Studying Sparse & Dense Retrieval

Riccardo Zuliani 875532

April 2023

1 Introduction

This assignment has the main purpose to analyse and study the consequences of the execution of the **Maximum Inner Product Search (MIPS)** over *dense* and *sparse* vectors embedding, in the case where each vector has both representation.

Mathematically speaking we have a query vector $q \in \mathbb{R}^n$ and a set of document vectors $\mathcal{D} \subset \mathbb{R}^n$ and we want to find the top k documents with the maximal inner product with q, such that:

$$S = \underset{d \in \mathcal{D}}{\arg\overset{(k)}{\max}} < q, d >$$

So, if we have documents and queries in the following form $u \bigoplus v$, where $\bigoplus$ denotes the concatenation, $u \in \mathbf{R}^n$ is a *dense vector*, and $v \in \mathbf{R}^n$ is a *sparse vector*, how do we retrieve the top-k documents with respect to a given query by maximal inner product?

The suggested path is to take advantage of the linearity of the **inner product operator**. The inner product operator of a query vector $q = q_{dense} \bigoplus q_{sparse}$ with a document vector $d = d_{dense} \bigoplus d_{sparse}$ is the sum of $q_{dense} \cdot d_{dense} + q_{sparse} \cdot d_{sparse}$.

We can now approximate the joint **MIPS** by breaking it up into two smaller MIPS problems:

- Retrieve the top-k' documents from a **dense** retrieval system defined over the **dense** portion of the vectors.

- Retrieve the top-k' documents from a **sparse** retrieval system defined over the **sparse** portion of the vectors.

Before *merging* the two sets and retrieving the top-k documents from the combined (much smaller) set. As k' approaches infinity we should see, the final top-k set becoming exact, but retrieval becomes slower.

2 Application Workflow & Keys Choices

First of all as code editor and environment we decided to use a jupyter notebook in Google Colaboratory Pro equiped with Python 3.10.11. The reason of this is that we already had a subscription to Colaboritory since we had also an other on going project that need a GPU.

Now as first choice we decided to use the suggested **Beir** since it offers a vast set of available dataset, but more interesting it already implements the Sentence-BERT and a retrieval procedure for the Elastic Search implementation of the BM25. Regarding the datsets to examine and study we decided to opt with **scifact** and **nfcorpus** since they are light weight and thus by far are easy to debug and goes through documents and queries. As you can imagine the early step of our application was actually the downloading of the queries - documents of our datasets. We retreived queries and documents for both datasets in the from of Python dictionary with as *key* the id of documents or queries and as value the text and title for the documents and only the text for the queries. Next we decide to apply to them some pre-processing that includes tokenization (so we split the query or documents text into sequence or words), lemmatization (process of reducing an inflected form of a word to its canonical form), stop word removing (remove the common word from the english language) and punctuations removing. As package to do these cleaning actions we use both NLTK (Natural Language Tool Kit) and spaCy, but after some comparison we decide to stay with *spaCy* since it is more recent and many expert suggest it rather than NLTK. To speed up the testing of the application we develop also a separate Python script that individually perform the text pre-processing saving also the modified text as a *.parquet* such that we can easily upload both files in google drive and then downloading and reading it in Colaboritory. Going back to the technicality of the pre-processing, we would like to note that this cleaning steps are applied only to the documents and queries that will feed into the BM25 sparse embedding, since in case we apply this pre-processing steps also to the dense embedding, SentenceBert will loose some contextual information from this modifications. As previously mentioned we had to take familiar with Elastic Search in order to use the Beir BM25. Elastic Search is nothing else than a distributed, free and open search and analytics engine for all types of data, including textual, numerical, geospatial, structured, and unstructured [1]. At this point we decide to see what a sparse embedding look like and thus test the BM25 over the previous two datasets and also using other bigger datasets. Indeed in order to do so we create a *EvaluateRetrieval* object setting the score function as *dot* and the number of value to evaluate as the dimension of the documents set. Then we call the *retrieve* methods passing the queries documents set pair obtaining the sparse embedding. However we note that the Beir retreive implementation returns an *AssertError* that lead the whole application to stop in case the model wants to return for a given query a num-

[1]https://www.elastic.co/what-is/elasticsearch

ber of documents which is higher than 10000. This behaviour had been explicitly reported to the Beir Team by the following GitHub issue. This problem motivate us to develop our self an implementation of the BM25 that easily exploit the BM25 score function for a given query document.

$$score(D,Q) = \sum_{i=1}^{n} IDF(q_i) \cdot \frac{f(q_i,D) \cdot (k_1 + 1)}{f(q_i,D) + k_1 \cdot (1 - b + b \cdot \frac{|D|}{avgdl})}$$

$$IDF(q_i) = log\left(1 + \frac{N - n(q_i) + 0.5}{n(q_i) + 0.5}\right)$$

As we can see the BM25 can be expressed as a product of two separate parts, the $IDF(q_i)$ and the $TF_{BM25}(q_i, d)$. This were the two step for coding the implemented version. Indeed first we feed the whole set of documents in our implemented version, doing so for each document we compute the relative term frequency, the document frequency of each term, the inverse document frequency and the average documents length of the set. Furthermore we evaluate the set of queries using the previous calculated metrics in parallel. The parallelization step take action by assign to each available thread the evaluation of all documents of a given query. This is done by using the *ThreadPoolExecutor* from the *concurrent.features* pakage.

