Università
Ca'Foscari
Venezia

Master's Degree Course in
Computer Science and Information Technology

Artificial Intelligence and Data Engineering

Learning with Massive Data

Assignment 2

Studying Sparse-Dense Retrieval

Author
Giovanni Costa
Student ID
880892

Academic Year
2022-2023

This report describes an implementation of one of the most common techniques for retrieving the top-k documents in a collection, using a real-world ranking dataset. In particular, the purpose is to analyze an algorithm that exploit the linearity property of dot product to solve approximately the Maximum Inner Product Search (MIPS) problem for sparse and dense vectors.
In the experiment, no efficient dynamic pruning approaches and dedicated data structures are used.

The analyses are executed on the ranking datasets "Trec-COVID"[1] containing data from about 171000 documents and 50 queries, and "SciFact"[2] composed by 5100 documents and 1100 queries. BEIR [3] homepage[3], in addition, denote the ratio between relevant documents and queries that is for the two dataset 493.5 and 1.1, respectively.

1 Problem formalization

The rapid growth of the Web has led to an overwhelming amount of information available to each individual. Therefore, the need for an efficient and effective Information Retrieval systems (IR) have significantly increased, making them crucial tool for many people. Due to increasing of the popularity and accuracy of Machine Learning approaches, nowadays "Learning to rank" is become a crucially important area in Information Retrieval field. In particular, Supervised Learning approach was found to be a great tool to allow the creation of state-of-the-art models for ranking. Specifically, the called "Learning to rank" methods have usually the following characteristics [1]:

- Feature based: All the documents under investigation are represented by feature vectors, reflecting the relevance of the documents to the query. That is, for a given query q, its associated document d can be represented by a vector $x = \phi(d, q)$, where ϕ is a feature extractor. Typical features used in learning to rank include the frequencies of the query terms in the document, the BM25 and PageRank scores, and the relationship between this document and other documents.

- Discriminative training: The learning process, based on the training data, can be well described by the components of discriminative learning (input space, output space, hypothesis space, and loss function). This is also highly demanding for real search engines, because every day these search engines will receive a lot of user feedback and usage logs indicating poor ranking for some queries or documents. It is very important to automatically learn from feedback and constantly improve the ranking mechanism.

So, the process that goes from the extraction of documents in the collection, stored by the IR system, to the return of the top-k documents related to a given query

[1]`https://ir.nist.gov/covidSubmit/index.html`
[2]`https://github.com/allenai/scifact`
[3]`https://github.com/beir-cellar/beir`

(sorted by user's relevance score) can be very sophisticated and not trivial. If this system is thought as a cascade of functions/models, from the easiest to the most complex, it's important that the pipeline solves some specific problems for giving to the user good results in small amount of time.

Probably the most crucial step is the so-called "First Stage Retrieval": in practice, due to the huge size of the document collection it's infeasible analyzing the whole of them because the time for that it's usually intractable, even more so if the ML model used is complex.
Thus, this part should implement a method that is both cheap and effective, because given the entire collection it's required to take a portion of good relevant document in short time. After that, the document subset could be passed to the next steps that examine it deeply and return the list that is considered most appropriate (notice that the previous function is limited due to its written constraints).
For this purpose, Maximum Inner Product Search (MIPS) seems to be perfect because it's composed by a linear function and there are algorithms that can solves it very efficiently.

Regarding the data representation problem, in mathematical terms this means finding some functions $\phi_q(\cdot)$, $\phi_d(\cdot)$ that maps every element of $\mathcal{D}$ (the documents' collection) and $\mathcal{Q}$ (the set of queries) into real-value vectors. This task can be solved using sparse or dense embedding: the first technique capture more the lexical signals of relevance in sparse vectors, and the second encode instead semantic signals of relevance in dense vectors.
Ideally, the goal is to use the benefits of both representations and to be able to calculate these mapping independently (in order to precompute document embedding offline). So, more formally, the global structure of the problem considering is:

$$\mathcal{S} = \underset{d \in \mathcal{D}'}{\operatorname{argmax}_k} \langle q, d \rangle$$

where $\mathcal{D}' \subset \mathbb{R}^n$ is the collection of document vectors and $q \in \mathbb{R}^n$ is the query vector

Unfortunately, even though it's special properties, effective MIPS can't be computed if sparse and dense representations are used in a blended way, but there are different techniques that allow fast calculation and dynamic skipping according to what embedding type it's chosen.
Exploiting this fact, it's possible to take advantage of the linearity of the dot product and compute it separately for both dense and sparse representation, and in the end sum up the two results.
So mathematically is equal to write:

$$\begin{aligned} \langle q, d \rangle &= \langle q_{dense} \oplus q_{sparse}, d_{dense} \oplus q_{sparse} \rangle \\ &= \langle q_{dense}, d_{dense} \rangle + \langle q_{sparse}, d_{sparse} \rangle \end{aligned}$$

where $\oplus$ means concatenation.

