Application for Studying Sparse & Dense Retrieval (Part I)

Riccardo Zuliani 875532

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.

So, if we have documents and queries in the following form $u \bigoplus v$, where $\bigoplus$ denotes the concatenation, $u \in \mathbb{R}^n$ is a *dense vector*, and $v \in \mathbb{R}^n$ 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

As code editor and environment we decided to use a jupyter notebook in Google Colaboratory Pro equipped with Python 3.10.11. The reason is that we already had a subscription to Colaboratory since we also had another ongoing project that needed the GPU.

2.1 Dataset Download & Pre-Processing

Now as first choice we decided to use the suggested **Beir** package since it offers a vast set of available dataset, but more interesting it already implements the SentenceBERT and a retrieval procedure for the ElasticSearch implementation of the BM25. Regarding the datasets to examine and study we decided to opt with **scifact** and **nfcorpus** since they are lightweight and thus by far are easy to debug and goes through documents and queries.

The first step of our application was actually the downloading of early cited datasets, so we retrieved queries and documents for both of them in the from of 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 ToolKit) 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 documents and queries text pre-processing in parallel. Saving also the modified text as a *.parquet*. We run this script in a local machine equipped with 8 cores CPU in order to get the pre-processed files more quickly. Uploading these files in a cloud storage like Google Drive we can import the pre-processed corpus and queries from there to Colaboratory. 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 lose some contextual information from this modifications.

2.2 Sparse Embedding Compuation

As previously mentioned we had to become familiar with Elastic Search in order to use the Beir BM25. ElasticSearch 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 looks like and thus test the BM25 over the previous two datasets but also using a bigger one. This is done by:

```
model = BM25(index_name=index_name, hostname=hostname, initialize=initialize)
retriever = EvaluateRetrieval(model, score_function='dot', k_values=[len(corpus)])
return retriever.retrieve(corpus, queries)
```

Indeed we first create the *BM25* object specifying the *index_name* which is equal to the dataset name, the *hostname* equal to *localhost* and *initialize* set to *True* to delete all existing indexes with same name and reindex all documents. Moreover we set the *EvaluateRetrieval* object with the score function as *dot* and the number of values to evaluate as the dimension of the document set. Finally we call the *retrieve* methods passing documents and queries sets obtaining the sparse embedding.

However we note that the Beir retrieve implementation returns an *AssertError* that leads the whole application to stop in case the model wants to return for a given query a number of documents which is higher than 10000. This behaviour had been explicitly reported to the Beir Team by the following GitHub issue. This problem motivates us to develop an implemented version of the BM25.

The implemented version exploit the BM25 formula by setting the two hyperparameters $b = 0.75$ and $k_1 = 1.25$. There for each document we compute the relative term frequency, the document frequency of each term, the inverse document frequency and the average document length of the set.

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

Furthermore we evaluate the set of queries using the previously calculated metrics in parallel, by making a parallelization step assign to each thread the task to compute the evaluation of all documents for a given query. This is done by using the *ThreadPoolExecutor* from the *concurrent.features* package.

Furthermore it can also being proved that the BM25 score function can be expressed as the dot product between two vectors like it follows:

- Let V be a vocabulary
- Let $\vec{d} \in \mathbb{R}_+^{|V|}$ be a vector whose i^{th} coordinate records the importance of the i^{th} term in V such that:

$$\forall i = \{1, ..., |V|\} : \vec{d}_{t_i} = \begin{cases} 0 & \text{if } t_i \notin d \\ \frac{TF(t_i,d)}{IDF(t_i)} & \text{if } t_i \in d \end{cases}$$

- Let $\vec{q} \in \mathbb{R}_+^{|V|}$ be a vector whose i^{th} coordinate records the count of the i^{th} term of V in the query such that:

$$\forall i = \{1, ..., |V|\} : \vec{q}_{t_i} = \sum_{t \in q} C(t_i, t) \quad \text{where} \quad C(t_i, t) = \begin{cases} 0 & \text{if } t_i \neq t \\ 1 & \text{if } t_i = t \end{cases}$$

Then we can write what follows:

$$Score_{BM25}(q, d) = \sum_{q_t \in q} TF(q_t, d) \cdot IDF(q_t) = \langle \vec{q}, \vec{d} \rangle$$

2.3 Dense Embedding Compuation

Regarding the computation of the dense embedding as already said we decided to use the implementation made available by the Beir package. In particular, Beir implements the *DenseRetrievalExactSearch (DRES)* class, which provides a simple interface to use SBERT (Sentence-BERT) pretrained models. This implementation takes as parameters one of the available Sentence Transformer and the batch size to split the data. We decide to use the suggested *all-MiniLM-L6-v2* as Sentence Transformer and the default value 16 as the batch size. This embedding process takes advantage of using the GPU to speed up the mapping procedure and can be summarized by these three lines of code:

```
model = DRES(models.SentenceBERT('all-MiniLM-L6-v2'), batch_size=16)
retriever = EvaluateRetrieval(model, score_function='dot', k_values=[len(corpus)])
return retriever.retrieve(corpus, queries)
```

Like the ElastichSerach BM25 implementation first of all we create the model object which in this case is the *DRES* and the the *EvaluateRetrieval* with score function as *dot* and number of value to evaluate as the dimension of the documents set. Finally we call the *retrieve* methods passing documents and queries sets obtaining the dense embedding.

