Algorithms For Massive Datasets: Finding Similar Items

Matteo Onger

July 2023

1 Project Description

This project aims to implement the MinHash-LSH technique to find similar reviews. The final output of the algorithm is a list of pairs of documents with a hopefully high Jaccard index (JS). The exact Jaccard similarity coefficient is not computed, but for all these pairs we calculate the estimated Jaccard index (EJS) and it is proven that

$$M \to \infty \;\Rightarrow\; JS(x, y) = EJS(x, y) \tag{1}$$

with M $(= B \cdot R)$ equals to the number of MinHashes computed for each document. Obviously, the bigger M is, the higher the computation costs will be, so the value of M must be balanced. Of course, we could easily compute the Jaccard index for some specific pairs of documents, perhaps of greater interest.

The parameters that can be set to customize the execution of the algorithm are:

- J: number of documents to analyze;
- N: all the shingles will be represented as an integer between $[0, N)$. So some collisions may occur, especially if the number of distinct shingles is bigger than N;
- B: number of bands used in the LSH technique;
- R: number of rows per band used in the LSH technique;
- F: function used to compute the shingles of a document. Four different functions have been already implemented:
  - `words`: every word in the document is a shingle;
  - `nonStopWords`: every word in the document is a shingle, but stop-words are removed;
  - `joinStopWords`: every word in the document is a shingle, but stop-words are joined to the next two words to create a single shingle;
  - `kgrams`: shingles are K-grams;
- K: length of the k-grams produced. This value is used only if the function `kgrams` is employed to compute document shingles.

These parameters obviously affect the accuracy of the results produced. At the same time, greater precision implies greater computational effort, so we should always consider the computational power available and the volume of the data to be processed.

2 Workflow Description

Analyzing the workflow of the algorithm, initially a Spark RDD (`dataRDD`) is created by reading the original dataset, but only two fields are retained:

- *review_id*: a string to uniquely identify a review;
- *text*: the body of the review.

So, from this moment on, we will only ever work with Spark, RDD, its standard methods and some functions/classes specially implemented. To use the methods of the latter, we need to instantiate three objects, one for each class:

- A `CharacteristicMatrix` object (`CM`) to compute characteristic vectors and, possibly, the Jaccard index;
- A `SignatureMatrix` object (`SM`) to compute signatures and the estimated Jaccard index;
- A `LSH` object (`L`) to apply the LSH technique and, by doing so, find pairs of candidate reviews as similar.

The exact behavior of these objects (how shingles are computed, the number of MinHashes or the number of bands used, etc.) depends on the parameters set, which are in fact passed as arguments to the constructors.

The first map function applied is used to preprocess the raw data: all the non-alphanumeric characters are removed and replaced by a white-space, while all the letters are made lowercase. Multiple consecutive white-spaces are collapsed into a single white-space.

Now we can compute the signature for each review using the appropriate method of `SM`: it calls `computeCharVect` of `CM`, which in turn uses the function F to compute the shingles of a review. Initially shingles are strings, but then they are mapped into integers using the following formula:

$$shingleIdx = hash(shingle) \mod N. \tag{2}$$

Basically we are applying the one-hot encoding: $shingleIdx$ uniquely[1] identifies a shingle and it coincides with its own row index, that is, its position within the characteristic vector. So these shingles' indexes are the only data we need to know. To compute M MinHashes, M permutations of the characteristic vector are necessary, but we can obtain the same result by simply computing, for each row, its position in the considered permutation: the R-th row of the original characteristic vector, in the i-th permutation, will be the $permR_i$-th row of the permuted characteristic vector, with

$$permR_i = ((\alpha_i \cdot R + \beta_i) \mod \gamma_i) \mod N. \tag{3}$$

Each α_i and β_i are random integers between $[0, N)$, while γ_i is a random prime number between $[N, 2N)$. A more detailed explanation about this family of hash functions and why they are great for this type of tasks can be found in the paper [1].

