MapReduce All Pairs Similarity

Learning with Massive Data - Tonetto Davide - 884585

1 Introduction

This project aims to implement a multi-threaded version of the All Pairs Similarity algorithm using the MapReduce paradigm. The algorithm is used to compute the similarity between all pairs of documents in a given dataset. The similarity is computed using an approximation of the cosine similarity. The algorithm is implemented in Python using the **PySpark** library.

The dataset used in this project contains a list of mail messages. The dataset is composed of 231533 messages and is stored in a single parquet file that can be downloaded using the following command: `wget https://huggingface.co/datasets/jacquelinehe/enron-emails/resolve/main/data/train-00000-of-00004.parquet?download=true`.

The project is developed using Python 3.11 and PySpark 3.5.1. The results are obtained by running the code on a machine with an Intel Xeon silver 4208 2.1ghz 8 core and 16 threads CPU and 64GB of RAM. The code is executed in a standalone cluster mode with a variable number of cores and workers for each test.

2 Spark Implementation

The algorithm is implemented using the MapReduce paradigm. The algorithm can be summarized in four main steps: 1) Tokenization of the documents and computation of the term frequency - inverse document frequency (TF-IDF) of each word in the dataset. 2) Computation of the simHash of length m of each document. 3) Division of the simHashes into $\frac{m}{p}$ blocks and grouping of the documents that share at least one block of the simHash. This is done to reduce the number of comparisons between the documents. 4) Computation of the similarity between all pairs of documents in each group.

The **first step** is implemented as follows. The Tokenization of the documents is performed firstly by removing all the special characters and numbers from the mails and then using the `Tokenizer` class of the `pyspark.ml.feature` module of PySpark. Then the `StopWordsRemover` class is used to remove the stop words from the list of words. The TF-IDF is computed using the `CountVectorizer` and `IDF` classes of the `pyspark.ml.feature` module. `CountVectorizer` is used to compute the term frequency of each word in the dataset, while `IDF` is used to compute the inverse document frequency.

In the **second step**, the simHash of each document is computed by multiplying the TF-IDF of each word in the document by a random vector of length m of 1 and -1 and then taking 1 if the result of the summation of each resulting vector is greater than 0 and 0 otherwise. Each word in the dataset is associated with a random vector of length m stored in a matrix of size $m \times n$, where n is the number of unique words in the dataset.

In the **third step**, the simHashes are divided into $\frac{m}{p}$ blocks and the documents that share at least one block of the simHash in the same position are grouped. The blocks are computed by dividing the simHashes into $\frac{m}{p}$ blocks of length p and then encoded as integers. The documents are grouped by the blocks of the simHashes using the `groupBy` method of the PySpark DataFrame.

In the **fourth step**, the similarity between all pairs of documents in each group is computed. The similarity is computed using an approximation of the cosine similarity using the following formula:

$$sim(d_i, d_j) = 1 - \frac{hamming\_distance(s_i, s_j)}{m}$$

where $sim(d_i, d_j)$ is the similarity between the documents d_i and d_j, $hamming\_distance(s_i, s_j)$ is the hamming distance between the simHashes s_i and s_j of the documents d_i and d_j and m is the length of the simHashes. Then the pairs with a similarity greater than a given threshold (0.9) are collected and stored in a list.

3 Results

In the following table, the execution time of the algorithm is reported for different values of m and p. The execution time is computed by running the algorithm with a total of 16 cores. The execution time is reported in seconds.

m	p	simHash exe time	grouping exe time	similarity exe time	total exe time
64	16	116.4	21.2	55.1	309.2
64	32	118.6	16.2	42.0	176.8
128	16	136.8	49.2	141.0	327.0
128	32	125.2	27.1	65.2	217.5
256	16	152.5	89.3	576.2	818.0
256	32	143.6	81.2	252.5	477.3

The results show that the execution time of the algorithm increases with the length of the simHashes and the number of blocks.

Speedup

In the following image, the speedup of the algorithm is reported for different values of m and p. The speedup is computed as the ratio between the execution time of the algorithm with 1 core and the execution time of the algorithm with x cores for $x \in \{4, 8, 12, 16\}$.

