Report 3

SNumbers: u264332, u264443, u264202

Names: Levente Olivér Bódi, Riccardo Zamuner, Giada Izzo

Github repository: https://github.com/Levilevi01/Info_Retrieval/tree/main

Repository TAG: **IRWA-2025-part-3**

Introduction

As for the third assignment, we designed a retrieval pipeline filtering documents based on all query terms, namely a conjunctive query. All query terms had to be in the results of the search engine.

Part 1

We would like to reflect on what we did in the code, particularly in this part we reflect on the scoring method.

Section a

The TF-IDF method assigns a weight to each term in a document based on how frequently it appears in that document (TF) and how rare it is across the entire document collection (IDF). Thus, terms that are frequent in one document, but rare in other documents will generally get a higher score:

TF-IDF(t, d) = TF(t, d) * IDF(t, d)

The documents and the query are represented as vectors in the TF-IDF feature space. The relevance score is the cosine of the angle between the query vector (q) and the document vector (d), which measures the angle between them, so together they form:

$$\text{Score}_{\text{TF-IDF+CosSim}}(q, d) = \frac{q \cdot d}{|q||d|}$$

Section b

BM25 is a non-linear ranking function that is an improvement over the classic TF-IDF model, particularly for short queries. It calculates the score as the sum of weighted scores for each query term:

$$RSV_d = \sum_{t \in q} \left[\log \frac{N}{\mathrm{df}_t}\right] \cdot \frac{(k_1+1)\mathrm{tf}_{td}}{k_1((1-b)+b\times(L_d/L_{\mathrm{ave}}))+\mathrm{tf}_{td}} \cdot \frac{(k_3+1)\mathrm{tf}_{tq}}{k_3+\mathrm{tf}_{tq}}$$

The first term after the sum sign, in the parenthesis is the IDF for term t. tf_td is the raw document frequency of term t and document d, meaning the count of how many times t appears in d. The document length normalization component is L_d/L_ave, which is the length of the document over the average length of the documents in the collection, and b is responsible for giving less or more importance to the document normalization component.

tf_tq represents how many times the term t appears in the query q. For most standard keyword searches, this is 1, as queries are typically short and terms aren't repeated. However, BM25's general formula allows for weighted query terms.

k_3 is the tuning parameter, that limits the contribution of the query term frequency tf_tq to the final score. Just like k_1, k_3 ensures that the query term's contribution does not guarantee and increasing score forever. Usually, it is set between 1.2 and 2.

Section c

FILL THIS IN

Part 2

In this part we compare the different ranking functions.

The first point to consider is term saturation. While by TF-IDF the ranking of a document can highly increase due to term frequency in the document infinitely, BM25 takes care that longer documents do not get higher scores just because they key terms frequently. This is capped by k_1.

Another really important point to consider is document length. Normalization is based on the vector length (Euclidean norm), which implicitly penalizes very long documents by TF-IDF, whereas BM25 explicitly handles document length via the parameter b and L_d/L_avg. It penalizes documents longer than the average and favors shorter ones, assuming relevant information is often condensed.

The complexity of BM25 is non-linear, while for TF-IDF it is. This justifies why TF-IDF is simple to implement and highly interpretable, great baseline and useful for comparing documents for *any* vector embedding, but it can be sub-optimal. It doesn't account for term frequency saturation, meaning a long document repeating a word many times might rank highly even if the topic is marginal. However, BM25 is still a state-of-the-art keyword search, used also by Google, which generally provides better relevance than TF-IDF due to saturation and length normalization. As the backside of BM25, it requires tuning, is harder to implement, and the parameters k_1 and b must be tuned for a specific dataset to be optimal.

Part 3

In this part we would like to reflect on Word2Vec.

These three methods are ways to convert text into numerical vectors that capture meaning, moving from individual words to larger pieces of text. Word2Vec creates a vector for every single word, understanding its meaning based on its neighbors, and to represent a sentence, you usually just average these word vectors together.

Doc2Vec improves upon this by directly learning a unique vector for an entire document or paragraph, capturing the overall context better than a simple average. Doc2Vec is better than simply averaging Word2Vec vectors for representing a text because it is a trained model for document-level meaning, while Word2Vec averaging is just a mathematical approximation.

The modern Sentence2Vec is far more powerful than Word2Vec averaging because it generates a contextual vector for the entire sentence at once, accounting for word order and the full meaning of the phrase, which makes it the best choice for semantic search.