Studying Sparse-Dense Retrieval: Part I

Michele Lotto 875922

1 Introduction

The second assignment for the Learning With Massive Data course consists in studying retrieval using both sparse vector and dense vector representations of the data; in particular, it is focused on efficiently retrieve the top-k documents given a query, by using Maximum Inner Product Search (MIPS) with vectors (documents and queries) that have both a dense subspace and a sparse subspace.

In other words, given a document $d = d_{sparse} \oplus d_{dense}$ and a query $q = q_{sparse} \oplus q_{dense}$ their inner product is defined as $d_{sparse} \cdot q_{sparse} + d_{dense} \cdot q_{dense}$. This can be approximated by breaking it into two smaller MIPS problems: retrieve the top-k' documents from a dense retrieval system defined over the dense portion of the vectors, and another set of top-k' documents from a sparse retrieval system operating on the sparse portion; this two sets are then "merged" and the top-k documents are retrieved from the merged set. As k' approaches infinity, the final top-k set will become exact, but retrieval becomes slower. This assignment focuses on studying the effect of k' by fixing k.

The provided code was executed on "Intel i7-4790 (8) @ 4.000GHz" (CPU) and on "NVIDIA GeForce GTX 1660 SUPER" (GPU) using "Python 3.10.6". All additional libraries used can be found in the "requirements.txt" file, included with this project.

2 Approach

The approach chosen to study the above problem was empirical. Indeed, the following steps are required to approximate the MIPS problem:

1. represent the given documents and queries using sparse vectors and dense vectors.
2. compute the dot product (score) for each document-query pair for the sparse representation and dense representation.
3. retrieve the top-k' sparse documents and the top-k' dense documents for each query.
4. merging the two representations for each query; the approach chosen for this step is simply summing up the sparse and dense scores for each document present in the top-k' sparse documents or in the top-k' dense documents. The merged set cardinality will be greater or equal to k' (equal if the sparse and dense retrieval systems contain the same documents) and less or equal to k' (equal if the sparse and dense retrieval systems contain all different documents).
5. retrieve the top-k documents from the merged set for each query.

The steps 1 and 2 are done together for both the sparse and dense representations by using respectively:

- ```
  def sparse_retrieval(documents: Dict[str, Dict[str, str]],
                       queries: Dict[str, str]
                       ) -> Dict[str, Dict[str, float]]
  ```

- ```
  def dense_retrieval(documents: Dict[str, Dict[str, str]],
                      queries: Dict[str, str]
                      ) -> Dict[str, Dict[str, float]]
  ```

Both these functions take in input:

- documents in the form of a `Dict` of:
  - key: document id (`str`)
  - value: `Dict` of:
    - key: `"title"` or `"text"` (`str`)
    - value: document title or document text (`str`)

- queries in the form of a `Dict` of:
  - key: query id (`str`)
  - value: query text (`str`)

Both these functions return a `Dict` of:

- key: query id (`str`)
- value: `Dict` of:
  - key: document id (`str`)
  - value: query-document pair score (`float`)

The steps 3, 4 and 5 are done using:

```
def merging(results_sparse: Dict[str, Dict[str, float]],
            results_dense: Dict[str, Dict[str, float]],
            k_prime:int,
            k:int) -> Dict[str, Dict[str, float]]
```

The input type is coherent with the return type of the **sparse_retrieval** and **dense_retrieval** functions. The return type is indeed the same as the **sparse_retrieval** and **dense_retrieval** functions.

2.1 The "sparse_retrieval" function

The **sparse_retrieval** function does two steps:

1. text cleaning and tokenization using the "**spacy**" [1] library for both documents and queries. This is done by:
   (a) loading the "en_core_web_lg" spacy model:
   ```
   _nlp = spacy.load("en_core_web_lg", disable=['parser','ner'])
   ```
   (b) applying the following lambda to queries and documents text:
   ```
   _tokenizer_cleaner = lambda text: [token.lemma_ for token in _nlp(text)
                                      if not token.is_stop and not token.is_punct]
   ```
   It constructs from the input text a list of lemmatized words which are not punctuation or stop words. The application of the above function is parallelized on CPU cores for documents.

