Studying Sparse-Dense Retrieval

1 Introduction

This paper will focus on retrieve information from a collection of documents by using Maximum Inner Product Search (MIPS) over dense vectors and sparse vector. Given a collection of documents, information retrieval helps in filtering out most important documents based on the keywords specified in the query provided by the user.

2 Sparse Representation

For the sparse representation we use the *TF-IDF (Term Frequency-Inverse Document Frequency)* measure. The formula that is used to compute the $tf - idf$ for a term t of a document d in a document set:

$$tf - idf(t, d) = tf(t, d) * idf(t) \tag{1}$$

and the idf is computed as:

$$idf(t) = log\frac{(1 + n)}{(1 + df(t)} + 1. \tag{2}$$

where n is the total number of documents in the document set and $df(t)$ is the document frequency of t.

- `TERM FREQUENCY`: This measures the frequency of a word within a document.
- `INVERSE DOCUMENT FREQUENCY`: IDF works by giving weight to the word. Weight here means how widely the word is used in the document. The more the word is used the lower will be its IDF.

Before creating the sparse representation of documents and queries, the data should be cleaned to improve the retrieval search. The following operations were applied to the data-set:

1. convert sentences to lower case
2. remove extra spaces
3. remove stopword

After cleaning up the text we can create our sparse representation of our dataset by applying the TfidfVectorizer class provided by Scikit.

In the following code-example, we create a TfidfVectorizer object. Then we fit the vectorizer to the corpus using the fit method, and transform the corpus

into a tf-idf matrix using the transform method. The tf-idf matrix is a sparse matrix, where each row represents a document and each column represents a term. The values in the matrix represent the tf-idf weight of each term in each document.

```
    # TF-IDF Vectorizer for documents
    def TFIDF(self,cleaned_corpus):
        self.tfidfvectoriser.fit(cleaned_corpus)
        tfidf_vectors=self.tfidfvectoriser.transform(
                                            cleaned_corpus)
        return (tfidf_vectors)
```

Similarly we do the same operation and create a sparse matrix also for the queries, where each row represents a query and each column represents a unique term in the corpus. The entries of the matrix represent the frequency of the corresponding term in each query.

To find the scores between each document and each query, we use the MIPS algorithm, which involves finding the documents with the highest inner product with each query. This can be done efficiently by computing the dot product between the tf-idf query matrix and the transpose of the tf-idf matrix (i.e., the tf-idf matrix where rows are terms and columns are documents).

	0	1	2	3	4	5	6	7	8	9	...	3623	3624
0	0.187755	0.305666	0.000000	0.000000	0.0	0.015051	0.014030	0.000000	0.0	0.0	...	0.077636	0.000000
1	0.000000	0.000000	0.000000	0.000000	0.0	0.000000	0.000000	0.006543	0.0	0.0	...	0.000000	0.005238
2	0.000000	0.000000	0.000000	0.000000	0.0	0.000000	0.000000	0.000000	0.0	0.0	...	0.000000	0.000000
3	0.000000	0.000000	0.000000	0.000000	0.0	0.000000	0.000000	0.000000	0.0	0.0	...	0.000000	0.000000
4	0.020986	0.021880	0.000000	0.000000	0.0	0.000000	0.000000	0.000000	0.0	0.0	...	0.000000	0.000000
...	...	...	...	...	...	...	...	...	...	...	...	...	...
3232	0.015058	0.000000	0.000000	0.000000	0.0	0.000000	0.030395	0.032723	0.0	0.0	...	0.000000	0.013098
3233	0.000000	0.000000	0.000000	0.000000	0.0	0.000000	0.000000	0.000000	0.0	0.0	...	0.000000	0.000000
3234	0.000000	0.000000	0.000000	0.000000	0.0	0.000000	0.000000	0.000000	0.0	0.0	...	0.000000	0.042291

Fig. 1: Sparse Score Matrix : $d_{sparse} \cdot q_{sparse}$

The resulting matrix [Fig.1] represents the similarity scores between each query and each document. The value in the i-th row and j-th column represents the similarity score between the i-th query and the j-th document. As we can see, since the scores matrix is sparse, it contains many cells containing 0. sorting the similarity scores in descending order for each query, we can retrieve the top-k documents that are most relevant to each query.

3 Dense Representation

The *all-MiniLM-L6-v2* model is a sentence-transformers model: It maps sentences & paragraphs to a 384 dimensional dense vector space and can be used to generate dense representations of input sentences. These representations are learned through a process of unsupervised training on a large corpus of text and

are designed to capture the semantic and contextual meaning of the text. The main advantage of the all-MiniLM-L6-v2 model is that it can generate high-quality dense representations that are well-suited for information retrieval. Here below the code for generating dense representations for our corpus using the all-MiniLM-L6-v2 pre-trained model in the SentenceTransformers library:

```
# Generate dense representations for the documents
    def embedder_documents(self,cleaned_corpus):
        embeddings = self.model.encode(cleaned_corpus)
        return embeddings
```

We apply the same operation also to compute the dense matrix for representing our queries. To compute the score matrix, we apply the same idea applied to the sparse representation, where we need to do the dot product between the document matrix and the query matrix.

