Information Retrieval: Assignment 1

Document Search and Ranked Retrieval (self-implementation)

Michel Dierckx, Benjamin Powell

Code for the assignment: https://github.com/MichelDierckx/InfoRetrieval-Assignment1

Result: https://github.com/MichelDierckx/InfoRetrieval-Assignment1/blob/main/results/result.csv

1 Conceptual Details of Key Components

1.1 Tokenization

The tokenizer is responsible for preprocessing both the queries and the documents. Given a raw text, it produces a list of meaningful terms. Tokenization filters out stopwords that don't significantly contribute to the semantics of a document or a query. It also attempts to find a consistent representation of words that differ only in capitalization or syntax, making the index more compact.

1.2 Inverted Index

The inverted index maps each unique term to the documents containing that term, storing the frequency of each term per document. Since both the collection of terms and documents can be large, it must be both space efficient and must allow for quick retrieval of relevant documents given a query term.

The inverted index has three main components:

1. **Term-ID mappin**g: A dictionary that maps each unique term to a unique identifier.
2. **Term frequency matrix**: A sparse matrix in which each row represents a term and each column a document. An entry in row i and column j represents the term frequency of term i in document j. Since this matrix will contain a lot of zero entries, a sparse matrix format is used (lil_matrix / csr_matrix).
3. **Document Lengths**: A dictionary that maps each document to its precomputed document vector length. This is used for efficiently computing cosine similarities during document ranking.

1.3 Index construction using Partial Indexing

Constructing the entire index at once requires large memory requirements. The system potentially runs out of memory during the indexing process. Partial indexing alleviates this by splitting the indexing process in smaller batches. For each batch a smaller partial index is constructed and saved to disk. To create the final index, the partial indexes are efficiently merged.

1.4 Document Ranking

Documents are ranked according to relevancy for a given query using a vector space model. The SMART schema that is used is ltc.ltc. Both queries and documents are represented as vectors of tf-idf weights. The vectors are normalized to reduce the effect of document length or query length. Precomputing and saving the document lengths avoids having to iterate over all terms for a document in the frequency matrix, allowing for faster ranking. Given a query, the cosine similarity between the document vector and the query vector is computed for every document. Documents with vector representations closer in angle to the query vector are considered more relevant.

2 Implementation Details of Key Components

2.1 Tokenization

Tokenization is implemented as a **Tokenizer** class and is utilized by both the **Indexer** (tokenize documents) and **DocumentRanker** (tokenize queries**)**. The **Tokenizer** class relies on the NLTK package. Via NLTK's RegexpTokenizer class sequences of word characters (digits, underscores and letters) are extracted from an input text via a regular expression '\w+'. All sequences are then converted to lowercase. Common English stop words (such as 'we', 'be', 'and', …) are filtered out. The remaining sequences are then lemmatized, converting each to its base form. For example sequences like 'walking' or 'walked' will be converted to 'walk'. The final list of tokens is then returned.

2.2 Inverted Index

An inverted index is represented by the class **InvertedIndex**. Given the number of unique terms (m) and the number of documents (n), it instantiates an m x n sparse matrix (scipy.lil_matrix) representing the **term frequency matrix**. The **InvertedIndex** is also used to store a dictionary representing the **Term-ID mapping**. Via the method *add_document(document id, list of tokens)* the t**erm frequency matrix** is updated. For each token the term id is looked up in the **Term-ID mapping** and the entry at (term id, document id) is incremented by one. The *calculate_document_lengths* function computes the lengths of document vectors based on TF-IDF weights. It iterates through all terms in the term-frequency matrix, calculating and accumulating squared TF-IDF weights for each document. Finally, it takes the square root of each accumulated sum to produce the final vector lengths. These final lengths are saved in a dictionary **Document Lengths** allowing for quick retrieval of a document vector length given the document id. The matrix format scipy.lil_matrix is suited for random insertion, but is not as compact in memory as a scipy.csr_matrix. Using the method *finalize_index* converts the frequency matrix to the compact scipy.csr_matrix format.

2.3 TF-IDF weight calculation using term frequency matrix

The TF-IDF weights are calculated using $w_{t,d} = (1 + log(tf_{t,d})) \cdot log(\frac{N}{df_t})$. N represents the number of documents and is known. $tf_{t,d}$ (the frequency of term t in document d) corresponds to the entry at row t and column d in **the frequency matrix**. df_t(the number of documents that contain term t) corresponds to the number of non-zero entries in the **terms frequency matrix** at row t.