2. BM25 score calculation using the **rank_bm25** [2] library; in particular it is used the **BM25Okapi** class (original formulation of BM25 from "Robertson et al".) with default parameters ($k_1 = 1.5$, $b = 0.75$, $\epsilon = 0.25$).

   Since in the BM25 original formulation some IDF values could be negative (if $CF > \frac{|D|}{2}$), this algorithm sets a floor on the IDF values to $\epsilon \times IDF_{avg}$, instead of adding a constant one before calculating the log value as "Lucene" variant does. [3]

   The **BM25Okapi** object is initialized by feeding the class constructor with the cleaned and tokenized document texts:

   ```
   bm25 = BM25Okapi(d.values())
   ```

   and the scores are retrieved for each query by:

   ```
   scores = bm25.get_scores(query)
   ```

   The queries scores calculation is parallelized on the CPU cores. For memory reasons, the query-document pairs which score is "0" are not included in the final results. For more details see Subsection 2.3.

[1] https://spacy.io/
[2] https://pypi.org/project/rank-bm25/
[3] https://www.ncbi.nlm.nih.gov/pmc/articles/PMC7148026/

2.1.1 BM25 as inner product of vectors

Let V be a vocabulary, d be a multiset of V terms contained in a document and q be a multiset of V terms contained in a query; it is possible to represent:

- each document as a vector $\vec{d} \in \mathbb{R}_+^{|V|}$ where $\vec{d_i}$ records the importance of the i^{th} term in V:

$$\vec{d_i} = \begin{cases} 0 & \text{if } t_i \notin d \\ \frac{TF(t_i,d)}{IDF(t_i)} & \text{if } t_i \in d \end{cases}$$

- each query as a vector $\vec{q} \in \mathbb{R}_+^{|V|}$ where $\vec{q_i}$ records the count of the i^{th} term of V in the query:

$$\vec{q_i} = \sum_{t \in q} \mathbf{1}_{\{t_i = t\}}, \text{ where } t_i \in q$$

Then:

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

2.2 The "dense_retrieval" function

The **dense_retrieval** function uses the BEIR framework [4]; in particular, it is used the **DenseRetrievalExactSearch (DRES)** class, which provides a simple interface to use SBERT (Sentence-BERT [5]) pre-trained models. The function does the following steps:

1. model loading:

   ```
   model = DRES(models.SentenceBERT("all-MiniLM-L6-v2"))
   ```

   As recommended in the homework proposal, it was chosen to use a lightweight model: "all-minilm-l6-v2".

2. construction of the retriever object:

   ```
   retriever = EvaluateRetrieval(model, k_values=[len(documents)],
                                 score_function="dot")
   ```

   The "`k_values`" parameter tells the retriever object in how many top document scores the user is interested; since in this use case we are interested in all the document-query pair scores, this parameter is set as a list with a single element: the number of documents. This forces the retriever object to compute all the scores.

   The score function is set to be dot product.

3. actual retrieval:

   ```
   results = retriever.retrieve(documents, queries)
   ```

2.3 The "merging" function

The **merging** does the following steps, for each query:

1. find the top k' sparse documents and the top k' dense documents by using the **heapq** Python module, as described in step 3.

2. merge the two above results by summing up the sparse and dense scores for each document present in the top-k' sparse documents or in the top-k' dense documents, as described in step 4:

[4] https://openreview.net/forum?id=wCu6T5xFjeJ
[5] https://www.sbert.net/

```
merged = { doc_id: relevant_sparse.get(doc_id, 0) +
                   relevant_dense.get(doc_id, 0))
           for doc_id in set(top_k_prime_documents_sparse) |
                         set(top_k_prime_documents_dense) }
```

If a document score is missing, the default value chosen is "0". As described in Subsection 2.1, for memory reasons, the sparse scores which are "0" are not included in the sparse result; indeed, in this step, is necessary to sum "0", when their missing score is required. In regards to dense results, there should not be missing values, but if it happens, the missing dense score should not interfere with the final result, for this reason the chosen default value is "0".