Summarizing, performing these two smaller MIPS problems, it's possible to approximate the joint MIPS efficiently by:

1. Get the top-k' documents with a dense retrieval system defined over the dense portion of the vectors
2. Recover the of top-k' documents from a sparse retrieval system operating on the sparse portion
3. Merge the two sets and obtain the top-k documents from the combined and much smaller set.

As can be thought, starting from k value, as k' approaches the document collection size the final top-k set will become exact, but retrieval becomes slower. In fact, to compute the exact solution, it would be necessary to compute all the scores from dense and sparse representations and get the top-k, that is usually infeasible.

In this experiment having a moderate collection size, exact solution it's computed anyway in order to evaluate the results, and it's used to get the recall w.r.t. it: this means to measure whether the approximate algorithm "recalls" all the documents from the right solution. Formally:

$$\frac{|\mathcal{S} \cap \mathcal{S}'|}{|\mathcal{S}|}$$

2 Implementation and key points

The entire experiment is implemented in Python programming language and the main libraries used are *Nltk*[4], *NumPy* and *Scikit-learn*, very popular for data science tasks. In addition, for partial result storing and sharing, *Apache Parquet*[5] was used: this is a particular data format that allows efficient compression of data and fast retrieval, due to it saves information in contiguous memory locations using column wise manner.
Focussing instead on the usage of *Numpy*, it provides a dedicated data structure for high-performance numerical computation, the multidimensional array[6], and implements tree specific techniques: vectorized computations[7], limiting the number of copies of data in memory, and minimizing the number of operations [4]. This was a particularly crucial point because using it and its library functions helps to speed up the performance, that would otherwise be poor.

Regarding the encoding of the dense and the sparse embedding and the computation of the exact and the approximate top-k relevant documents, the following procedure was carried out, now explained in detail:

1. From the string of text, after the application of some preprocessing method, sparse vectors were build. In particular, only words and numbers were extracted from original text data, that after was made in lower case and stemmed.

[4]`https://www.nltk.org/`

[5]`https://parquet.apache.org/`

[6]Numpy is also used by many other software projects, such as Pandas or Scikit-Learn

[7]Vectorized operations are thus simply scenarios in which operations on vectors, such as dot product, transpose and other matrix operations, are performed on the entire array at once

Stop words are not removed in this case in order to not alter the text's meaning.
For convenience, TF-IDF implementation of Scikit-learn library was used: it allows optimized computation due to the usage of NumPy and enable the possibility to pass as schema one vocabulary V of terms $t_i \in V$. This features it's especially important because using the same vocabulary to encode both documents and queries the representations are made independent, so some operations can be precomputed offline. After that, dot product between the two well-formed vector was made.
More formally:

$$\vec{d} \in \mathbb{R}_{+}^{|V|}, \text{ where } d^{(t_i)} = \begin{cases} 0 & t_i \notin d \\ \frac{TF(t_i,d)}{IDF(t_i)} & t_i \in d \end{cases}$$

$$\vec{q} \in \mathbb{R}_{+}^{|V|}, \text{ where } q^{(t_i)} = |\{x \in q : x = t_i\}|$$

$$score_{sparse}(q, d) = \langle \vec{q}, \vec{d} \rangle$$

2. For the dense representation, it was sufficient to pass the vanilla text to the Sentence Transformers model [2] from Hugging face community[8] that return the contextual embedding of queries and documents. After that, dot product between the two vector was made. Also in this case, the document representation for example can be computed offline.

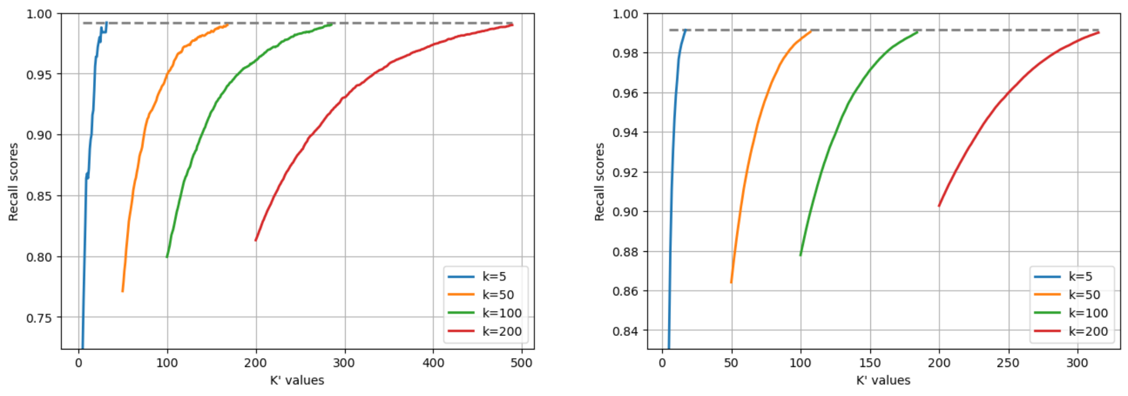
3. Now the two scores for all document-query pairs are summed taking into account the linearity of the dot product and the exact top-k documents solution was retrieved from these values.