After calculating the signatures, the next step is to assign each review to a bucket for each band: this is done by using as a flat map the method `L.computeBuckets`. The key-value pairs of `dataRDD` now have the following structure:

```
(
    (band_i, bucket_j),
    {'review_id':ID, 'sign':signature}
).
```

Then, using the Map-Reduce-Filter framework, a collection, containing all and only the buckets with more than one document, is produced. Obviously, we can find similar reviews only in these buckets. Consequently `dataRDD` is filtered, and all the buckets that do not appear in this collection are discarded.

Finally, we can group the RDD by key: for each bucket in each band we have a list of all the reviews assigned to it. We know the ID and signature of each review in this list, so we can easily compute the EJS of all the possible pairs. Having done this, we have exactly the desired output. The MinHash-LSH technique remains a probabilistic algorithm, so different executions may produce (hopefully slightly) different results and there may be false positives (FP) or false negatives (FN). But it is proven that, by setting the parameters wisely, we can make the probability of getting FP or FN acceptable, if not negligible. And of course, to remove all the false positives pairs, we could filter the resulting list again, keeping only distinct pairs with a degree of similarity, either exact or estimated, above a certain threshold.

[1] We do not consider possible collisions, which remain unlikely for good hash functions and for a reasonable value of N with respect to the total number of possible distinct shingles.

3 Implementation Details & Computational Cost Analysis

In this section, we try to analyze the computational costs. To do this, some implementation details need to be considered, as they deeply affect the performance. The algorithm can be summarized in the following steps:

```
START
for each document:
    | preprocess                    (1)
    | compute shingles              (2)
    | map shingles to integer       (3)
    | compute signature             (4)
    | assign to buckets             (5)
[opt: filter and] group by key      (6)
for each bucket:
    | for each pair:
        || compute EJS              (7)
END
```

So by analyzing these steps, we can show that:

- For the preprocessing (1) and the computation of shingles (2), it depends on the functions used, but at least for the most common ones, a single iteration on the entire document is sufficient to obtain the desired output. To store shingles, Python `set` is perfect because it allows data to be entered and retrieved in constant time, while ensuring the uniqueness of the elements present. So, by doing this, the time complexity is $\mathcal{O}(L)$, with L equal to the length in characters of the processed review. Only if we're working with very large documents, these two steps can be a bottleneck.

- During the mapping phase (3), the time required to apply the hash function to a shingle is negligible for two main reasons: first, good hash functions should be fast to compute, and second, the shingles are usually small in size, so they should be processed in no time at all. Therefore the time complexity of the mapping procedure is linear with respect to the number of shingles in a document. This number is obviously affected by the function F, but definitely, it is less than L.

- By definition, the MinHash is equal to the lowest row index, in the permuted characteristic vector, associated with a shingle in the document. The permutation is computed thanks to the hash function 3, for which the consideration in the previous paragraph applies, and all and only the indexes of the shingles that appear in the review are exactly the information saved in the Python `set`, that represents the characteristic vector. So we have what we need to calculate the signature (4) efficiently, and, to be even faster, we can compute all the M MinHashes in parallel by viewing them as a `Numpy.array`, which uses SIMD type parallelism to enhance the performance. In light of this, and given that usually $M \ll \min(L, N)$[2], the final time complexity is equal to $\mathcal{O}(\min(L, N))$.

- Each review is assigned to a bucket for each band by applying an hash function to a different sub-vector of the signature. Keeping the previous assumptions about the hash function valid, the most expensive operation, which will be repeated B times, is the slicing of the signature and its cost depends on the length of the slice produced, which in this case is equal to R. So, for the step (5), the time complexity is equal to $\mathcal{O}(B \cdot R) = \mathcal{O}(M)$.

Note that, following this approach, all the steps performed so far are the body of a loop that iterates over all documents, so all the reviews can be processed in parallel and the only limit to the degree of parallelism is the number of compute node we can exploit.

The grouping by key (6) is the real bottleneck, because it requires redistributing $B \cdot J$ elements, having the structure 2, in different nodes and so, usually, it turns out to be by far the slowest step. To reduce the workload, a collection of all and only the buckets containing more than one document is computed by using the Map-Reduce-Filter framework: this job is computationally less costly because it considers only the keys of the elements, neglecting the remaining fields.

[2] $\min(L, N)$ is an upper bound of the number of shingles per review.