	0	1	2	3	4	5	6	7	8	9	...	3623	3624
0	0.298085	0.357741	0.173295	-0.030394	0.014089	0.105433	0.088003	0.071574	0.018314	0.057710	...	0.325889	0.079962
1	0.015641	-0.018157	0.141233	0.084428	0.034168	-0.006077	0.071059	0.083993	0.109229	0.077600	...	0.072040	0.199295
2	0.166974	0.070494	0.173557	0.097512	0.140254	0.030020	0.098412	0.099059	0.206172	0.053680	...	0.248363	0.174810
3	0.085646	0.043782	0.164696	0.090844	0.115396	-0.002187	0.114407	0.111804	0.209825	0.133208	...	0.121491	0.174173
4	0.095827	0.055058	0.199045	0.120322	0.121814	0.003207	0.127507	0.127377	0.222800	0.147304	...	0.106256	0.175801
...	...	...	...	...	...	...	...	...	...	...	...	...	...
3232	0.064157	0.099452	0.223826	-0.014170	0.003560	0.013020	0.017448	0.067791	0.172141	0.196244	...	-0.017157	0.250344
3233	0.121772	0.147618	0.301386	0.051548	0.028571	0.098959	0.162023	0.247617	0.376419	0.304458	...	0.173963	0.244643
3234	0.050589	0.003400	0.110520	-0.026404	0.117410	0.002353	0.053139	0.070707	0.138257	0.085332	...	-0.076929	0.161893

Fig. 2: Dense Score Matrix: $d_{dense} \cdot q_{dense}$

The result [Fig.2] of the dot product is a matrix where the rows are the queries and the columns are the documents. The cells represent the score for that particular document-query combination. In this way, by sorting the score matrix and taking the first top-k values, we can have the best documents that respond to that particular query.

4 Sum of Sparse and Dense Matrices to find top-k documents

If a set of documents and queries represented as matrices have both sparse and dense representations to find the top-k documents for a specific query, we can take advantage of the linearity of the inner product operation and do the following computation:

$$S = d_{sparse} \cdot q_{sparse} + d_{dense} \cdot q_{dense} \tag{3}$$

The result will return a scores matrix $S \in R^{m*n}$, where m is the number of queries and n the number of documents and each cell represent the the sum of the scores of sparse and dense matrix for that determinate query and document.

Then by taking the first k rows, we can take the top-k most relevant documents for each query. Doing this is to ensure that the final top-k results are the most relevant and informative for the given query. However, even if this approach [Fig.

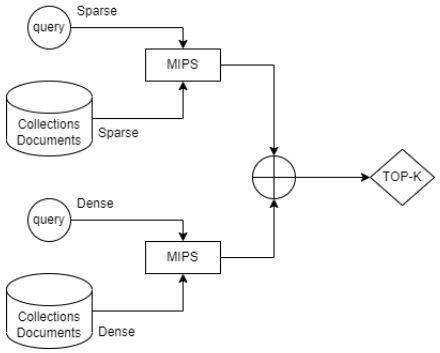


Fig. 3: Sum of Sparse Score Matrix and Dense Score Matrix Technique

3] return an accurate retrieval search, calculating the top-k documents related to a query this way is not very efficient in terms of time, so a different way will be implemented.

5 Merge top-k' to find top-k documents

To improve the computational time, instead of concatenating two relatively large matrices, we can reduce the dimensions by taking only the top-k' sparse score matrix documents and the top-k' dense score matrix documents, for each query [Fig 4]. We then create a new representation by merging the top-k' documents from the sparse and dense representation. This new matrix will be of size $m \cdot 2k'$, where m is the number of queries. From this new matrix we will extract the first top-k documents for each query as the most relevant documents.

Merging a sparse representation and a dense representation and take only top-k' documents can be still efficient (if the right values of k' and k are chosen) to implement because it allows to reduce the *dimensionality* of our problem by creating a merged matrix of size $m * 2k'$. With a smaller matrix the computation time is reduced [Table 1]. The problem now lies in the quality of the information retrieved.

