Assignment 1: Vector Space Model

Stig Hellemans, Can Çetinsoy

Conceptual Details

The high level implementation of a vector space model on large scale consists of several components:

1) **Text preprocessing:** This is applied both on the documents during index construction as on queries during search. Preprocessing includes tokenization, filtering and additional techniques like lemmatization and the removal of stopwords.
2) **Document index construction:** To efficiently handle the large dataset of approximately 500,000 documents, parallelized index construction is essential to complete processing within a reasonable timeframe and ensure memory feasibility. Single-Pass In-Memory Indexing (SPIMI) is used to create multiple smaller partial indexes that fit into memory, which are then periodically flushed to disk. In a subsequent phase, these partial indexes are merged into a single, consolidated inverted index. Due to the substantial memory requirements of loading the complete inverted index, it is stored in a binary format on disk for optimized reading and writing.
3) **Document search and retrieval:** The Vector Space Model (VSM) represents each document and query as vectors in a multidimensional space, where each dimension corresponds to a term in the vocabulary. The relevance of a document to a query is computed using cosine similarity between their vectors. The documents are ranked based on their similarity scores to the query. The higher the similarity, the more relevant the document is considered to be. Additionally, we implemented a phrase query boosting. The system can handle phrase queries by looking at the proximity of terms in the document using a k-width window on positional indices, where the proximity between query terms within the document is measured to provide a boost to phrase matches. When processing a large number of queries, parallelization is also needed in this step.
4) **Evaluation metrics:** To compare the performance of different implementations, we utilized evaluation metrics such as Mean Average Precision and Recall at K (MAP@K/MAR@K), as these provide more robust performance indicators compared to simple accuracy. Additionally, we considered important factors like 'database size' and 'query time,' since variations in these aspects can significantly impact overall implementation performance.

Implementation Details

Text preprocessing

The preprocess function which handles lemmatization and stop-word removal alongside the tokenization. Stop-word removal is the elimination of common words that do not contribute to relevance and lemmatization is converting words to their base forms, so that tenses or pluralities are negated. In this iteration, numbers are also completely removed to help with performance.

Initially, we attempted to use NLTK, taking inspiration from the assignment guide, however, due to integration issues, we implemented spaCy for stop-word removal and lemmatization. spaCy is a NLP (Natural Language Processing) library for Python that can do both of these processes [1].

Document index construction

To enable Single-Pass In-Memory Indexing, a batching process is employed to manage working memory demands with large document datasets. By processing documents in batches, each batch is assigned to multiple CPU cores, enabling parallelized construction of individual document indices. These per-document indices are then merged into a partial inverted index, where terms are sorted and the resulting structure is stored on disk as a binary file, serving as an intermediate output. A basic binary encoding and decoding architecture, implemented with Python's *struct* module, ensures efficient storage and retrieval of these indices. This setup allows the system to quickly serialize and deserialize data while maintaining a compact and performant structure.

In a second phase, all partial inverted indices are merged into a single, comprehensive inverted index. A priority queue (*heapq*) is used to incrementally read terms from each partial index, helping to control memory usage. This final inverted index is then ready for efficient document search and retrieval.

```
for batch in batched_document_dataset:
    build_partial_index(batch)
merge_partial_indices()
precompute_doc_norms()
```

Two different postings are implemented as a dictionary to enable fast look-up time. A simple posting dictionary with (DocID, $tf_{t,d}$) as key-value pairs serves as a base case. Next, the second implementation encompasses the collection of positional indices with (DocID, List['pos_idxs']) as a posting dictionary. This additional information is useful because it enables the ability to process phrase queries and use other proximity scores. Finally, document norms are precomputed to normalize the dot products between query- and document weight vectors during document search.

Document search and retrieval

Vector Space Model:

Our query function operates on a Vector Space Model (VSM), where each document is represented as a vector in a high-dimensional space. Each dimension corresponds to a unique term from our vocabulary. To identify relevant documents, we compute the cosine similarity between the vectors of the query and each document. Scores closer to 1 indicate a high relevance to the query, while scores near 0 suggest lower relevance. Documents are then ranked by their similarity scores.

This similarity is determined using term weighting, where each term's weight is calculated based on its term frequency (TF) within the document and its inverse document frequency (IDF) across the corpus. Terms that appear frequently in a document but are rare across other documents are given higher weights.

