullman: Mining of Massive Datasets. 3rd edition. Cambridge University Press
2. Ukraine Conflict Twitter Dataset. Kaggle, 2023. https://doi.org/10.34740/KAGGLE/DSV/4831911