Table of Contents

[**Classical Information Retrieval (IR)** 2](#_Toc208518907)

[1. **Search Engines** 2](#_Toc208518908)

[2. **Digital Libraries and Archives** 2](#_Toc208518909)

[3. **Enterprise Search** 2](#_Toc208518910)

[4. **E-Commerce Search** 2](#_Toc208518911)

[5. **Q&A and Chatbots** 2](#_Toc208518912)

[**1.** **Indexing documents** 2](#_Toc208518913)

[2. **Processing user queries** 2](#_Toc208518914)

[**3.** **Retrieval and Ranking** 2](#_Toc208518915)

## **Classical Information Retrieval (IR)**

Classical Information Retrieval is the process of storing, organising, and retrieving unstructured information (mainly text documents) from a large collection, based on a user’s query. This type of IR typically uses models like the Boolean model, Vector Space model (TF-IDF), as well as Probabilistic models such as BM25. The major focus is to match those documents to user queries in order of relevance, rather than searching by exact keyword only.

Classical IR has been applied in numerous areas which are listed as follows:

1. **Search Engines** (Google, Bing, etc.) – The foundation of ranking documents before modern deep learning methods
2. **Digital Libraries and Archives** – The retrieval of academic papers, books, or even legal documents.
3. **Enterprise Search** – Searching documents, reports, or emails within an organisation.
4. **E-Commerce Search** – Matching customer queries with relevant products.
5. **Q&A and Chatbots** – Retrieving text snippets which may respond to a user’s query.

At its core, Classical IR performs three major tasks:

1. **Indexing documents**
   1. Preprocesses raw text (tokenisation, stop-word removal, stemming/lemmatisation).
   2. Creating an inverted index (maps words onto documents that contain them).
2. **Processing user queries**
   1. The idea is to take the query, apply the same preprocessing, and represent it in a form which can be compared with documents such as vector representation.
3. **Retrieval and Ranking**
   1. Uses a chosen IR model:
      1. Boolean Model - Retrieves documents that exactly match keywords with AND/OR/NOT.
      2. Vector Space Model (TF-IDF) - Computes similarity between queries and documents using cosine similarity.
      3. Probabilistic Models (BM25) – Estimates the probability of the relevance of those documents and rank them accordingly.
   2. Returns the most relevant documents ranked by score.

|  |  |  |  |
| --- | --- | --- | --- |
| Step | Description | Code Snippet | Rationale |
|  | Install required Python packages. | !pip install scikit-learn pandas | The scikit-learn library provides the TF-IDF vectoriser and cosine similarity function. Pandas is used to create and manage the results in a structured DataFrame. |
|  | Import the cosine similarity function. | from sklearn.metrics.pairwise import cosine\_similarity | Cosine similarity is a metric used to measure how similar two vectors are, regardless of their size. It is the standard way to compare TF-IDF vectors. |
|  | Define a function to rank documents for a list of queries using TF-IDF and cosine similarity. | def rank\_tfidf(queries: List[str], top\_k: int = 5) -> pd.DataFrame: | This function encapsulates the logic for the TF-IDF ranking process, making it reusable for any list of queries. It returns a clean DataFrame for easy analysis. |
|  | Transform the text query into a TF-IDF vector. | q\_vec = vectorizer.transform([q]) | The same vectoriser that was fit on the documents must be used to transform the query, so they are in the same feature space. |
|  | Calculate similarity scores between the query and all documents. | sims = cosine\_similarity(q\_vec, X)[0] | X is the precomputed TF-IDF matrix for all documents. This line calculates the cosine similarity between the query vector (q\_vec) and every document vector in X. |
|  | Get the indices of the top-k most similar documents. | ranked = sims.argsort()[::-1][:top\_k] | argsort() gets the indices that would sort the array. [::-1] reverses it to get descending order (highest score first). [:top\_k] slices the list to get only the top-k indices. |
|  | Populate a list of dictionaries with result details for the top-k documents. | rows.append({ "query": q, "doc\_id": idx, ... }) | This builds a list of records that will later be converted into pandas DataFrame. It captures the query, document ID, its rank, the similarity score, and a text preview. |
|  | Install the BM25 ranking library. | !pip install rank\_bm25 pandas | The rank\_bm25 package provides an efficient implementation of the BM25 ranking algorithm, which is an alternative to TF-IDF. |
|  | Import the BM25 implementation class. | from rank\_bm25 import BM25Okapi | BM25Okapi is a specific and effective variant of the BM25 algorithm used for ranking documents based on query terms. |
|  | Initialise the BM25 model with pre-tokenised documents. | bm25 = BM25Okapi(tokenised\_docs) | BM25 requires the corpus (collection of documents) to be pre-processed into tokens (lists of words). tokenised\_docs are a list of lists, where each sub-list contains the words of a document. |
|  | Define a function to rank documents for a list of queries using the BM25 algorithm. | def rank\_bm25(queries: List[str], top\_k: int = 5) -> pd.DataFrame: | This function encapsulates the BM25 ranking process, mirroring the structure of the rank\_tfidf function for easy comparison. |
|  | Tokenise the input query. | q\_tokens = q.split() | The BM25 model expects queries as a list of tokens. This uses a simple space-based split, which is a basic method that might be improved with better tokenisation (e.g., lowercasing, removing punctuation). |
|  | Get the BM25 relevance scores for all documents against the query. | scores = bm25.get\_scores(q\_tokens) | The get\_scores method of the bm25 object calculates the BM25 score for every document in the index relative to the provided query tokens. |