Università Ca'Foscari Venezia

Master's Degree Course in
Computer Science and Information Technology

Artificial Intelligence and Data Engineering

Learning with Massive Data

Assignment 3

Similarity search for document pairs

Author
Giovanni Costa

Student ID
880892

Academic Year
2022-2023

This report describes the implementation of two different algorithms for solving the well known problem of finding all the element pairs that are similar more than a given threshold. Experiments are performed on textual document collections, but the framework could be extended to all elements that could be represented as a numerical vector. Evaluations are made by comparing exact and approximate sequential algorithms on a single machine, and by analyzing distributed parallel algorithm on different number of nodes in a server cluster.
The analyses are executed on the ranking datasets "NF-Corpus"[3] containing data from about 3633 documents, and "SciFact"[4] composed by 5100 documents. Regarding instead the part implemented with Apache Spark's Python interface [1], this was evaluated using a subset of the whole data due to performance and time issues. The experimental setup for Spark phase was a server cluster (`https://www.unive.it/pag/30351/`) with 8 cores and up to 6 nodes (assumed to have fair job scheduling policy and no anomalous congestion) and for sequential phase was a mid-range pc with 8 cores.

1 Implementation and key points

The implementations are divided as said into two main parts, sequential and parallel distributed, and to sum up what's written more in details in the Jupyter notebook files, the major components of both are:

Preprocessing and sparse document embedding: From the string of text, after the application of some preprocessing method, sparse vectors were build. In particular, only words and numbers were extracted from original text data, that after was made in lower case and stemmed. Stop words are not removed in this case in order to not alter the text's meaning.
For convenience, TF-IDF implementation of Scikit-learn library was used. Indeed, it allows optimized computation due to the usage of NumPy and return, as vector representation format, the so called CSR (Compressed Sparse Row) matrix implemented by *Scipy* team: this particular object exploits the sparsity of the TF-IDF score and manage the elements very efficiently. The sparse embedding in addition capture more the lexical signals of relevance, so could be suitable for the wanted purpose.

For using data with the developed distributed algorithm, instead, some additional preprocessing steps are needed. In particular, every row of the compressed matrix was decompressed and sorted according to terms' document frequency (number of documents that contains a particular term) in decreasing order; in this sense also the index of non-zero elements was remapped to becomes in increasing order (this is important to apply prefix filtering technique that will be explained and to avoid multiple similarity computation by the different reducer workers). With this data, PySpark $SparseVector$ object row is finally created. Also $d*$, that contains the max TF-IDF score of every term among all the documents vector, was computed (will be used into the below explained Map function). So, at the end, a list of pairs containing $< doc\_id, \vec{d} >$ was spread through PySpark cluster's nodes using Resilient Distributed Dataset (RDD) (kind of distributed fault-tolerant vector) data structure.

Exact similarity computation: In this step, using the Numpy's method *dot()* the cosine similarity matrix between pairs was calculated (notice that only dot product between vector are needed because data have been already normalized in the previous process). One obtained these scores, they are filtered by scanning the half triangle of the symmetric matrix and only pairs that pass the threshold are maintained.

The parallel distributed implementation instead basically relies on *Map-Reduce* paradigm and, according to Spark file system and RDD implementation, data must be key-value pairs. The general idea is to attach to every document representation a key t equal to one of its term's index (in the Map phase); if two documents have the same key t then they have one term in common and similarity with respect to threshold can be obtained (in the Shuffle+Reduce phases). In addition, observe that for documents that don't share terms, no similarity computations are needed.
This simple idea, however, is not free of problems: if document have n terms this is replicated n times and if two documents share m terms after threshold filtering their similarity is computed and output m times. Hence, to overcome the first, Map part was implemented using $flatMap()$ PySpark's function that emits tuple $< term\_id, < doc\_id, \vec{d}, threshold >>$ using prefix filtering: basically this technique reduce the number of document replica associated with its term ids according to the property that if $sim(\vec{d_i}, d^*) < threshold$, by construction, document has no term shared with all the other documents. To solve the second, $Reduce$ part was implemented using $groupByKey()$ and $flatMap()$: here similarity between all the document that have the same term id t in common are computed, but, to ensure that only

