Documents All Pairs Similarity

Michele Lotto 875922

1 Implementation

This project was executed on "Intel i7-4790 (8) @ 4.000GHz" utilizing "Python 3.10.6". All additional libraries used can be found in the "requirements.txt" file, included with this project.

1.1 Pre-processing

It was chosen to have a common initial documents representation for both the sequential and spark algorithms to ensure the correctness of the test. The data pre-processing is done by the function:

```
def data_preparation(dataset: str, limit:int=None) -> Tuple[List,csr_matrix]
```

Given a data-set name, this function downloads the data-set from beir, cleans it, limits the number of documents and vectorizes each document. The return type is a tuple of document ids and the document matrix in Compressed Sparse Row (CSR) format; this format allows very fast dot product and can be easily converted into a classical "numpy" array if needed. The vectorization phase is done using "`TfidfVectorizer()`" from "sklearn": given a corpus this function returns the TF-IDF matrix in CSR format, where each row represent a document TF-IDF vector of terms. The "`TfidfVectorizer()`" function anso ensures that every document is normalized by L^2 norm and thus the cosine similarity between two vectors is their dot product.

1.2 Sequential Algorithm

It was chosen to optimize the naive algorithm simply using the "numpy" library. This is done by the function:

```
def sequential( doc_matrix: csr_matrix, keys:list, threshold:float )
```

Given a csr documents matrix, this function:

1. Computes the dot product of the matrix and the transposed matrix (cosine similarities) and converts the result matrix in a "numpy" array.
2. Fills its diagonal with "-1": this excludes similarities of the same document.
3. Takes only the similarities >= threshold.
4. Maps the matrix row indexes with the correct document id.
5. Returns a list of `((doc_id1,doc_id2),similarity)` pairs along with execution time.

1.3 Spark Algorithm

The Spark Algorithm is done by the function:

```
def spark_( doc_matrix:np.array, keys:list, threshold:float, n_workers:int=8, n_slices:int=1 )
```

It was chosen to allow the use of "Spark slices" to test their effects on the execution time. They are the number of parallel data divisions for the RDD. The default number is "1": a single slice per worker. Furthermore, it was chosen to set each worker to have a single CPU. Given a "numpy" matrix, this function:

1. Sorts the document matrix by maximal term frequency in the entire corpus. This is done sequentially and not included in the spark execution time calculation for the fairness of the test.
2. Zips each document id with its vectorial representation and creates "`vectorized_docs_rdd`". This RDD is cached for performance reasons: it is required multiple times in the execution.

3. Broadcasts variables to be used by spark. This ensures that each spark worker has a copy of these variables. In particular it is broadcasted: the threshold, "d*" (the "maximum document"), the vectorized documents and the non-zero term ids for each document. The "maximum document" is calculated by Spark by applying "`values().reduce(compute_d_star)`" RDD methods to "`vectorized_docs_rdd`". The non-zero terms ids are calculated by applying "`mapValues(non_zero_terms).collect()`" RDD methods to "`vectorized_docs_rdd`".

4. Applies the "MAP" function to "`vectorized_docs_rdd`" by using the "`flatMap(MAP)`" RDD method. The "`MAP()`" function calculates "on the fly" $b(d)$ (the largest term such that a document similar to d must share with d at least one term $t > b(d)$) and it appends each term $t > b(d)$ to the actual document id.

5. Groups by key by applying the "`groupByKey()`" RDD method to the "MAP" resultant RDD. This creates a list of pairs where the first value is the term id and the second value is a list of document ids.

6. Applies the "REDUCE" function to the "groupByKey" resultant RDD by using:
   "`flatMap(filter_pairs).map(compute_similarity).filter(similar_doc)`" RDD methods.

   The "`filter_pairs()`" function generates `(doc_id, doc_id)` pairs on which the similarity will be computed by iterating over each combination of doc ids for the actual term. If the maximum of the intersection of the two document terms is equal to the actual term then it appends the actual doc id pair. The "`compute_similarity()`" function computes the similarity for the actual doc ids pair. The "`similar_doc()`" function returns "`True`" if the actual similarity is >= threshold else "`False`".

7. Returns the result of "`collect()`" RDD method applied to the "RESULT" resultant RDD along with execution time.

2 Testing and Results

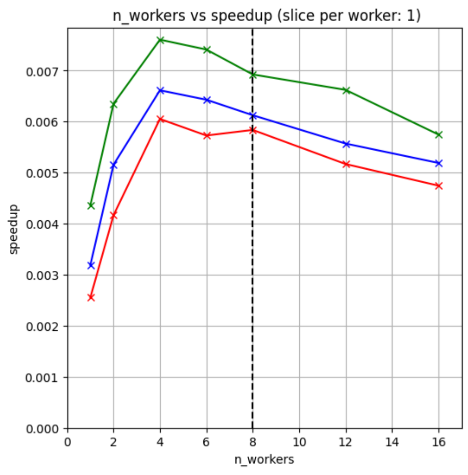
Both Sequential and Spark algorithms have the same return type: (`Tuple[List[Tuple[tuple,float]],float]`) for test commodities. The test is done by the functions:

