Assignment 3: Document All Pair Similarity
Runjie Xia 879779

Problem:
Documents all pair similarity is the process of finding pairs of documents within a set that have a similarity exceeding a given threshold. The objective of this assignment is to discuss the performance of both the sequential and Spark versions, which use the MapReduce pattern, in solving this problem.

Hardware components
The test was executed on a MacBook Pro 2017 with the following specifications:
- Processor: 2,3 GHz Dual-Core Intel Core i5
- RAM: 8 GB
- SSD: 256 GB
- macOS Monterey 12.0.1

Python 3.10.10 was used, and all additional libraries can be found in the README.md file.

Dataset
The dataset used to obtain the test results is the trec-covid dataset from the BEIR Repository.
Due to the size of the dataset, 3 samples of 1000 documents were sampled for the test:
- 1000 longest documents
- 1000 shortest documents
- 1000 random documents

After downloading the dataset, it is sorted by document length. The three smaller samples are then sampled from the dataset.

Pre-processing

The samples are preprocessed and saved in a .pkl file in order to be used later without recomputing the preprocessing at each run.
The cleaning process consists of concatenating the title and text fields of each document for a compact form, tokenization, lowercase conversion, stopword removal, punctuation removal, and lemmatization (using NLTK). Finally, the documents are vectorized using the TfidfVectorizer() function and sorted by IDF weights and document length, returning the TF-IDF sparse matrix of the documents (in the csr_matrix format, which allows for very fast dot product calculations and can be easily converted into a numpy array if needed).
The TfidfVectorizer() function also ensures that every document is normalized by L2 norm, so the cosine similarity between two vectors is equal to their dot product.
The preprocessing phase is the same for both the sequential and parallel versions.

Sequential Algorithm:

The implementation of the sequential version of the algorithm, proposed in the assignment, closely follows the provided pseudo code. It was implemented using the cosine_similarity() function from the scikit-learn library that computes the pairwise cosine similarities between all documents in the given TF-IDF matrix.
To optimize the algorithm's performance, the function iterates through all possible unique pairs of documents using the combinations() function from the itertools library. This approach reduces the number of iterations and avoids redundant calculations, as the similarity matrix is symmetric.
The implemented function takes the TF-IDF matrix and a threshold for document similarity as input. It returns a tuple containing a list of similar document pairs and a dictionary of metadata information. The function's purpose is to compute all pairs document similarity with a similarity above the given threshold using cosine similarity.
Additionally, a heuristic version was added to the function. This version skips the similarity calculation for pairs of documents whose length difference is too large (set to more than half of the longer document). This heuristic should improve the algorithm's performance by reducing the number of unnecessary similarity calculations.

