Data Mining 21/22 Homework 2

Marco Calamo

November 2021

All work is tracked on Github at `https://github.com/IlKaiser/DM22_HW`

1 Exercise 1 - Search Engine

In order to implement a Cosine Similarity-based search engine, I did the following steps:

1. Using *ntlk*, I pre-processed, tokenized (using white space as separator) and normalized the text descriptions and the titles of the jobs by:
   - generating the tokens
   - converting all tokens to lowercase
   - removing punctuation
   - filtering out Italian stop words (since the datset is in Italian)
   - stemming the tokens
2. I created a simple ***inverted index*** as a python dictionary, where every pre-pocessed token is the key and the value is a list of document where the token occurs with the number of total occurrences in the corpus
3. Using the inverted index as a basis, I created the **TF**,**IDF** and **TFIDF** indexes as python dictionaries:
   TF has a token per key and a dictionary (where document index is the key) with $\frac{tokensPerDoc}{|allTokens|}$ as a value
   IDF has a token as key and $\frac{|N|}{tokenFreq}$ as value
   TFIDF has a token as key and a dictionary (where document index) with tf-idf product per document as value
4. I saved all computed indexes to files in JSON format as requested
5. I computed Cosine Similarity by pre-processing with the same steps described before, and by comparing the resultant tf-idf query vector with all tf-idf vectors in dataset
6. I printed on screen the top 10 most matching result
7. I used the ***Tensorflow*** library with *tf-idf vectorizer* as a benchmark

1.1 Results

Figure 1: "Programmatore Java" as query

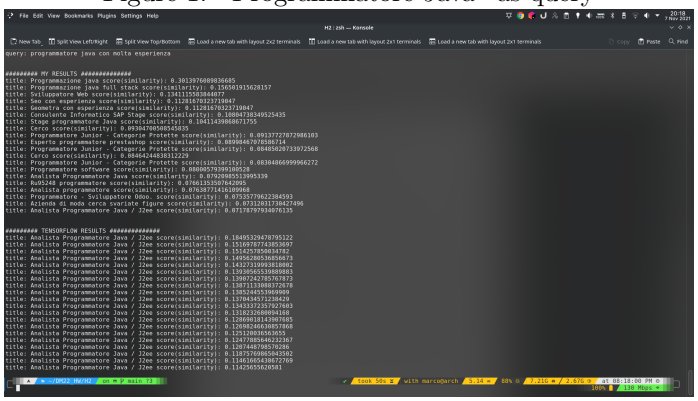
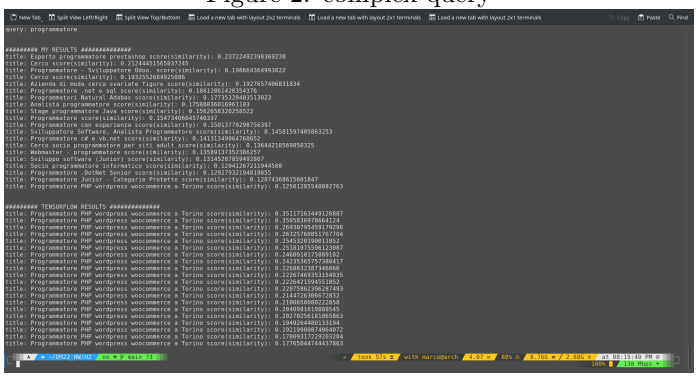


Figure 2: complex query



As we can see my implementations filters out duplicates and does even a better job when it is required to match an entire description.

For more details see *search.py*

Figure 3: whole job as a query

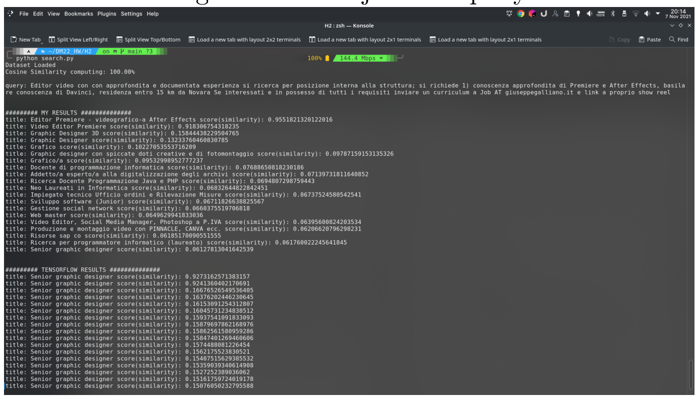
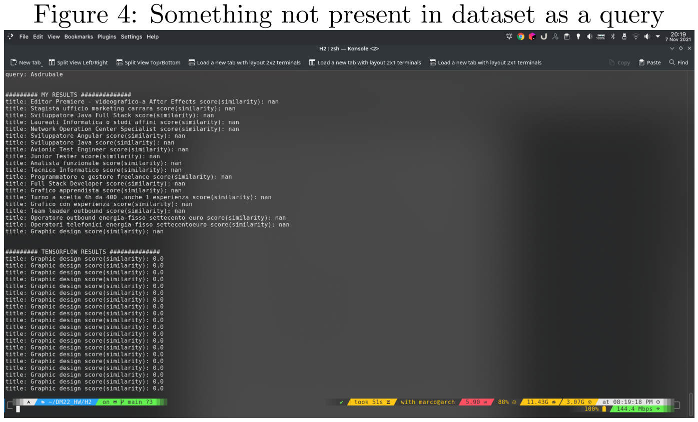