The price to be paid is that this collection must be broadcast to all the nodes, nevertheless this approach remains advantageous, especially if the computational power is a greater constraint than memory or transmission speed.

It has already been mentioned that after the grouping we get an RDD in which elements are a key-value pairs where: keys are a bucket of a band and values are a list of reviews.

```
(
    (band_i, bucket_j),
    [{'review_id':ID, 'sign':signature}, ...]
)
```

Each pair of documents of a list could be a pair of similar reviews, so analyzing the last step:

- The estimated Jaccard similarity is the ratio of the number of MinHashes that are equal, comparing the two signatures element-wise, to the total length of the signature. Thus, the time complexity of this final step (7) is linear with respect to M.

Once more, the only limitation on the degree of parallelism is the number of computing nodes available, but most benefits are obtained by paralleling the processing of buckets. In fact, although the number of possible pairs y, given a list of x documents, is equal to

$$y = \binom{x}{2} = \mathcal{O}(x^2) \tag{4}$$

and so it grows fast, in practice x is not a problem because we can force x to be small by wisely setting the parameters. And this is fine because it makes no sense to work with large x, since if it were too big, all the advantages of the LSH technique would be lost.

In conclusion, in light of the fact that:

1. Spark provides high scalability and excellent use of distributed systems;
2. Except for the grouping by key, the only limitations to the degree of parallelism depend on the number of compute nodes available;
3. Some data structures, such as Python `set` and `Numpy.array`, ensure easy data access and efficient processing;
4. All the phases have a linear time complexity and the memory footprint required does not grow exponentially.

We can be confident about the scalability of the algorithm.

4 Experimental Results

The following section reports some experimental results. Computation times related to different executions were collected, each time with different parameters. Unfortunately, it was not possible to run a test in a real distributed system; the machine used had the following characteristics:

- Cores: 2 (2.3 GHz)
- RAM: 13 GB
- Disk: 100 GB

The following table contains some average times collected changing only the function F used to compute the shingles.

Test	#Reviews J	Max #Shingles N	#Bands B	#Rows R	Function F	K-gram size K	Execution Time (sec)
A	25.000	250.000	5	10	$words$	---	36,12
B	25.000	250.000	5	10	$nonStopWords$	---	30,68
C	25.000	250.000	5	10	$joinStopWords$	---	39,95
D	25.000	250.000	5	10	$kgrams$	4	95,74

Table 1: Average computation time as the function F varies.

The results obtained seem to be reasonable: the function used to calculate the shingles affects the calculation time, not only because its execution may take longer, but also because the number of shingles produced depends on it, and this determines the complexity of the next steps.

The table 2 and the plot 1 show the data collected by changing the number of documents processed. Note that, with these values for B and R, the threshold, that is, the JS required to have a 50% chance of becoming a candidate pair, is quite high, being 0.85.

Test°	#Reviews J	Max #Shingles N	#Bands B	#Rows R	Function F	Execution Time t (sec)	Incr. Ratio $\Delta t/\Delta J$
01°	50,000	250,000	5	10	$nonStopWords$	44.22	---
02°	100,000	250,000	5	10	$nonStopWords$	100.11	1.06
03°	200,000	250,000	5	10	$nonStopWords$	212.42	1.12
04°	300,000	250,000	5	10	$nonStopWords$	381.60	1.09
05°	400,000	250,000	5	10	$nonStopWords$	600.18	2.19
06°	500,000	250,000	5	10	$nonStopWords$	899.30	2.99
07°	600,000	250,000	5	10	$nonStopWords$	1509.20	6.10
08°	700,000	250,000	5	10	$nonStopWords$	3124.37	16.15
09°	800,000	250,000	5	10	$nonStopWords$	3210.20	0.86
10°	1,300,000	250,000	5	10	$nonStopWords$	6942.91	7.47
11°	1,600,000	250,000	5	10	$nonStopWords$	9836.10	9.64

Table 2: Average computation time and incremental ratio as the number of reviews J varies.

Analyzing the data, it appears clear that something unexpected is happening: the execution time seems to grow approximately linearly with the number of reviews J, but the slope changes significantly. Such a big difference must depend on the algorithm and cannot be determined by the execution environment.