4. Also the top-k' documents are retrieved from the previous computed scores and the sets are merged. Finally, the top-k documents are returned from this union set.

As can be noted by looking at the specific Jupyter notebook file, almost all the operations are done in vectorized manner, and data (also the score used in top-k' retrieval) are retrieved using indexes to cells of already done operations; the whole process it's in this way very efficient.

[8]`https://huggingface.co/sentence-transformers/all-MiniLM-L6-v2`

3 Experimental evaluation



(a) *Recall values with "Trec-COVID"* **(b)** *Recall values with "SciFact"*

Figure 1: *Recall w.r.t. exact solutions of the analyzed datasets*

Looking at the plots in figure 1, that show the analyses varying the top-k values in *[5, 50, 100, 200]*, it can be clearly seen the intuition anticipated in the previous chapter: starting from k, as k' approaches the size of the entire collection, the final top-k set will become the exact solution. In this case, an exit threshold was fixed due to a numerical stability problem and because the process, that for large k' become slow, could go ahead a lot in time only for one document that don't match exactly the solution (the average recall don't go to 1). However, considering the threshold 0.99 the results are considered reliable and appropriate.
The trend in all plots is increasing and also the recall values at the starting points of k', when they are equal to k, are quite good and rise when k gets higher, as expected for both plots.
Comparing instead the two plots related to the different datasets, some explanations are needed:

- The small fluctuations, more evident in plot 1a, and the fact that the minimum recall score of "Trec-COVID" is the lower between the two, may be caused mainly by the variety of the collection, which is larger and more heterogeneous. Also the ratio between relevant documents and queries for "Trec-COVID" is 493.5 compared to the value 1.1 of "SciFact" is influential: in this sense, it's intuitively easier to find the exact document list if there are less relevant documents because they are the few that have a higher score.

- Regarding the behavior of the k' values used to reach the recall threshold varying the values of k, it seems reasonable in both as well: in fact, it's not wrong to trust that as k rise, the top-k documents would contain elements less and less relevant with respect to the query, in particular if relevance D/Q value is high. In this sense also the scores are in some way flattened, and they can differ one to another by only a small contribution, taking consequently also the top-k' documents retrieved all near the same relevance value.

All these observations are dependent on the major limitation of this approximation technique: if the total document score is big but one of the specific dense or sparse's contribution is small or medium, that specific document could not appear in merged top-k' retrievals and consequently in the approximate solution set, even if it's present in the exact set. So it's needed to find a good tradeoff between effectiveness and efficiency.

Concluding, as summarized also in table 1, this technique is considered satisfying to approximate the exact MIPS problem. As can be seen, only by retrieving few documents for the top-k' sets, the mean recall values reach ≥ 0.99 and the computation time remains acceptable. Some future improvements may be using dedicated data structures and methods to dynamic prune the dense and sparse representations, and consequently solve the two specific MIPS problem in the optimal manner.

		k: 5	k: 50	k: 100	k: 200	Threshold	Relevant D/Q
Trec-Covid	k' values	32	168	285	489	0.99	493.5
SciFact		17	107	184	315	0.99	1.1

Table 1: *Experiment's summary table*

References

[1] Tie-Yan Liu. "Learning to Rank for Information Retrieval". In: *Foundations and Trends in Information Retrieval* 3.3 (2009), pp. 225–331. ISSN: 1554-0669. DOI: `10.1561/1500000016`. URL: `http://dx.doi.org/10.1561/1500000016`.

[2] Nils Reimers and Iryna Gurevych. "Sentence-bert: Sentence embeddings using siamese bert-networks". In: *arXiv preprint arXiv:1908.10084* (2019).

[3] Nandan Thakur et al. "BEIR: A Heterogeneous Benchmark for Zero-shot Evaluation of Information Retrieval Models". In: *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 2)*. 2021. URL: `https://openreview.net/forum?id=wCu6T5xFjeJ`.

[4] Stefan van der Walt, S. Chris Colbert, and Gael Varoquaux. "The NumPy Array: A Structure for Efficient Numerical Computation". In: *Computing in Science & Engineering* 13.2 (2011), pp. 22–30. DOI: `10.1109/MCSE.2011.37`.