Command Line Information Retrieval System

Bhakti S. Khot
Dublin City University
Dublin, Ireland
bhakti.khot2@mail.dcu.ie

Abstract

This paper presents the design, implementation, and evaluation of an Information Retrieval (IR) system built to analyze and rank documents from the Cranfield dataset. The system integrates three different retrieval models: Query Likelihood Model (QLM), Boolean Retrieval, and Vector Space Model (VSM), each utilizing distinct indexing and ranking techniques. A detailed analysis of these models highlights their strengths and limitations in document retrieval. The system's performance is evaluated using standard IR metrics, including Mean Average Precision (MAP), Precision at 5 (P@5), and Normalized Discounted Cumulative Gain (NDCG). The evaluation process is streamlined using the trecEval library, enabling efficient and consistent assessment of retrieval effectiveness. This study provides insights into the comparative performance of different retrieval models and their suitability for structured text collections.

Keywords

IRS, Vector Space, Boolean, Indexing, Ranking, CranField dataset

ACM Reference Format:
Bhakti S. Khot. 2025. Command Line Information Retrieval System . In *Proceedings of Dubin City University*. ACM, New York, NY, USA, 5 pages. https://doi.org/bhakti.khot2@mail.dcu.ie

1 Introduction

In today's digital landscape, we face a constant challenge: finding the needle of relevant information in the ever-growing haystack of digital content. Our information retrieval system addresses this challenge by combining three powerful search paradigms to deliver more accurate and comprehensive results than any single approach could achieve alone.

Built to work with the Cranfield collection—a classic benchmark in information retrieval research—our system demonstrates how theoretical approaches to search can be implemented in practical applications. While modern search engines employ numerous sophisticated techniques, our focus on three fundamental models provides clarity on the core principles that underpin even the most advanced systems.

What makes our approach unique is not just the implementation of multiple models, but how these models complement each other, each bringing different strengths to the search process. The Vector Space Model captures semantic similarities, the Boolean Model handles precise logical queries, and the Query Likelihood Model incorporates probabilistic reasoning—together providing a more holistic approach to determining document relevance.

Dubin City University,

ACM ISBN 978-x-xxxx-xxxx-x/YYYY/MM
https://doi.org/bhakti.khot2@mail.dcu.ie

2 Architecture Overview

2.0.1 Document Preprocessing & Linguistic Analysis. Before any searching can begin, our system transforms raw text into a more analyzable form through:

- **Tokenization** - Breaking text into meaningful units
- **Stopword removal** - Filtering out common words that add little meaning
- **Lemmatization** - Reducing words to their base forms

This linguistic foundation ensures that "running," "ran," and "runs" are all recognized as variations of the same concept, improving both recall and precision.

2.0.2 Indexing & Data Structures. The heart of our system lies in its indexing mechanisms:

- **Inverted indexes** mapping terms to documents
- **Term frequency calculations** capturing how often words appear
- **Document-term matrices** representing the relationship between documents and terms

These carefully constructed data structures allow for near-instantaneous retrieval that would be impossible with naïve sequential searching approaches.

2.0.3 Multi-Model Retrieval Engine. Each of our three implemented models approaches the search problem from a different angle:

- **Vector Space Model**: Represents documents and queries as vectors in a high-dimensional space, using cosine similarity to identify documents whose content most closely aligns with the query's direction. This model excels at capturing semantic relationships beyond exact matches.
- **Boolean Retrieval Model**: Provides precise control through logical operators (AND, OR, NOT), finding documents that exactly satisfy the query's conditions. This approach offers unambiguous results when exact matching is required.
- **Query Likelihood Model**: Takes a probabilistic perspective, asking "what is the probability that this document would generate this query?" Using Dirichlet smoothing, it balances document-specific information with general collection statistics, often capturing relevance that other models miss.

3 Indexing

The `build_inverted_index()` function represents a critical component of the information retrieval system. This function creates the necessary data structures to enable efficient document retrieval

and ranking across multiple retrieval models. Below is a detailed explanation of its implementation and significance.

3.1 Document Processing Pipeline

The function implements a comprehensive document processing pipeline that transforms raw XML documents into structured data suitable for retrieval operations:

(1) **Document Extraction**: The function iterates through each document (`doc`) in the XML root element, extracting the document ID (`doc_id`) and combining the title and text fields to form the complete document content.
(2) **Text Preprocessing**: Each document undergoes preprocessing via the `preprocess_text()` function, which likely performs:
   - Tokenization (breaking text into individual terms)
   - Case normalization (converting to lowercase)
   - Stopword removal (filtering out common words with little semantic value)
   - Stemming or lemmatization (reducing words to their base forms)
(3) **Corpus Building**: The processed tokens are stored in the `corpus` dictionary with the document ID as the key, creating a clean representation of each document for further processing.

