LEARNING WITH MASSIVE DATA
Approximate Near-Duplicate Detection with SimHash

Serena Zanon 887050

Introduction The proposed assignment aims to compute the similarity between pairs of documents using Locality-Sensitive Hashing (LSH) with SimHash. The basic idea is to associate to each document a binary signature and to check pairs of signatures: the similarity is based on the equal number of bits that they have in common.
The code is implemented with PySpark utility.
Implementation To achieve the assignment's goal, the first important point is to obtain the binary signatures of the documents. To do so, some models and transformations are applied to each single document: after retrieving the TF-IDF vectors, with the support of library functions, each document is combined with a random matrix composed by only 1 and -1 as a way to have the final signature. These signatures are computed as follows:

- Each signature is initialised to a vector full of 0 whose dimension is m (m is a user-defined constant, m = [64, 128, 256]);
- Given the TF-IDF vector of the i^{th} document, each single one of its [i, j] cells is multiplied by the j^{th} column of the random matrix and the resulting vector is added to the initial signature;
- The final step is to convert the signature's values into 0 or 1: given the k^{th} cell of the signature, if its value is greater or equal than 0 then it is set as 1, 0 otherwise.

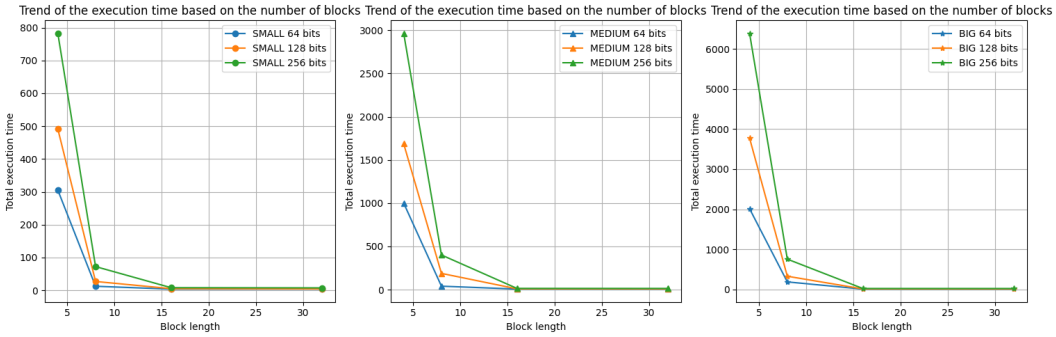
After obtaining the signatures, the map phase begins and it consists in dividing each signature into blocks of a pre-defined length (p = [4, 8, 16, 32]).
The next phase is the reduce phase: it groups together signatures that have the same block in the same position, it computes all possible candidate pairs of documents and then retrieves their similarity using the Hamming distance of their SimHash.
Just for instance, the small dataset contains 5.000 documents which means that there are 12.497.500 possible candidate pairs: if 2 documents share at least a piece of their signature then they are most likely to be similar and the number of similarity computations is drastically reduced.
Finally, only the pairs of documents whose similarity exceeds a pre-defined threshold (0.95) are maintained and considered as the actual similar pairs of documents.
PySpark As already mentioned, this assignment is implemented with the support of PySpark utility. In this section, some functionalities of this API are illustrated since they are used in the implementation:

- The principal adopted data structures are RDD and Dataframe;
- Functions *distinct()* and *dropDuplicates()* refer respectively to RDD and Dataframe data structures and their purpose is to delete all duplicated rows and to maintain just one of them. These function are expensive since identifying duplicates in a large data set can be computationally intensive, requiring a comparison of many elements;
- Function *cache()* persists the contents of the data structure across operations after the first time it is computed. This function is expensive since caching requires distributing the stored data evenly among the cluster nodes. This may involve a considerable transfer of data between nodes;
- Function *collectAsMap()* returns the key-value pairs in the associated RDD to the master as a dictionary;
- Functions *groupByKey()* and *mapValues(list)* permit to identify rows with equal key and to create a list of the remaining values of the row.

Analysis Most executions are computed with all the resources provided by the university cluster, so with 12 cores and 14 GiB. In the table below are presented the execution times obtained by the computations with block length only equal to 32, due to report space. (for full information see additional material).

Dataset	Num bits	Execution time
small	64	3.89521 s
small	128	4.48202 s
small	256	7.63725 s
medium	64	6.26071 s
medium	128	8.6298 s
medium	256	14.1854 s
big	64	8.76112 s
big	128	12.8303 s
big	256	23.9868 s

As presented in the full table of the additional material, the execution times become lower with an increasing block length and the number of signature bits. This is due to the fact that the implementation proposed is an approximation, this means that there could be false positive in the final result. However by adopting a bigger block length and a bigger number of signature bits, the number of possible matches is reduced and so the number of possible candidate pairs of documents on which compute the similarity. Here down below the trend that the execution time assumes based on the block length (p = 32).



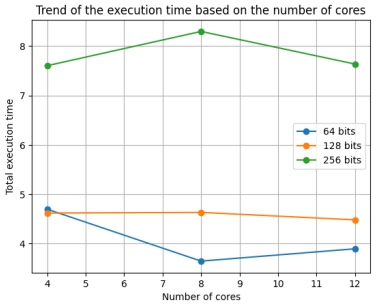
Despite the fact that more interesting analyses would have been derived from the large dataset, for reasons of execution time, further analyses were performed on the small dataset. In this sense, the number of duplicate pairs and distinct pairs on which the similarity is computed are retrieved and reported in the table below (the executions were done with full resources), while in the next table the execution times with less resources are shown (p = 16).

Num bits	Block len	Num duplicate pairs	Num distinct pairs
64	16	1614	1305
128	16	3762	2759
256	16	7321	4719

As it can be noticed, some duplicates are removed and this is a great news since the number of candidate pairs of documents on which the similarity is computed is reduced! Of course, in a larger dataset, the difference between the duplicates and the distinct pairs can reach hundreds, thousands and even millions pairs, benefiting computation.

Num cores	Num bits	Execution time
4	64	4.69728 s
8	64	3.64644 s
12	64	3.89521 s
4	128	4.61967 s
8	128	4.63238 s
12	128	4.48202 s
4	256	7.60397 s
8	256	8.29716 s
12	256	7.63725 s

As planned, with less resources the execution time increases since there are less core that can complete subtasks. Unfortunately, the differences aren't that big to be considered interesting, probably due to the fact that the dataset on which this test was run is small. Down below the trend of the execution time based on the number of adopted cores.



DISCLAIMER: the further analyses were made on the small dataset since not only *cache()* and *distinct()* are expensive, but even *count()*: it counts the number of rows of the data structure and it is expensive since it must traverse the entire dataset to count the elements and especially for large datasets this would lead to high execution time.