Finding All Pairs of Similar Documents

Riccardo Zuliani 875532

Assignment Topic

The topic of this assignment is to perform some benchmarks comparing the sequential version and the PySpark version (using the Map & Reduce pattern) of an algorithm that aims to find the similar documents pairs that exceed a given threshold within a set of document.

Proposed Solution and Keys Choices

The first step was to be able to read the documents of a given dataset, for this assignment we decided to adopt only a single dataset the nfcorpus dataset, however our application can handle multiple ones. After obtaining all documents set we performed a sampling of a given number of documents and then some pre-processing steps in order to clean up the set. Obtained the cleaned dataset we saved it in a *.parquet* file in order to use it for later debugging without computing the pre-processing at each run. Then was time to feed our documents set into the TfidfVectorizer that convert our collection to a NumPy matrix of *TF-IDF* features. We decide to opt to use a NumPy matrix since we can easily apply useful methods for later computation. An other strategy could be using the *scipy.sparse.csr_matrix* which lead to a more compact representation of sparse row but we decided to stick with the NumPy version. Having the TF-IDF matrix we moved to the implementation of the sequential version, precisely we implement the pseudo code provided by the professor in class, in which as *cosine_similarity* function we used the famous sklearn version. Moreover we performed a slight optimization in the *two nested for loop*, in particular we force the second for loop to start with the index equal to the first for loop index plus one, doing so we exclude pairs that we have already seen. Additionally we decided to look for a more efficient version. In particular *np.argwhere* caught our attention since it can perform in an efficient way the positions of a given NumPy array that satisfy a given condition, in our case the condition was that the cosine similarity between a pair of documents has to be greater or equal than a given threshold.

Now regarding the PySpark implementation, first of all we downloaded the compressed package from the slide file as professor suggested in class, then we moved the uncompressed directory into the */opt* directory and we configured the new environment variables in the *profile* file in order to be able to start *master* and *workers*. However in order to start and use the PySpark modules we installed an additional package called findspark that add PySpark to *sys.path* at runtime, this allow us to use all PySpark functionality in Python. Regarding the *SparkSession* configuration we set *spark.executor.memory* and *spark.driver.memory* to 10 GBs and *spark.executor.cores* to 1, this last configuration was needed since we want to assign a single core for each worker. In case we do not add this configuration, spark will automatically give to whatever number of workers we impose the permission to use all the available cores. So even in the case there would be a single worker we will have all the available core at 100% usage. One of the key part of PySpark implementation is the RDD (Resilient Distributed Dataset) which has the *parallelize* method to distribute a local Python collection to form an RDD that has the interesting attribute *numSlices*, which indicates the number of partitions of the new RDD. Commonly users suggested this value to be a multiplier of the number of current running workers so cores, we decided to run the PySpark version within a given set of factors as the benchmark part explains. Now the implementation of the PySpark algorithm was divided into 3 evolution steps, the one that caused us the most problems was the second evolution, in which we had to implement the so-called *Prefix Filtering*. It was difficult because we misunderstood the sorting point, indeed we apply the index sorting with respect to the *TF-IDF* value in each row, but one of the correct methods was to make a global sorting of all the sets of documents with respect to the decreasing term document frequency. Furthermore regarding the actual implementation of the **Map & Reduce** pattern, we decide to make a slither longer transformation pipeline, indeed we apply *flatMap* (as **Map**) $\rightarrow$ *GroupByKey* $\rightarrow$ *flatMap* (as **Reduce**), this decision was made since in our opinion this looks more intuitive and easy to debug instead of having the last two transformations in a single one. Finally after having developed the whole application structure we decide to add a saving step that memorize two *.csv* files, one with some useful statistics for each execution types the other containing the pairs of documents that exceed a given threshold with the relative cosine similarity.