Figure 4: Something not present in dataset as a query



2 Exercise 2 - Locality Sensitive Hash

2.1 Shingles

In order to create k-shingles i used a simple straightforward function in python that takes the text to shingle and k as parameters.
The function iterates over the string and outputs a k-shingles list.
An option to create 4-bytes hash signatures to save space is present as a standalone function.

2.2 Min Hashing

Minimum Hash signature matrix is computed for every document (job announcement) in the corpus, by applying the desired number of hash functions generated with the provided *hashFamily* utility and computing the minimum hash values per shingle among every function.

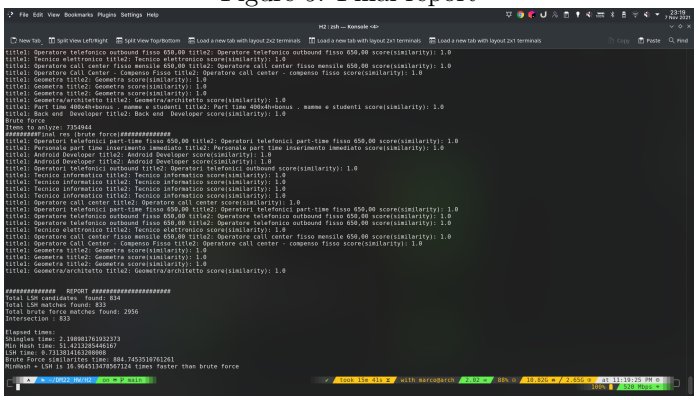
2.3 LSH

The LSH is computed with r (rows per band) and b (number of bands) as parameters on the signature matrix. On the LSH candidate pair –different documents that were a hit at some point in LSH algorithm – is computed the Jaccard Similarity on Shingles.
Finally the program outputs the top 20 most similar documents.

2.4 Results

The resulting program does a good job in duplicates and near-duplicate detection on dataset. Using 10 as r for LSH (and $\frac{|dataset|}{r}$ as b) i obtained the best results, with a good speed to correct match ratio. My program resulted being about 20x faster than brute force similarity computing, while finding about a third of all correct similarities.

Figure 5: Final report



For more details see *lsh.py*

3 Exercise 3 - Locality Sensitive Hash on Spark

The challenge of this exercise was in porting the code I wrote for the previous exercise to Spark platform using ***pyspark*** library.
The steps are the same form 2, but every time i needed to to alter the initial dataframe I used the map function: for shingles, min-hash signatures and LSH.

3.1 Results

As expected –despite it is not accurate the way of measuring Spark execution time on python– the final results are the same compared to 2, with Spark that outperforms simple python on computing shingles, min-hash signatures and LSH candidates.
Spark, however, despite having the dataset cached for faster memory access, is slower on outputting the final results on screen since, for the way my program works, it is necessary to do many subsequent access (that result quite slow) on the dataset in order to retrieve all fields corresponding to the similar jobs.
Finally Spark was even slower than simple python in computing brute force similarity, done by joining two sets of shingles with different IDs (by avoiding to create duplicate rows) and mapping the shingles to their Jaccard Similarity. The computation took more than 20 mins, and thus its execution is discouraged; the code is available at the bottom of the file but it is commented out for fast execution of the program.

For more details see *spark_lsh.py*

4 Dataset

The file used as a dataset for benchmarking the results of my work is ***jobs.txt*** (included in the zip file), generated by *crawler.py* from previous homework in date 02/11/2021, and has the full job description available on
`https://www.kijiji.it/offerte-di-lavoro/offerta/informatica-e-web/`