Implementation of the BM25 model

Andrea Barbieri

Data Science and Scientific computing - University of Trieste

Information Retrieval Final Project

A.Y. 2022/2023

Outline (a path is formed by laying one stone at a time)

1. The BM25 Model
2. Python Implementation
3. Evaluating the system

The BM25 model

Why the BM25?

- The **BM25** (also known as *Okapi BM25* or *Okapi weighting*) is a probabilistic information retrieval model born as an extension of the BIM
- One of the limitations of the BIM was that its formulation did not take into account **term frequency** and **document length**
- Thus, to improve the BIM model in 2000 the BM25 was proposed by Sparck Jones et. al.

BM25 scoring method (1)

- As for the BIM, the BM25 is based on a scoring system which assigns a score to each document given a query, where the higher the score the higher the relevance of the document
- The score is then used as criterion for **ranked retrieval** of documents
- The BM25 scoring systems takes also in account the aforementioned quantities, allowing also customization by means of tunable parameters

BM25 scoring method (2)

- The *Retrieval Status Value* for a document d is defined as:

$$RSV_d = \sum_{t \in q} \mathrm{idf}_t \cdot \frac{(k_1 + 1)\mathrm{tf}_{t,d}}{k_1((1-b) + b \cdot \frac{L_d}{L_{avg}}) + \mathrm{tf}_{t,d}},$$

where t is a term included in a query q, L_d is the length of document d and L_{avg} is the average length of all documents.

BM25 scoring method (3)

- For very long queries, it is possible to use an alternative scoring system which takes into account the term frequency inside the query:

$$RSV_d^{\text{alt.}} = RSV_d \cdot \frac{(k_3 + 1)\mathrm{tf}_{t,q}}{k_3 + \mathrm{tf}_{t,q}},$$

where $\mathrm{tf}_{t,q}$ is the frequency of term t inside query q and k_3 is a positive tunable parameter

The role of the parameters

- The BM25 has two (or three) tunable parameters:
  - $b \in [0, 1]$: normalization with respect to the length of the document, 0 no normalization, 1 full scaling
  - $k_1 \geq 0$: strength of term frequency scaling, 0 will take us back to the BIM, $k_1 \to \infty$ will use raw term frequency
  - $k_3 \geq 0$: strength of term frequency scaling for the query
- Parameters can be tuned to optimize the system in retrieving useful documents using a test collection

Relevance feedback (1)

- If feedbacks for relevance of the documents are available we can include them in the scoring method
- Let
  - $|VR_t|$ be the number of relevant documents containing term t
  - $|VNR_t|$ be the number of non-relevant documents containing term t
  - $|VR|$ be the overall number of relevant documents
  - N be the total number of documents
- Let S be the scaling factor of the retrieval status value of the term t in document d

$$S_{t,d} = \frac{(k_1 + 1)\mathrm{tf}_{t,d}}{k_1((1-b) + b \cdot \frac{L_d}{L_{avg}}) + \mathrm{tf}_{t,d}}.$$

Relevance feedback (2)

- Let finally R_t be the introduced relevance factor for term t and formulated as follows:

$$R_t = \frac{(|VR_t| + 1/2)/(|VNR_t| + 1/2)}{(df_t - |VR_t| + 1/2)/(N - df_t - |VR| + |VR_t| + 1/2)}.$$

- We can then use as scoring method the following:

$$RSV_d^{rel} = \sum_{t \in q} \log \left[R_t \cdot S_{t,d} \right].$$

Python Implementation

Classes

- For the Python implementation of the BM25, two classes were implemented:
  - `Document`: base class to represent documents, under the assumption that each document has a title and a content
  - `ProbIR`: the BM25 model, to correctly work it must have the internal members `corpus`, `idx`, `tf`, `idf`

ProbIR members

- `corpus`: a list of `Documents`
- `idx`: inverted index, it is a dictionary with terms as keys and the posting list as value
- `tf`: it is a dictionary with terms as keys and the sparse vectors containing the tf per document
- `idf`: it is a dictionary with terms as keys and the idf as values

ProbIR initialization

- If only the corpus is given, it is possible to automatically compute the needed objects calling the method `from_corpus()`
- The method will call the external functions `make_dict()` to create the dictionary and `inverted_index()` which will return a tuple containing the inverted index, the term frequency dictionary and the idf dictionary
- There is also the opportunity to use a stemmer while creating the dictionary
- It is also possible to directly initialize the class importing pre-computed objects

Queries

- Once the class is initialized, the user can perform queries using the `query()` method
- Besides the query, the user can modify the number of showed results (thus the first k ranking documents will be printed) and enable the stemmer (only if the class was initialized using it)
- User can also modify the parameters b, k_1 and k_3 of the scoring function, here renamed `b, k, k2`
- Finally, the user can also enable pseudo-relevance feedback specifying the number of documents to consider

Relevance feedback

- The `query()` method, once printed the results, will ask if the user is satisfied
- If not satisfied, the user will be prompted to highlight the relevant documents among the printed ones
- Given the relevant documents suggested by the user and assuming all the other ones were not relevant, the method `__query_relevance()` will be called, computing again the scores for the documents and returning an updated list
- The previous method will iterate until the user is satisfied

Evaluating the system

Dataset

- To test the implemented system the `CISI` ("Centre for Inventions and Scientific Information") dataset was used
- It contains 1,460 documents and 112 test queries
- Furthermore, 76 queries also had a list of relevant documents

Average R-Precision

- To assess the quality of the retrieval the Average R-Precision was used
- Let Q be a set of n test queries and R_i the number of relevant documents for the i-th query. The ARP score is then defined as:

$$ARP(Q) = \frac{1}{n} \sum_{i=1}^{n} \frac{\#\text{ rel. doc. in the first } R_i \text{ results}}{R_i}$$

- As baseline, the system with no tweaks in the entire test collection reaches an ARP of 9.31%

Parameters tuning

- To tune the parameters, the corpus was split in a train (50 docs) and test set (26 docs)
- The b and k_1 parameters were then tuned using a grid search with 11 values each
- The final parameters found were $b = 1$ and $k_1 = 1.2$, resulting in a training ARP of 34.78%
- On the test set, the tuned system returned an ARP value of 21.36%

Pseudo-relevance and actual relevance

- A sensible question would be, given the ground truth, can pseudo-relevance actually be useful and improve the system?
- Given that the average number of relevant documents per query is ~ 41 the system was tested with different pseudo-relevance values

PR value	ARP
NO PR	21.36%
5	8.11%
10	8.10%
25	7.9%
50	8%
100	8%

Future works

- **Error correction**: in the given code the k_3 score is implemented in the wrong way, correction is needed[1]
- **Export indexes**: implement a method to export the computed indexes[1]
- **Spelling correction** and **wildcard queries**
- **Low idf skip**: query optimization by avoiding words with low idf present in the query
- **BM25F**: go beyond the title-text assumption and put weights for the different zones

[1] Implemented posthumously on GitHub

Thank you for the attention!

And thank you for all the fish!