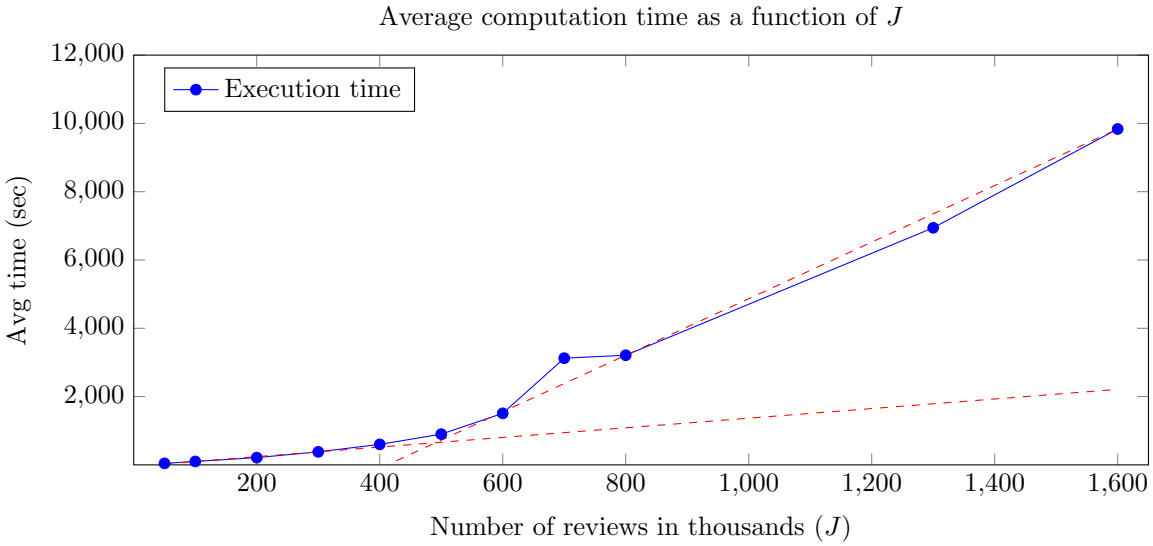

Figure 1: The chart shows the execution times of the table 2.

A possible explanation is the following: we have shown the linearity of the operations performed by the algorithm, and in our case it was not possible to exploit multiple nodes, so if $\mathcal{O}(Z)$ is the total complexity of the body of the first loop and we are processing J documents, the overall complexity is $\mathcal{O}(J \cdot Z)$, linear with respect to J. Therefore, if we double J, the overall complexity of the first part doubles, which is exactly the expected behavior.

The problem is in the second part of the algorithm: we know that the calculation of the EJS is linear to M, but the question is: how many more pairs of reviews, and/or buckets, will we have by increasing J, for example doubling it? Because we have already mentioned that this can be the most challenging part. The reality is that it is extremely difficult to calculate useful bounds for these values. Of course, if we double J, the maximum number of new pairs will be

$$\Delta p = \binom{J}{2} + J^2 = \mathcal{O}(J^2)$$

and we could have at most J new buckets, but in practice these values are far from the borderline cases[3]. Getting more precise estimates is tricky because they depend, in addition to the parameters chosen, on the similarity of the specific documents read, over which we obviously cannot exercise any control.

Thus, while the first part of the algorithm is linear with respect to J, and the second part is linear with respect to the number of pairs/buckets, the latter is not necessarily asymptotically equivalent to J and its irregular, as well as nondeterministic, course introduces irregularities in the computation time.

As proof of this, the table 3 shows the number of buckets with more than one document, the number of pairs processed and the number of similar reviews found as a function of J. While the figure 2 shows their incremental ratio, an approximation of the derivative.

Test°	#Reviews J	#Buckets b	#Pairs p	#Similar docs
01°	50,000	93	97	24
02°	100,000	353	380	95
03°	200,000	1420	1526	374
04°	300,000	2869	3122	817
05°	400,000	4538	4990	1260
06°	500,000	6793	7424	1960
07°	600,000	9464	10506	2852
08°	700,000	11512	12791	3463
09°	800,000	12448	13493	3770
10°	1,300,000	22147	25735	6793
11°	1,600,000	26636	33417	9430

Table 3: Number of buckets with more than one reviews, pairs processed and similar reviews found in the previous tests.