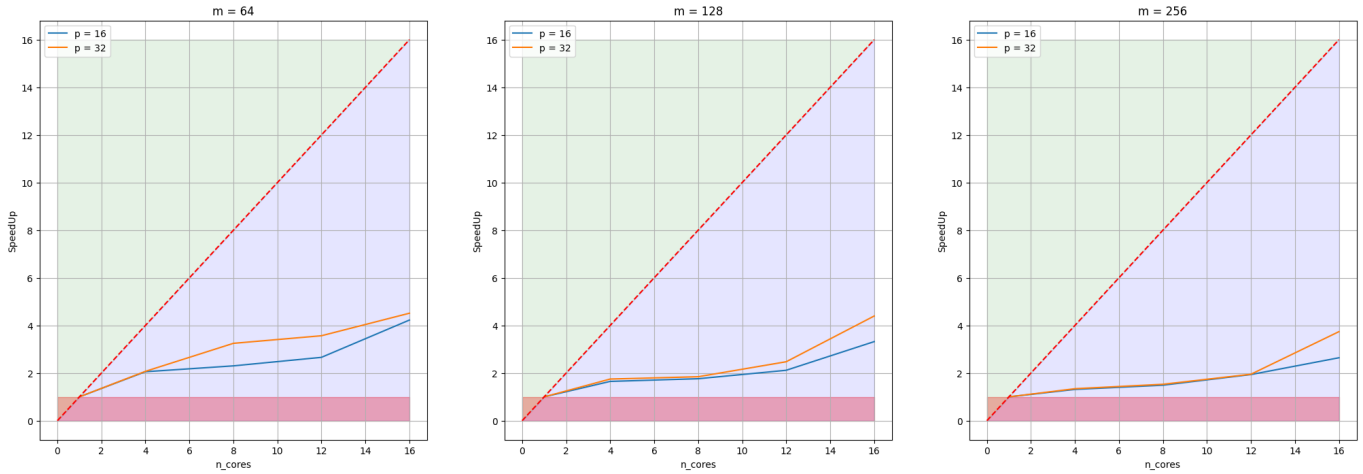

Figure 1: Speedup of the algorithm for different values of m and p.

% of comparisons

The following image shows the % of the number of comparisons between the documents for different values of m and p. The % is computed as the ratio between the number of comparisons between the documents after the grouping and the total number of pairs of documents in the dataset. It's possible to see that the number of comparisons decreases with the number of blocks, this explains the decrease in the execution time of the algorithm with the increase of p seen in the previous table.

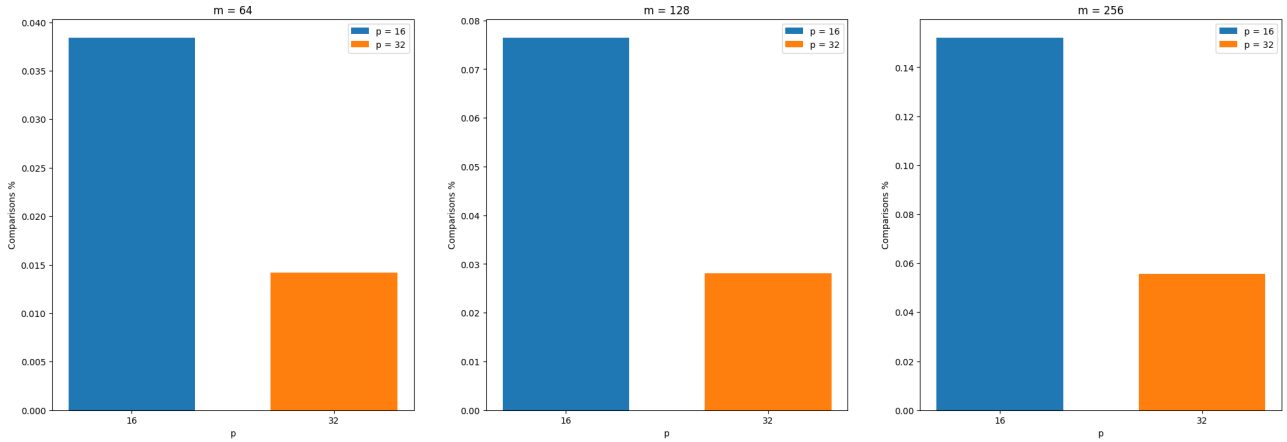

Figure 2: % of number of comparisons between the documents for different values of m and p.

Number of similar pairs

The following image shows the average number of similar pairs for different values of m and p. As expected, the number of similar pairs is not always the same since the algorithm is an approximation of the cosine similarity meaning that we can have false positives and false negatives. It's interesting to notice that the number of similar pairs decreases with the increase of the number of blocks, this is because the number of comparisons between the documents decreases and so does the number of false positives and false negatives.

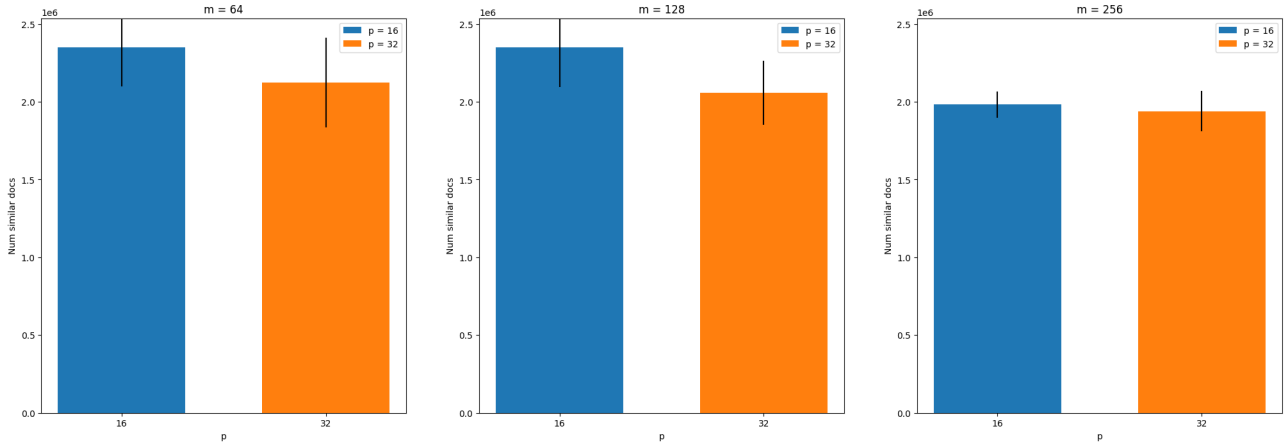

Figure 3: Number of similar pairs for different values of m and p.