Furthermore we step into the implementation of the SentenceBert for the dense embedding of our queries documents. We use the Beir implementation which take as input the SentenceTransformer type which in our case is the *all-MiniLM-L6-v2* and the batch size. This embedding takes advantage of using the GPU to speed up the mapping procedure. Furthermore as in the Elastich Serach BM25 implementation we set the EvaluateRetrieval score function as *dot* and we set the number of value to evaluate as the dimension of the documents set.

The SBERT model is based on the siamese network architecture, which takes two input sentences and produces a similarity score between them. The model consists of two identical deep neural networks that share the same set of parameters, and the output of each network is a fixed-length vector representation of the input sentence The key innovation of SBERT is the use of contrastive loss during training, which encourages the model to learn embeddings that are close together for similar sentences and far apart for dissimilar sentences. This helps to ensure that the model captures the semantic meaning of the input sentences rather than just their syntactic structure.

Regarding the strategy to obtaining the ground truth at k from the retrieved dense and sparse embedding, and the merged version of the embedding both at k' we decided to divide the problem into sub-problems by first obtain a dictionary describing the ground truth at k for each dataset and then computing the merged embedding at k from the sparse and dense embedding at k'. An other strategy could be computing at each iteration the ground truth and the merged embedding but we decided to stick with the initial strategy in order to gradually have a look on the partial results and thus being able to easily debug the application in case of errors.

At this point we reach the first key part of the project, the computation of the ground truth of query document relevance at k. More deeply given both embedding firstly we compute the ground truth scores of each query by taking the sum of the sparse and dense score of each documents. We set it to *zero* in case the specific document is not present in the sparse embedding and to *-infinity* if the document is not present in the dense embedding. This last action is done since we note that sometimes some documents were retrieved even if the context was completely different from the one of given query. Obtained the ground truth score of each query we sort it by score getting the rank results. Finally for all the set of k we create the ground truth at k by taking only the first top k documents for each query. All these steps of course are done for each documents - queries dataset.

The next key part of the application is the merging phase obtaining our actual output result. In order to do so for each query associated to a sparse and dense document embedding we sort these two vector, and for each k' we get the sum of the dense and sparse embedding score of each document. Moreover as in the previous step if a given document is not present in the sparse embedding firstly we get the value of the ground truth if present, but in case this is false we set it to *zero*, similarly in case a given document is not preset in the dense embedding first we set its score to the ground truth value and if even in this case it is no present we set its score to *-infinity* like in the ground truth computation. Again these procedure is done for all documents - queries dataset.

Finally as last step since we have all what we need we apply the evaluation procedure. Indeed given the ground truth at k and the merged embedding at k' we continue as follow. For each k and for each k' first we retreive the merged embedding at k since we want to have the ground truth ad the merged result with the same number of retreived document for each query. Then we evaluate using the *EvaluateRetrieval.evaluate* from Beir with number of evaluation document equal to k. This evaluation returns the following score: *NDCG*, *MAP*, *Recall*, *Precision*.

As analysis we plotted the four graphs for dataset each for scores, in which as x axis there are the k' tested value and as y axis the score. We have multiple line plot reflection the number of possible k that we have tested.

3 Benchmarks & Results

In this section we have a look on the results of the two examined dataset. Each set of four graph described the obtained score for the scifct or nfcorpus dataset. Starting from the top left plot we have the *NDCG*, on the right the *MAP*, on the bottom let the *Recall* and on the right the *Pecision*. Each plot has alwasys the same number of line plot describing the different ground truth at k taking for the actual evaluation.