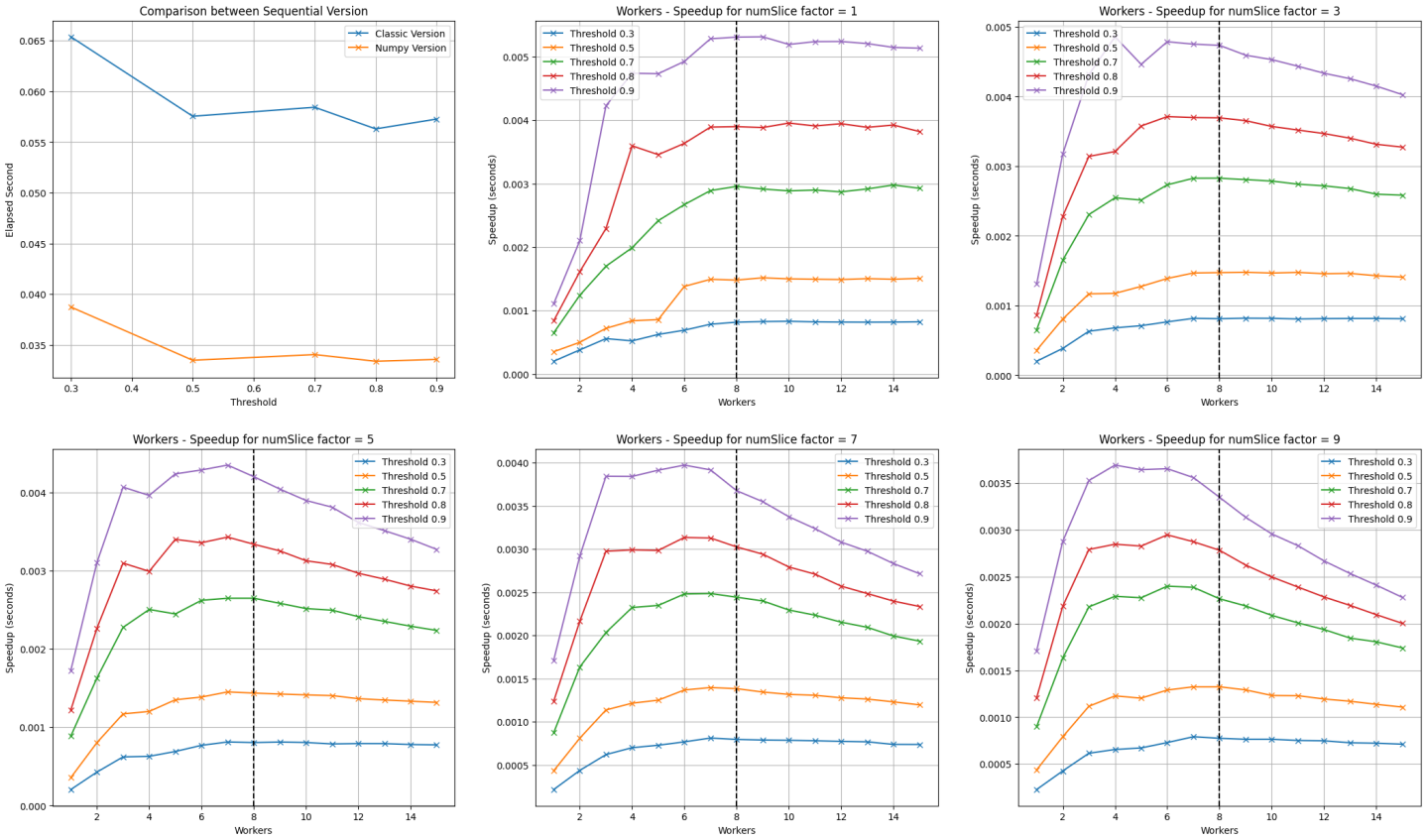
How to Run the Application

After having downloaded the package from the GitHub repository, we suggest to create a specific *virtual environment* and then install all the needed packages listed in the README.md. Now you can run *python main.py* in the *app* folder.

Benchmarks & Results

First of all the following results were obtained from a sample of 750 documents from the nfcorpus dataset, we decided to use this amount of documents since there was a good balance between computation times and number of documents.

The first benchmark was the comparison between the two sequential versions. We applied this additional benchmark in order to use the faster sequential algorithm to perform the speedups between this one and all the results from the PySpark implementation. Both for the sequential and in the PySpark implementation we applied the same sampled set for the following set of **thresholds** $[0.3, 0.5, 0.7, 0.8, 0.9]$, in addition the PySpark version was run multiple time for each thresholds with the sequence of **workers** $[1, ..., 15]$ and **numSlice Factors** $[1, 3, 5, 7, 9]$. We run multiple times the PySpark version in order to investigate how the speed up change in terms of *thresholds* and *numSlice Factors*.



All these results were obtained from a HP EliteDesk 800 G1 TWR supplied with 8 cores i7-4790 CPU and 16 Gigabytes of RAM. In the upper left graph we can see the comparison between the two sequential versions and as expected the NumPy version beat the classic version, that is why we apply its elapsed time to compute the speedups w.r.t. all PySparck runs. In the other 5 graphs, each corresponding to a specific value of numSlice factor, we can analyse the trend of the speedup in seconds with respect to the number of workers for the five threshold values.

It is easy to see that the achieved speedups in all 5 graphs are not great since there is the overhead of PySpark management that takes a high amount of computation. In fact ideally these experiments could be much significant considering an actual cluster much more performant than a standalone machine. Continuing as expected the speed up for each threshold value in each graph is higher if we consider higher values. This happens since doing so we also increment the terms to ignore during the computation of the prefix filtering and the similarity between two pairs must be higher to be added in the list of similar pairs. More interesting looking at the last graph of the first row and the others in the second row we can see a decreasing trend in the speedup, which is more noticeable for the thresholds $[0.7, 0.8, 0.9]$. Indeed if we increment the numSlice factor, we increment the number of partitions in the order of the deployed workers. This might improve speedup as it does not, it depends on the available hardware and on the dataset dimension. In this experiment we can see that the best speedup performance was obtained using the numSlice factor equal to one, or in other word with the number of partitions directly proportional to the number of workers. Even if we consider a factor of 3 we can see a decrease in performance on threshold 0.9 with already 8 workers, this trend is much more evident with factor of 9 in which the speedup of threshold 0.9 starts decreasing already with 5 workers. Finally, considering a larger set of documents could benefit from the number of partitions, this is an interesting development that could be taken into account.

Dataset Name	Execution Time NumPy Version (sec)	Threshold	Similar Pairs	Equal PySpark Similar Pairs
nfcorpus	0.03873419761657715	0.3	618	True
nfcorpus	0.03347897529602051	0.5	33	True
nfcorpus	0.03403663635253906	0.7	1	True
nfcorpus	0.033380746841430664	0.8	0	True
nfcorpus	0.033562660217285156	0.9	0	True