6 Conclusion

Our goal is to examine the correctness of the algorithm above and study the effect of top-k' on the quality of the final top-k set. To verify this we calculate

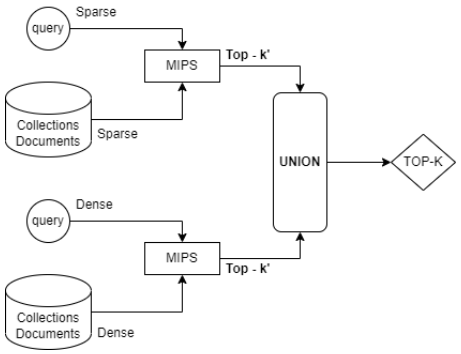


Fig. 4: Merge top-k' documents from Sparse Score Matrix and Dense Score Matrix Technique

Sum Technique Time	Merge Technique Time
332.98 seconds	9.53 seconds

Table 1: computational time for dataset "BeIR/nfcorpus", with top-k: 10 and top-k': 100

the quality measure recall:

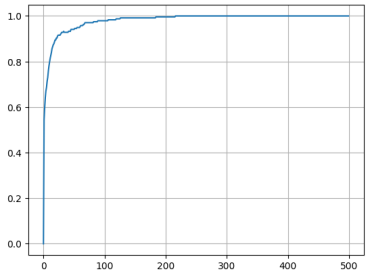
$$r = \frac{|S \cup S'|}{|S|} \tag{4}$$

Where S is the vector of the top-k documents for a query using the "sum technique" [Section: 4], and S' is the vector of the top-k documents of the same query coming from the "merge technique" [Section: 5].

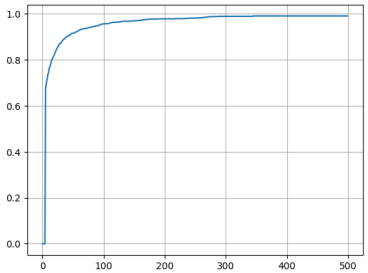
Our aim is to understand if it is possible to use the "merge technique" in which we merge the top-k' documents instead of the "sum technique" to reduce our computation time. To demonstrate this we take from the set created by the union of the top-k' documents and the top-k documents (denoted as S'), if the result is the same as the top-k documents obtained from the sum of the sparse scores and the dense scores (denoted as S) then it means that we can use the "merge technique". In this way, by reducing the dimensionality of the search set for the top-k documents we would have both a lower computation time and our retrieval algorithm returns the desired result.

After transforming the documents and queries into their corresponding sparse and dense representations and found the top-k documents for each query using "sum technique" and "merge technique", the *recall* is calculated as we see in equation (4) for each query in our dataset. Then, we can use these results to plot a graph [Fig. 5] and analyze the data.

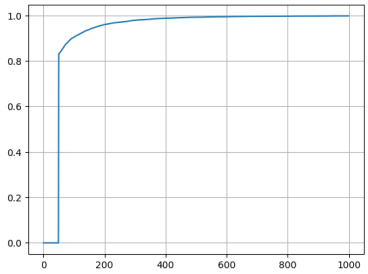
As we can see from the graphs in Fig. 5 if top-k' is similar or close to top-k the *recall* is low, and as top-k' tends towards infinity and moves away from top-k the *recall* also increases accordingly. Our goal is to achieve *recall* equal 1. When we take the top-k' documents, by construction the most relevant documents



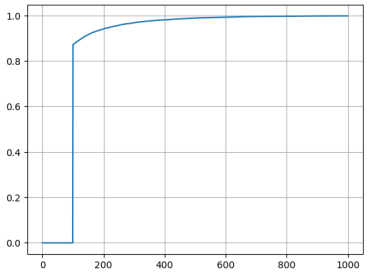
(a) Top k' = 500 top k = 1



(b) Top k' = 500 top k = 5



(c) Top k' = 1000 top k = 50



(d) Top k' = 1000 top k = 100

Fig. 5: Recall Plot for the dataset "BeIR/nfcorpus"

according to the sparse score matrix and the dense score matrix might not be the same. Because for the sparse matrix a document might be relevant but for the dense matrix it might not be, and vice versa. When we later create the set with the union of these results and and we choose a value of top-k' that is close to top-k, noise may have been introduced into the set created by the merge, and as this set is not large enough by selecting the top-k documents we may get the wrong results, however the computation is faster.

Conversely, as top-k' tends to approach the number of documents present in the dataset, the recall tends to reach 1. This is because the set created by merging the top-k' documents of both matrices is larger and our algorithm gives more importance to documents in both matrices by counting them twice, thus ignoring documents relevant to only the sparse matrix or only the dense matrix, however the computation is slower.

In conclusion the "merge technique" of the top k' documents of two sparse and dense matrices could be used to decrease the computation time and have good results in terms of retrieval search. The only problem is to find a value of top-k' bigger than top-k to select the right documents but at the same time it has to be much smaller than the number of documents or it wouldn't make sense to use this technique (it's perfectly fine to use the "sum technique").