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.

2.4 Ground Truth & Merged Embedding Compuation

Regarding the strategy to obtain 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. First we obtain a dictionary describing the ground truth at k for each dataset and then we compute the merged embedding at k from the sparse and dense embedding at k'. Another 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 at the partial results and thus being able to easily debug the application in case of errors.

2.4.1 Ground Truth Computation

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 we compute the ground truth scores of each query by taking the union of the document index and then summing the sparse and dense score for each of them. We set the each document score to *zero* in case the specific document is not present in the sparse embedding and also in case the document is not present in the dense embedding. Obtained the ground truth score of each query we sort it by score getting the rank results. Finally for all sets 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 document - queries dataset.

2.4.2 Merged Embedding Computation

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 with a sparse and dense document embedding we sort these two vectors by document score, then for each k' we get the union of the documents index of the first k' documents score from both embedding. Obtained this we can now compute the merged result by summing the documents sparse and dense score from the merged union vector. Moreover, 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* like in previous computation. Similarly in case a given document is not present in the dense embedding first we set its score to the ground truth value and if even in this case it is not present we set its score to *zero* like before. Again this procedure is done for all documents - queries dataset.

2.5 Evaluation Procedure

Finally as last step we apply the evaluation procedure. Indeed for each value of k and for each value of k' we retrieve the merged embedding at k since we want to have the ground truth and the merged result with the same number of retrieved documents for each query. Then we apply the evaluation step via the following line of code.

```
ndcg, _map, recall, precision = EvaluateRetrieval.evaluate(ground_truth,
                                merged_emb_at_k, k_values=[k])
```

Using the *EvaluateRetrieval.evaluate* from Beir with number of evaluation document equal to k, we obtain the following score: *NDCG*, *MAP*, *Recall*, *Precision*.

At the end we visualize all the results by plotting a graph describing the relation over the ground truth at k and the merged embedding at k' for each score, as we can see in the next section.

For any doubt or technical insight you can check the project GitHub repository.

3 Benchmarks & Results

In this section we have a look at the results of the two examined dataset. Each set of four graphs 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 always the same number of line plots describing the different ground truth at k taking for the actual evaluation.