2.4 Index construction using Partial Indexing

The **Indexer** class is responsible for constructing the inverted index using partial indexing. The class method *create_index_from_directory* returns an instance of **InvertedIndex** for the specified documents directory using the following steps:

1. One by one the documents are tokenized using an instance of the **Tokenizer** class. Unique terms are collected as keys in a dictionary and assigned a unique identifier. For development purposes the resulting tokens for each document are saved as well.
2. The documents are grouped into batches, by default one batch will contain 1000 documents. For each batch a partial index will be constructed:
    a. For each batch, an instance of **InvertedIndex** is instantiated. The dimensions of the **terms frequency matrix** can be specified, since both the number of documents and the number of unique terms in the entire document collection are known (step 1).
    b. Each document's tokens are added to the **term frequency matrix** using *add_document(document id, list of tokens)*
    c. The **term frequency matrix** is converted to a more compact scipy.csr_matrix format using *finalize_index* and saved to disk
3. One by one the term frequency matrices of the partial indexes are loaded from disk and simply added together. This operation is very efficient for scipy.csr_matrices. This produces the final **term frequency matrix.** This final **term frequency matrix** is passed to a new instance of the **InvertedIndex** class together with the **Term-ID mapping**.
4. The document vector lengths are calculated and stored using *calculate_document_lengths*
5. This final **InvertedIndex** object is then saved to disk for later use.

2.5 Document Ranking

The **DocumentRanker** class ranks documents according to relevance given a query. A **DocumentRanker** object contains a **Tokenizer** object for query tokenization and an **InvertedIndex** object representing the final inverted index. The main method of interest is *rank_documents(query_string)* which returns a list of relevant document ids sorted in descending order by relevance to a given query string.

To accomplish this, the query string has to be converted to a query vector using *get_query_vector*. Using the **Tokenizer** the query string is split into terms. For each term that appears both in the query and the inverted index, the TF-IDF weight $w_{t,q}$ is calculated similarly to section 2.3, but the tf variable now corresponds to the number of occurrences of the term in the query. The vector of these TF-IDF weights is then normalized by dividing by the L_2 norm. The vector is represented as a dictionary mapping the term id t to the TF-IDF weight $w_{t,q}$.

The documents are then ranked via cosine similarity with the query vector. The process in *rank_documents(query_string)* is described in pseudo code (inspired by https://trevorcohn.github.io/comp90042/slides/WSTA_L2_ir_vsm.pdf):

```
set accumulator $a_d$ to zero  for all documents d
**for all** <t, $w_{t,q}$> in query-vector.values **do**
    retrieve posting list for t (using term_frequency_matrix)
    calculate $df_t$
    **for all** <$tf_{t,d}$, d> in posting list **do**
        calculate $w_{t,d}$ using $df_t$ and $tf_{t,d}$
        divide $w_{t,d}$ by the document vector length document_lenghts[d]
        $a_d \mathrel{+}= w_{t,d} \cdot w_{t,q}$
    **end for**
**end for**
sort documents with non zero $a_d$ by decreasing $a_d$
return the top k documents of the sorted list
```

Note that by precomputing and storing the document vector lengths we avoid having to iterate over all terms in a relevant document for each query.

3 Evaluation and analysis

Computing MAP@k and MAR@k for the large dev dataset yielded the following results:

- MAP@3 ≈ 0.43
- MAP@10 ≈ 0.41
- MAR@3 ≈ 0.09
- MAR@10 ≈ 0.21

A value of 0.43 for MAP@3 indicates that around 43% of our top 3 results for a query are relevant on average. For k = 10, the mean average precision drops to 41%. This means that if we were to pick one recommendation in the first 3 we would be slightly more likely to find a relevant one then when we would pick a recommendation from the first 10.

MAR@k measures the proportion of relevant documents retrieved out of the total number of relevant documents for each query, making it sensitive to the true total number of relevant items. If the true number of relevant items for a query is larger than k, it is impossible to reach a value for MAR@k of 1. Unlike for mean average precision, our mean average recall increases when k increases. This makes sense, since expanding our top list allows for more relevant documents to be captured in this list. For MAR@10 we obtain a value of 21%, which would be a perfect score if the true number of relevant documents would be around 50 on average. Looking at the given *dev_query_results.csv* file, this does not seem to be the case and our recall seems quite low. A possible culprit could be differences in tokenization. For example for query 930483 containing 'amf', the given *dev_query_results_small.csv* file recommends a document containing 'amfAR'. Our tokenization would produce a token 'amfar', thus the document would not be recommended by our VSM.

Our implementation does not use a positional index for phrase queries. The implementation builds the final inverted index incrementally by merging partial indexes. Additional preprocessing schemes were implemented: stop word removal, lemmatization and case formatting.