3. find the top k document from the merged set by using the **heapq** Python module, as described in step 5:

```
top_k_documents_merged = heapq.nlargest(k, merged, key=merged.get)

result[quey_id] = { doc_id: merged[doc_id]
                    for doc_id in set(top_k_documents_merged) }
```

3 Experimental Setup

The goal for this assignment is to study the effect of k' as k is fixed. For this reason it is necessary to have the exact top-k documents for each query to compare them with the approximated top-k documents for each query retrieved by the **merging** function. This is done by using:

```
def ground_truth( results_sparse: Dict[str, Dict[str, float]],
                  results_dense: Dict[str, Dict[str, float]],
                  k:int ) -> Dict[str, Dict[str, int]]
```

This function input and output types are the same of the "**merging**" function. See Section 2.

The comparison between the exact top-k documents and the approximated top-k documents for each query is done by:

```
def metrics_calculation( dataset:str,
                         results_sparse: Dict[str, Dict[str, float]],
                         results_dense: Dict[str, Dict[str, float]],
                         ks: list[int] = [50,100,150],
                         k_primes: list[int] = [x for x in range(20, 170, 5)]
                        ) -> Dict[int, Dict[int, Dict[str, float]]]:
```

This function takes in input sparse and dense results as the "**merging**" and the "**ground_truth**" functions, plus some additional parameters: `dataset (str)`, `ks (list[int])`, `k_primes (list[int])`; respectively the dataset name, the list of k values and the list of k' values.
The return type is a `Dict` of:

- k value (`int`)
- `Dict` of
  - k' value (`int`)
  - `Dict` of
    - `"ndcg"`, `"recall"` or `"precision"` (`str`)
    - associated metric score (`float`)

3.1 The "ground_truth" function

The **ground_truth** function does the following steps, for each query:

1. sum up the sparse and dense scores for all the documents.

```
documents_per_query = { doc_id: relevant_sparse.get(doc_id, 0) +
                                relevant_dense.get(doc_id, 0)
                        for doc_id in set(relevant_sparse) |
                        set(relevant_dense) }
```

If a document score is missing the same strategy described in Subsection 2.3 is applied.

2. find the top k documents:

```
top_k_documents = heapq.nlargest(k, documents_per_query,
                                 key=documents_per_query.get)

real_result[quey_id]={ key:1 for key in top_k_documents }
```

The scores are replaced with "1" for compatibility with the **EvaluateRetrieval.evaluate(...)** function from BEIR. For more details see Subsection 3.2.

3.2 The "metrics_calculation" function

The **metrics_calculation** function does the following steps, for each value of k:

1. calculate the ground truth by applying the homonymous function.
2. for each value of k':
   (a) calculate the merging result by applying the homonymous function.
   (b) evaluate the merging results by comparing them with the ground truth:

```
ndcg, _, recall, precision = EvaluateRetrieval.evaluate(ground_truth_k,
                                                         results,
                                                         k_values=[k])
```

The above function asks for a ground truth in which the scores are all set to "1". The "`k_values`" parameter tells the function on which values of k the metrics should be calculated; in this use case we only have a fixed k value, in which we are interested.

At the end of the computation the function saves the total results to file by using the **pickle** Python module:

```
with open(dataset+"_metrics.pkl", 'wb') as outp:
        pickle.dump(metrics_per_k, outp, pickle.HIGHEST_PROTOCOL)
```

and returns the total results.

3.3 Results

For experiments, it was chosen to use two different BEIR dataset: "TREC-COVID" and "SciFact"; for each of which it was applied the following pipeline:

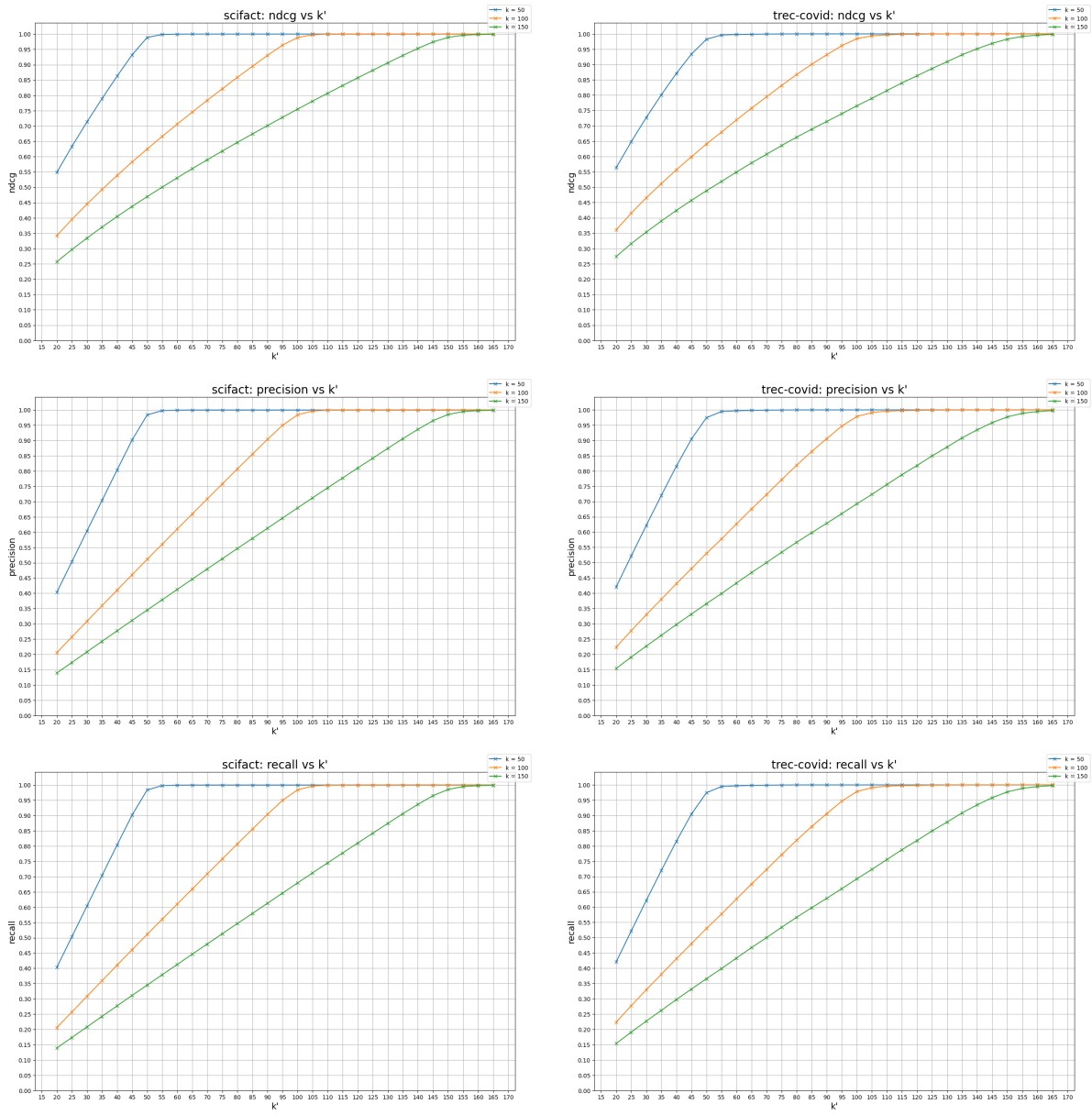
1. ```
   results_sparse = sparse_retrieval(documents, queries)
   results_dense = dense_retrieval(documents, queries)
   ```

2. ```
   metrics_per_k = metrics_calculation(dataset, results_sparse,results_dense)
   ```

3. ```
   plot_top_k_metrics_vs_k_prime(metrics_per_k)
   ```

   This last function generates three "metric vs k' value" plots: one for each used metric. Furthermore, each plot contains three different curve trends: one for each fixed k value.
   ```

For the sake of clarity, each dataset's pipeline is explored in a different notebook. Follow the plots for each metric.



4 Conclusions

In this report, it is discussed the implementation of the Maximum Inner Product Search (MIPS) approximation, by breaking it up into two smaller problems: "sparse MIPS" and "dense MIPS". As it can be seen from the plots, as k' approaches infinity the final top-k set becomes exact; in particular:

- the 0.90 value for the "precision" and "recall" metrics is reached when $k' \approx 0.90k$, as expected.
- the 0.90 value for the "ndcg" metric is reached when $k' \approx 0.85k$.

These results prove empirically that:

- the Maximum Inner Product Search (MIPS) can be approximated as described above.
- it is possible to use a value of k' smaller than k and obtain still decent results utilizing less computational power.