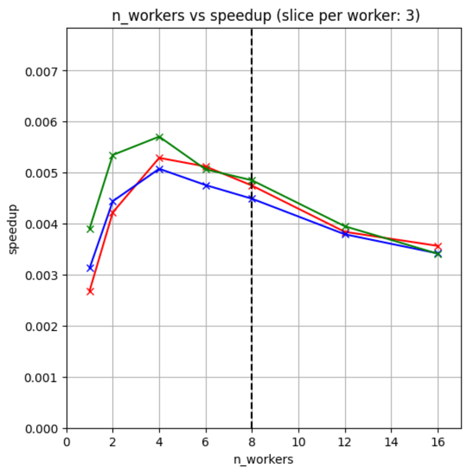
```
def comparison( keys:list, doc_matrix:csr_matrix, threshold:float,
                n_workers:int, n_slices:int) -> Tuple[float,float,set]
def test() -> None
```

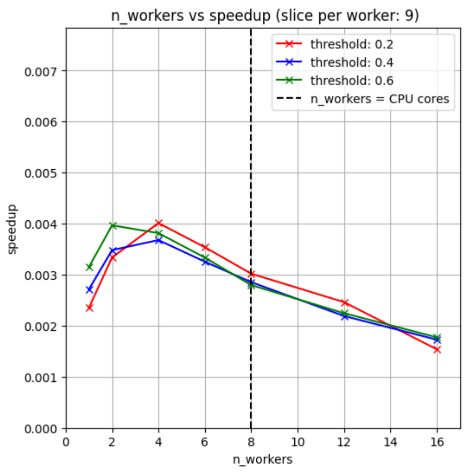
The first function calls both "`spark_(...)`" and "`sequential(...)`" functions and prints the results and the execution times. Furthermore it calculates the missing document pairs from the Spark Algorithm. It then returns the spark execution time, the sequential execution time and the set of missing document pairs. The second function calls the first one by changing the test configuration at each iteration. This function spawns a new Python process to ensure that each spark execution is independent for the previous one. The results for the test are then saved in a "pandas dataframe" and saved to file in "data.parquet". The test is executed using the first 1000 document of the "beir" data-set "nfcorpus" with thresholds=[0.2, 0.4, 0.6], workers=[1, 2, 4, 6, 8, 12, 16] and slices=[1, 3, 9]. Follows a table for threshold=0.2:

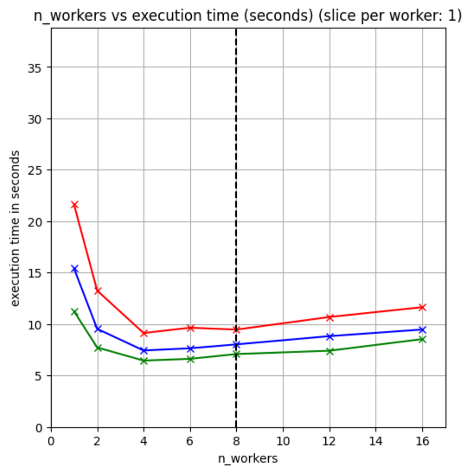
	threshold	n_workers	spark_time	seq_time	missing
0	0.2	1	21.631669s	0.051736s	[]
3	0.2	2	13.244916s	0.051758s	[]
6	0.2	4	9.117023s	0.054447s	[]
9	0.2	6	9.633517s	0.052234s	[]
12	0.2	8	9.452828s	0.068206s	[]
15	0.2	12	10.674502s	0.054711s	[]
18	0.2	16	11.627685s	0.053046s	[]

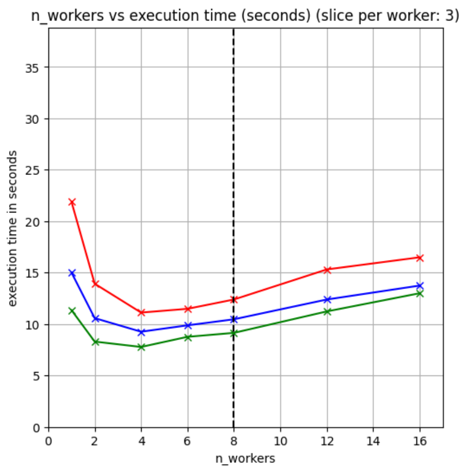
As it can be seen from the table spark execution time is extremely high compared to the sequential one: this is because it needs to copy data back and forth to the Java and Scala back-ends. Furthermore, there are not missing document pairs, thus the Spark Algorithm is correct. Follows "number of worker vs speedup" and "number of worker vs execution time" plots:

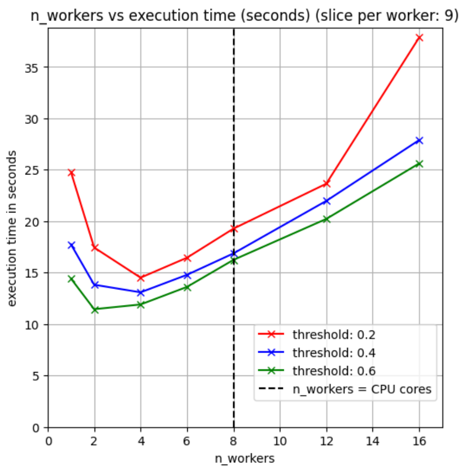












As it can be seen from the plots: augmenting the number of workers corresponds to an increasing in the speedup and a decreasing in the execution time. It is interesting to notice that the maximum speedup and the minimum execution time are reached with a number of worker different from the real number of CPU cores; this is because Spark spawns different process that do not allow all the worker processes to be parallel. Furthermore, an higher threshold results in an increasing in the speedup and a decreasing in the execution time; this is because the "MAP" function excludes a much higher number of documents. Moreover, an higher number of slices per worker corresponds to a decreasing in the speedup and an increasing in the execution time; this is because more data need to be scheduled to the Spark workers and that results in an higher overhead.