Our system relies on an inverted index to keep track of term occurrences. This index contains document frequency information and posting lists, which allow us to quickly locate documents containing the query terms. Additionally, each document has a normalization factor, ensuring that document length does not skew the relevance score.

To optimize performance, each query term is processed iteratively, so computations are restricted to only the essential parts.

Phrase query boosting: It can be a desired feature to favor documents in which query terms are close to each other. An additional boost besides cosine similarity can achieve this. K-window boosting is being used extensively in search algorithms [2]. But it’s important to keep the balance between promoting proximity of query terms and promoting the presence of as many (important) query terms as possible. To illustrate this notion, let’s suppose you promote proximity in excess. A sparse number of available query terms close to each other could result to be more relevant that a more dispersed presence of all query terms, and thus not only a few. Therefore we implemented the following:

```
$if \frac{(number\ q\ terms\ in\ doc)}{(total\ number\ q\ terms)} > threshold$:
    $boost = 1 + (boost\_factor) \cdot \frac{(max\_number\_terms\_in\_k\_window)}{(total\_number\_q\_terms)}$
$else$:
    $boost = 1$

$similarity\_score = cosine\_similarity \cdot boost$
```

Only the query terms with a certain fraction of query terms present get a proximity boost. Additionally, the boost is proportional to the maximal number of terms present in a window of k width divided by the total number query terms. This serves as an additional correction for the overpromotion in the case of few query terms closeby each other. The boost factor can tune the boosting effect even further.

Evaluation metrics

To evaluate the effectiveness of our retrieval model, we used Mean Average Precision at K (MAP@K) and Mean Average Recall at K (MAR@K). These metrics calculate how well our model ranks and retrieves documents. Precision and recall are calculated at a cutoff rank K. This limits the results to the top-K documents.

MAP@K measures the proportion of relevant documents with the top-K retrieved documents. Higher precision indicates higher relevancy. MAR@K measures the fraction of all relevant documents that appear in the top-K results. It shows how well the model retrieves relevant documents from the set. Both metrics are averaged across queries.

To increase performance, we used a parallel processing approach. It loads the database for each worker to speed up the calculations of MAP@K and MAR@K. We also account for factors like database size and query time, as these can impact the model’s efficiency in practical settings. Larger databases or complex queries can slow down response times, so measuring these aspects help us assess and improve overall performance.

Analysis

This final implementation is a result of several iterations. Main reasons for implementing more complex efficiency measures were purely because of resource constraints. The need for parallelization was to enable fast search and index construction. On the other hand, both partial index building and merging as well as the decision to have posting lists only on disk was because of working memory limits. Because otherwise, the VSM could not be scaled to the large dataset.

	VSM	VSM + phrase query boosting
MAP@3	25.6%	42.1%
MAR@3	5.1%	8.4%
MAP@10	21.3%	32.4%
MAR@10	14.0%	21.3%
Inverted index size (original dataset: 3.51 GB)	1.15 GB	2.32 GB
Index construction time	3.5 h	3.5 h
Query time (ms/query)	44	549
Vocab size (# terms)	7180128	7180128

The differences between the basic Vector Space Model (VSM) and the one enhanced with phrase query boosting are evident. Phrase query boosting improves both precision and recall, though it incurs notable costs: the inverted index size doubles, and query times increase twelvefold. Potential optimizations could help mitigate these downsides. Additionally, increasing the value of k leads to a decrease in precision but an increase in recall. Finally, the index size for the basic VSM reduces by a factor of 3, while with phrase query boosting, the reduction is only by a factor of 1.5.

Our implementation can be found at this Github link. For the k=10 ranked queries, we chose to upload the retrievals of the VSM with phrase query boosting because it achieved the best results.

Future work

A key challenge in index construction is managing concatenated and misspelled words, which inflate vocabulary size as each unique instance is treated as a separate term. Several enhancements could address these issues and improve index performance. For instance, incorporating synonym handling would likely enhance recall metrics. Leveraging semantic vectors could further improve performance by implicitly capturing synonym relationships. Additionally, index size could be reduced through techniques like delta encodings combined with efficient binary encoding, such as Variable Byte encoding [2]. Lastly, switching to a more efficient programming language, like C++ or Java, could significantly boost overall efficiency.

References

[1] Honnibal, M., & Montani, I. (2017). *spaCy 2: Natural language understanding with Bloom embeddings, convolutional neural networks and incremental parsing*.
[2] Manning, C. D., Raghavan, P., & Schütze, H. (2008). *An Introduction to Information Retrieval.* Cambridge University Press.