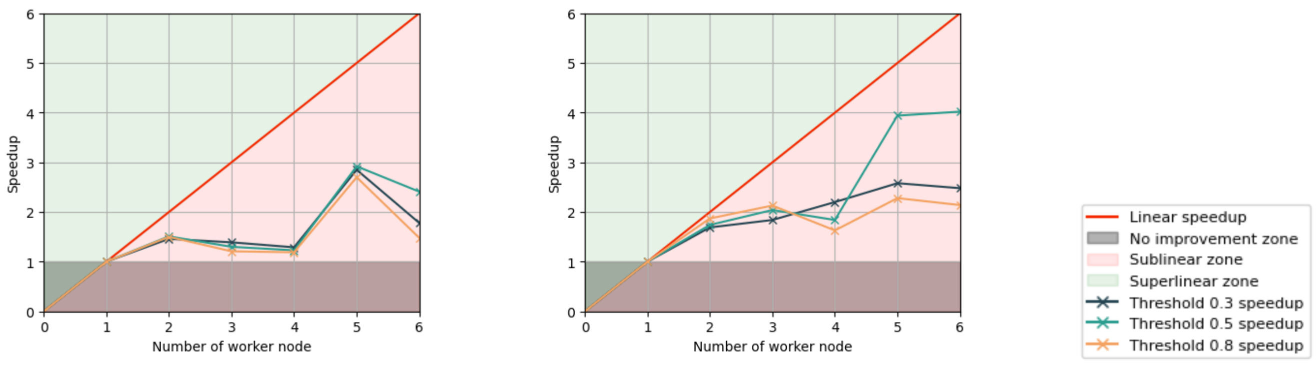
one reducer produces this operation, the calculation happens only when t it's equal to the max term id in the intersection between the two compared documents (each reducer receives the two full documents, so it can understand which are the terms in common without exchange messages with others).

Approximate similarity computation: Similarity calculation is the same as for exact sequential algorithm but, in this specific case, data are projected using a linear transformation to a lower dimensional space that has the property to maintain the Euclidean distance with a choosable distortion/error according to Johnson–Lindenstrauss lemma[2]. This was implemented using also in this case *SparseRandomProjection* class of Scikit-learn library and epsilon constant was fixed to 0.1.

This experiment is not exempt from critical choices, both regarding performance and efficiency. Focussing on what discussed before, CSR representation offer a lot of advantages in terms of space occupied and vectorized operation, especially the row-wise one, but are a bit slow for array random access operations by construction, bringing more memory overhead into procedures. Also PySpark's $SparseVector$, specifically designed for data processing using Spark efficient and distributed storage, suffer from the same problem. However, the algorithm is designed to make less random access as possible and, even in parts using for-loops, access is always to the non-zero elements only.
In addition to the optimizations already described in the Map-Reduce algorithm explanation, both PySpark $broadcast()$ function to distribute efficiently the d^* vector, and a dictionary to avoid returning both a pair and its inverse that in this case have the same semantic meaning were used.

2 Experimental evaluation

The data regarding the times spent to execute exact and approximate sequential algorithm on a single machine and the related Jaccard scores (for measuring the effectiveness of approximate solution) can be found in the Jupyter notebook file. Summarizing, version with Random Projection performs much worse than sequential exact version in terms of time due to less sparseness: in fact the CSR representation works well with very sparse vector but, in this case, dimensionality reduction bring to a lower sparse ratio and consequently more operations given by more non-zero elements.



(a) *NF-Corpus sampled dataset's plot* **(b)** *SciFact sampled dataset's plot*

Figure 1: *Speed-up plots of Spark implementation*

As can be seen instead from the plot in figure 1 regarding the distributed algorithm, speedup on different nodes is quite good but it never reaches a linear increment for any threshold value. In particular, the intuition behind the speedup jump in node 5 for both plot could be related to the fact that a lot of terms were skipped with prefix filtering and job partitions were sent to nodes in the more properly way. Also, the speedup trend of the other threshold scores don't change much.
However, the reason of these performances probably are due to the fact that overhead for managing low amount of data is higher than benefits of increasing node's number. One possible solution could be fine-tuning the different configuration parameters of Spark framework, for examples the number of data partitions or the number of task per nodes, to reach a better task granularity. Another improvements could be use some dedicated efficient libraries or data structures for high performance distributed vector computations.

References

[1] *Apache Spark Python API documentation.* URL: `https://spark.apache.org/docs/latest/api/python/`.

[2] William B Johnson. "Extensions of Lipschitz mappings into a Hilbert space". In: *Contemp. Math.* 26 (1984), pp. 189–206.

[3] *NF-Corpus dataset documentation.* URL: `https://www.cl.uni-heidelberg.de/statnlpgroup/nfcorpus/`.

[4] *SciFact dataset documentation.* URL: `https://github.com/allenai/scifact`.

[5] Stefan van der Walt, S. Chris Colbert, and Gael Varoquaux. "The NumPy Array: A Structure for Efficient Numerical Computation". In: *Computing in Science & Engineering* 13.2 (2011), pp. 22–30. DOI: `10.1109/MCSE.2011.37`.