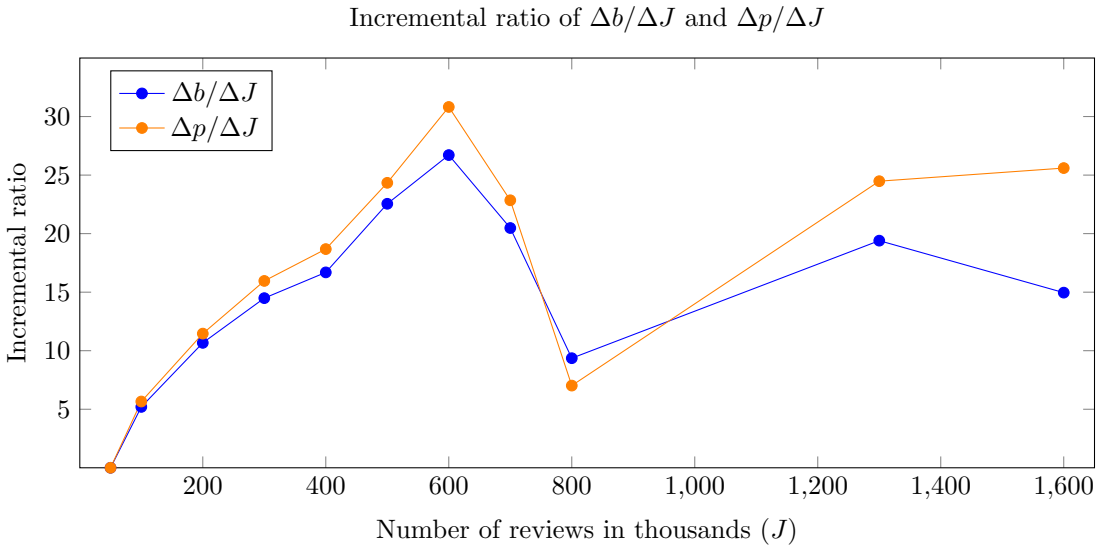

Figure 2: Incremental ratio chart of the number of buckets and pairs over J.

5 Conclusions

In conclusion, we can consider the performance of the algorithm satisfactory and suitable for real-world applications, obviously if implemented and set up carefully.
The parameters chosen deeply affect the execution and the results obtained, and they should be set to maintain at least a linear growth with respect to J of the number of buckets/pairs.
Depending on the function F, we are also defining the similarity between two documents differently. For example, if we are using the function `kgrams`, also the word order in a review can make the

[3] A reasonable choice of parameters and hash functions is enough to ensure this. If not, the algorithm would be difficult to use in real-world applications.

difference, whereas it makes no difference if we are employing one of the other functions.
Obviously, there may be some further improvements whose impact on performance depends very much on the specific case: if we were working on long documents, focusing on the efficiency of function F could bring good results. As well as, during the filtering, we could choose to broadcast the list of the buckets to keep or delete depending on which is the shortest, although to achieve real performance improvement, the buckets to be kept should always be the minority.
Thus, this work can be regarded as an interesting starting point for additional analysis and improvements.

Examples of similar reviews:

```
M: 50
F: nonStopWords
--------------------------------
JS:  0.15
EJS: 0.24
--------------------------------
Review A:
- Id: czqH49oQmoFQRWAXmmWBKQ
- Text: The food is good but please get better
    quality coffee!!!  It's very weak or just cheap.
    I have always had  a good meal for breakfast or lunch...
    The employees are very friendly and pleasant.
--------------------------------
Review A:
- Id: URpcr-j4RL64ClODEX5jOA
- Text: The people here are very friendly,
    pleasant and polite. Food always very hot
    and good quality. Fast and efficient service too.
```

Declaration

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.

References

[1] Alistair Sinclair, CS174: Lecture 15, (1998). URL: https://people.eecs.berkeley.edu/~jfc/cs174/lecs/lec15/lec15.pdf.