Parallelized Algorithm: Spark and MapReduce

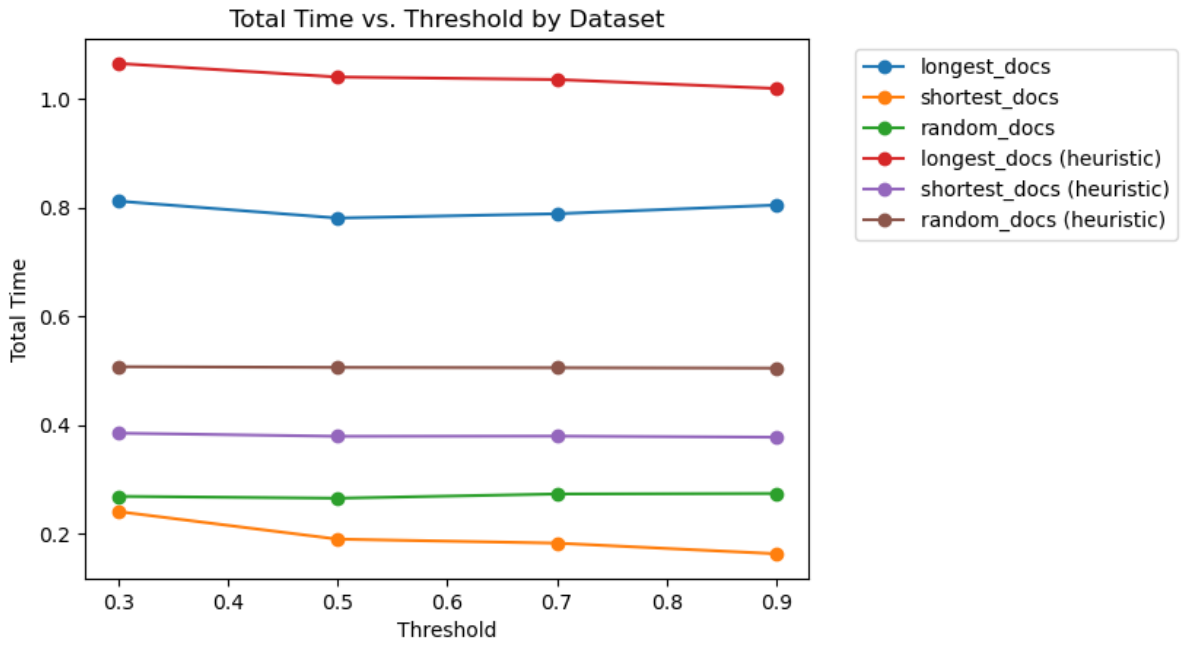
The parallelized version of the algorithm is an implementation of the MapReduce programming paradigm with prefix-filtering discussed in class. The implementation uses PySpark, with a single worker, 8 GiB of available memory, and 4 cores (spark session created with default values). The input to the algorithm is a sparse CSR matrix of TF-IDF values, a threshold, the number of workers, and the number of slices to partition the RDD.
The first step of the algorithm is to sort the terms in the TF-IDF matrix in decreasing order by document frequency. Then, the algorithm creates an RDD (Resilient Distributed Dataset) using PySpark's parallelize() function, zipping the document IDs and the corresponding TF-IDF vectors and setting the number of partitions based on the number of slices and workers.
Next, it computes the maximum document d* (“d_star”). A CSR matrix that contains the maximum score value of each term in any document (i-th entry is the maximum value of i-th column (term)).
After that, the algorithm computes the dictionary of boundaries b(d), where the keys are doc_ids and the values are the indices of the largest terms in each document immediately before the threshold is reached. Both the threshold and b(d) are broadcasted to efficiently share them across the workers.
The algorithm then applies a series of transformations on the RDD: mapping, grouping, reduction, and deduplication operations.
Additionally, the algorithm includes a heuristic version that skips the similarity calculation for pairs of documents whose length difference is too large, similar to the sequential implementation.

Experimental evaluation

The algorithms were executed for all combinations of the following settings on the samples mentioned before:
thresholds = [0.3, 0.5, 0.7, 0.9], number of workers = [4, 7, 9, 12], number of slices = [4, 7, 10].

The plots show the execution times of each algorithm as the number of executors varies.
The Spark-based algorithm is correct since the number of document pairs coincides with that of the sequential algorithm. However, there is some overhead involved in transferring the data from the local machine to the Spark cluster, which can lead to slower performance compared to the sequential algorithm. Furthermore, the resources allocated to the PySpark session can also affect the performance and computation times.
In PySpark, increasing the number of workers can increase parallelism and potentially speed up the computation, but it also requires more resources and may lead to resource contention and slower performance if the resources are limited. The optimal number of workers depends on the available resources and the size of the data. This would explain why, as the number of workers increase, the execution times get slower.
Looking at the results of the sequential algorithm, since all the samples have the same size, it is clear that the dimension of each document matters. Having longer documents means more data to process. For the Spark-based algorithm, the execution times are lower for higher values of threshold, as expected, since this reduces the number of document pairs to consider during the computation of the similarity thanks to the prefix filtering.
The heuristic versions of the algorithms correctly gave the same number of pairs as their normal counterpart.
The threshold didn't really influence the times of the sequential algorithm, since we compute the cosine similarity directly with the sparse TF-IDF matrix, and for this same reason the sequential algorithm with the heuristic tends to be slightly slower since we just added some operations.
While for the Spark-based algorithm, the heuristic version is always slightly better than the normal version. However, the difference in execution times might be small because the difference in length allowed by the heuristic is too lax. Setting it stricter might filter out more pairs, but also lead to a loss of pairs.
Finally, it is worth noting that the resources used in this experiment may not be sufficient for larger datasets. Conducting a separate experiment with a larger dataset and more resources would provide more insights into the performance and scalability of the algorithms.

Sequential



MapReduce Spark