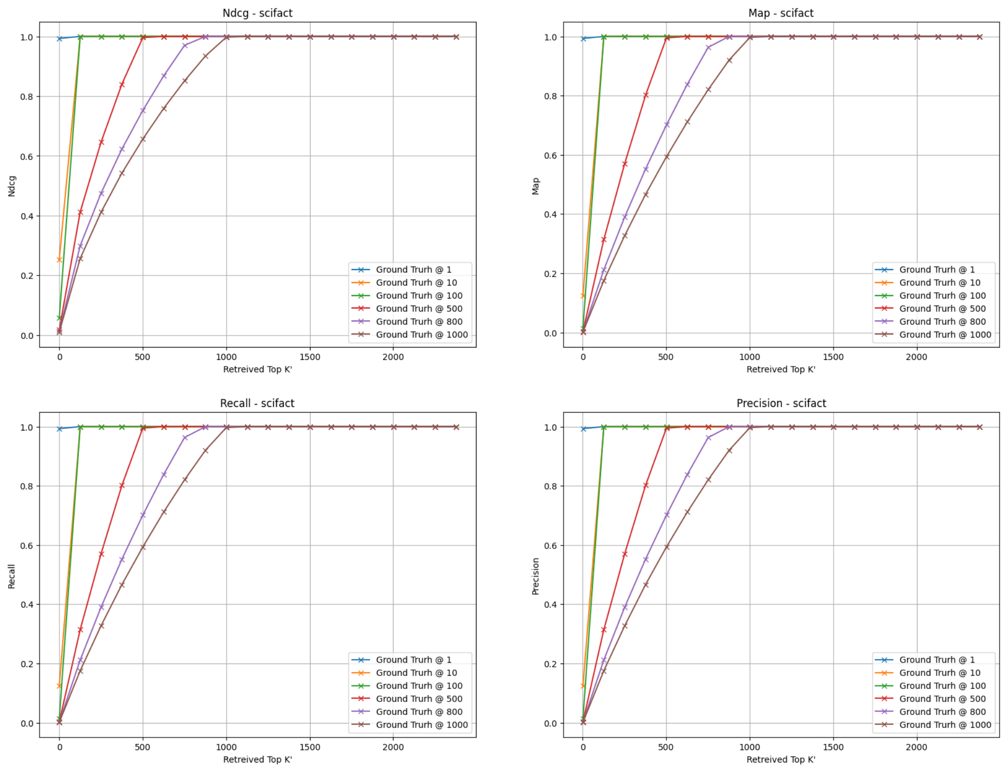

Figure 1: scifact dataset results

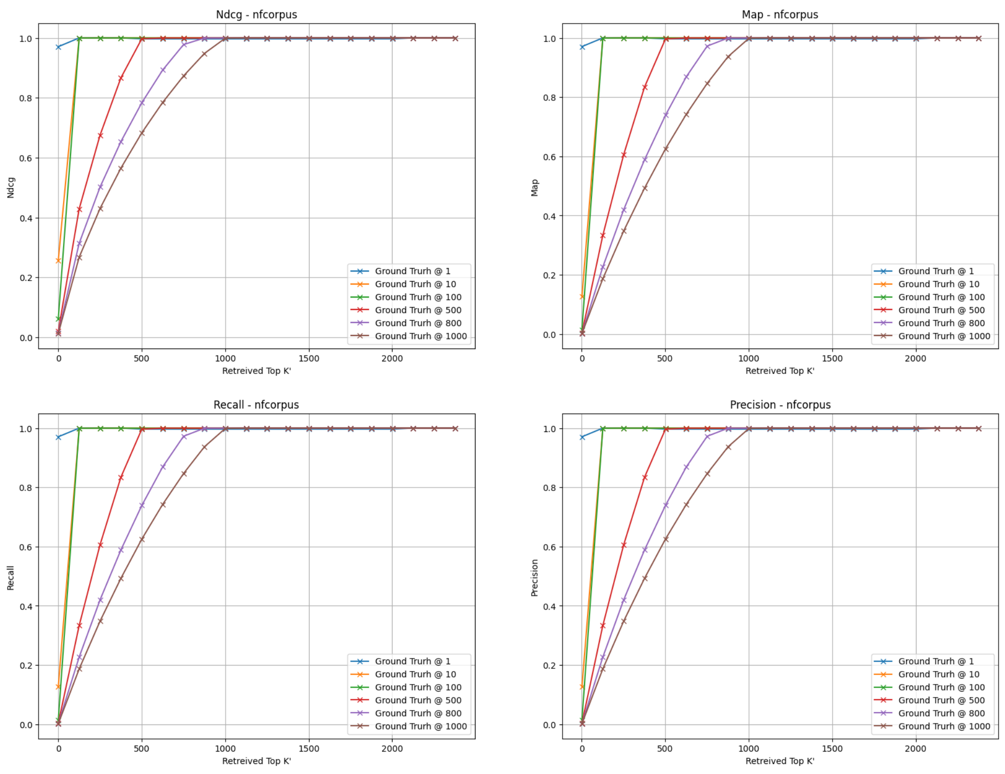

Figure 2: nfcorpus dataset results

As expected the behaviour of the relation between the *ground truth* at k and the *merged dense and sparse embedding* at k' is: the lower we choose k (like $k = 1$) the more quickly it will converge to have all the scores equal to 1 respect the choice of k'. In other word this phenomena can be described as the more k' document we retrieve in our merged embedding the more accurate our retrieval gets.

4 Conclusion

In this article we ave discussed an application that perform the Maximum Inner Product Search between the ground truth of the query document relevance at k and the merged version between the dense and sparse embedding of the documents at k', arguing the keys choices with specific motivations. Furthermore we studied this relation seeing as previously mentioned that the deeper we go with the merged k' the more accurate are our results.