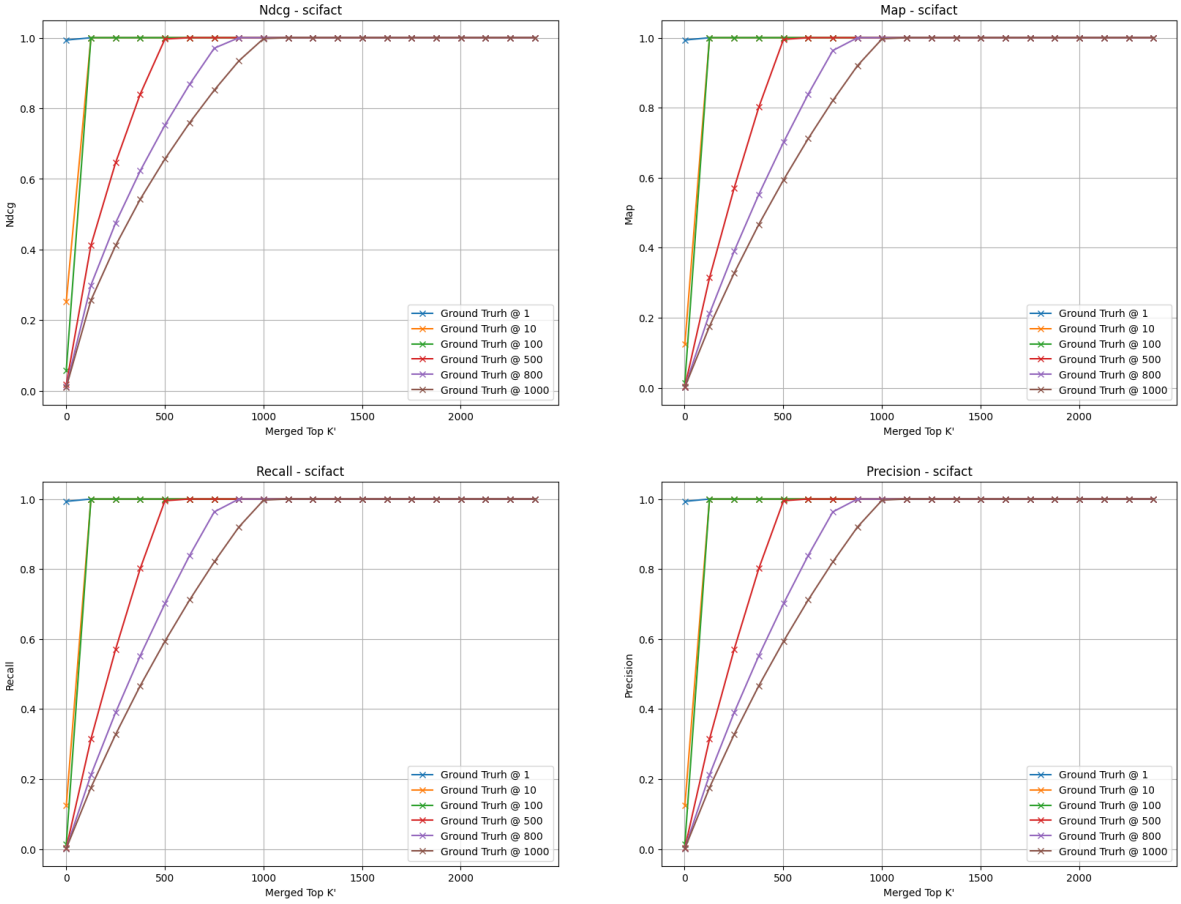

Figure 1: scifact dataset results

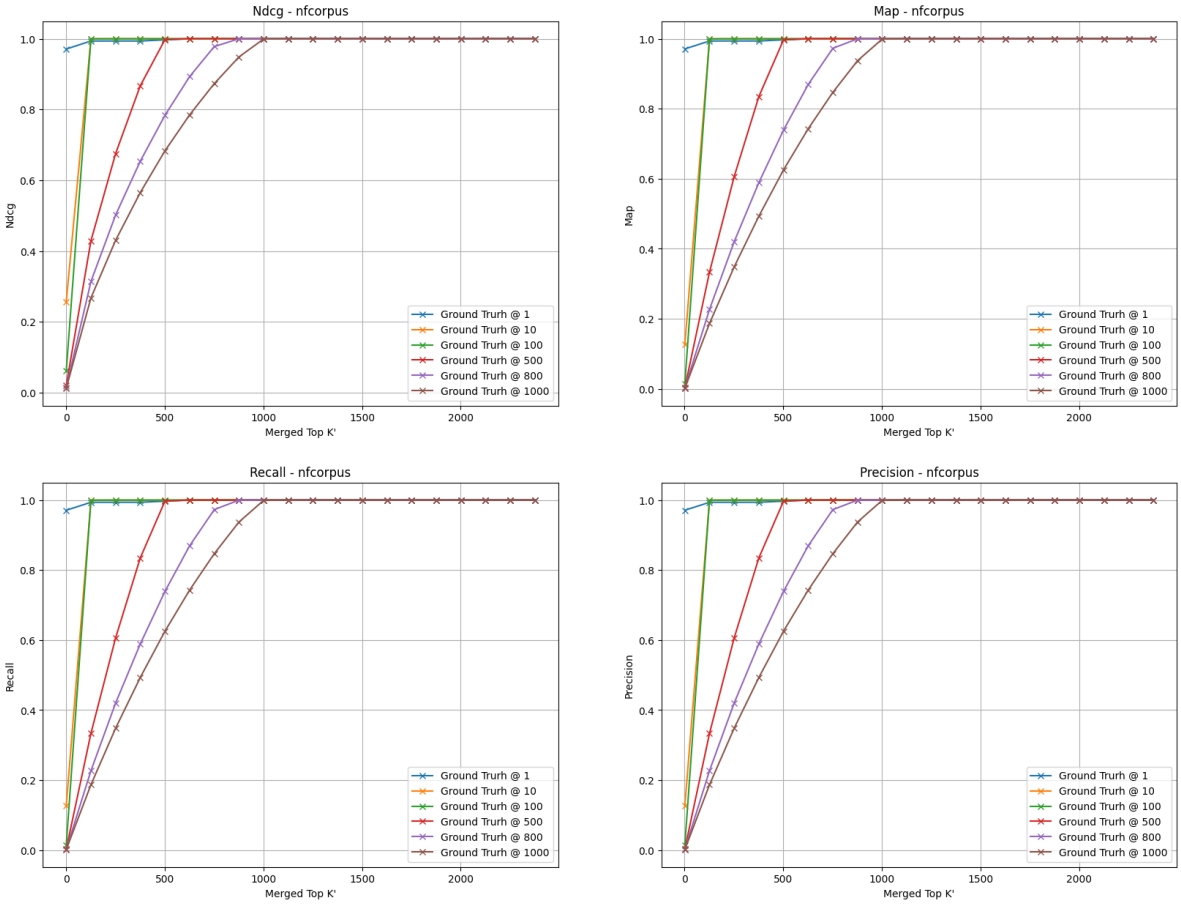

Figure 2: nfcorpus dataset results

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

4 Conclusion

In this article we have 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 our results will be.