3.2 Data Structure Construction

The function simultaneously builds several interconnected data structures that support different aspects of the retrieval models:

(1) **Document Representation**:
   - `doc_texts`: Stores the original, unprocessed text of each document
   - `corpus`: Contains the preprocessed tokens for each document
   - `doc_length`: Records the number of tokens in each document, essential for length normalization
(2) **Term Frequency Analysis**:
   - `doc_term_counts`: Captures the frequency of each term within each document
   - `collection_term_counts`: Tracks the global frequency of each term across the entire collection
   - `collection_length`: Maintains the total number of tokens in the collection
(3) **Inverted Index Construction**:
   - `inverted_index`: Maps each term to a list of documents containing it, with weights calculated using log scaling
   - `doc_freq`: Counts the number of documents containing each term, used for IDF calculations

3.3 Weighting Mechanism

The function implements a sophisticated weighting approach for terms in the inverted index: This logarithmic scaling formula:

- Reduces the impact of high-frequency terms
- Accounts for document length variation
- Implements a form of TF normalization where the weight is inversely proportional to the term's frequency in the document
- Uses a log base-2 scale to compress the range of values while preserving relative differences

4 Ranking

Ranking models in Information Retrieval (IR) are like the brains behind search engines—they decide which documents are the most relevant to a user's query. Their job is to sort search results in a way that brings the most useful information to the top. Different models take different approaches to scoring relevance, each with its own strengths and methods for understanding what the user is looking for.

4.1 Vector Space Model

The Vector Space Model (VSM) represents documents and queries as vectors in a high-dimensional space where each dimension corresponds to a unique term in the collection. This mathematical model enables quantitative assessment of document-query similarity through vector operations. Our implementation follows the classic TF-IDF weighting scheme and cosine similarity measure to produce a ranked list of relevant documents.

4.1.1 Key Components and Implementation Details.

4.1.2 ***Term Frequency (TF) Calculation.*** The `compute_tf` function serves as the foundation for document representation by calculating normalized term frequencies:

This function counts occurrences of each term in a document and then normalizes them by the document length. This normalization is crucial as it prevents bias toward longer documents, ensuring that term frequency reflects relative importance rather than document size.

4.1.3 ***TF-IDF Vector Generation.*** The `compute_tfidf_vector` function extends the term frequency with the inverse document frequency to create the final vector representation:

TF-IDF weighting scheme, which:

- Emphasizes terms that appear frequently in a specific document (high TF)
- De-emphasizes terms that appear across many documents (low IDF)
- Creates a balanced representation that highlights distinctive terms

The resulting vectors capture the semantic signature of each document, with higher weights assigned to terms that are both frequent in the document and distinctive across the collection.

4.1.4 Similarity Measurement with Cosine Similarity. The `cosine_similarity` function measures the degree of similarity between the document and query vectors:

- Calculates the dot product between vectors, considering only terms present in both
- Determines the magnitude (norm) of each vector
- Divides the dot product by the product of the norms
- Handles edge cases where vectors might have zero magnitude

Cosine similarity elegantly captures the angular similarity between vectors, focusing on direction rather than magnitude, which is particularly suitable for sparse document vectors.

4.1.5 Document Retrieval Process. The `vector_space_retrieval` function orchestrates the entire retrieval process:

(1) Preprocesses the query using the same method applied to documents
(2) Constructs a TF-IDF vector for the query
(3) Computes similarity scores between the query and all document vectors
(4) Sorts documents by decreasing similarity
(5) Returns the top 100 most similar documents

4.2 BM25 Search

4.2.1 Understanding BM25 Ranking Algorithm. BM25 (Best Matching 25) is a ranking formula that helps us find the most relevant documents for a search query. It's like asking, "How well does this document match what I'm looking for?" The algorithm gives each document a score based on how many of our search terms it contains and how important those terms are.

4.2.2 Document Length Normalization. First, the code calculates a normalization factor for document length using this function:

Copy`get_kval(dID, k1=1.2, b=0.75)`

This helps adjust scores based on document length. The formula used is: K = k1 * ((1 - b) + b * (dl/avdl))

Where:

- dl = length of the specific document
- avdl = average document length in the collection
- b = parameter that determines how much to consider document length (0.75 is common)
- k1 = parameter that controls term frequency scaling (1.2 is common)

For very long documents, we don't want to overvalue term frequency, so this normalization helps balance things out.

4.2.3 The Core BM25 Formula. (dID, ni, fi, qfi, k1=1.2, k2=100, b=0.75)|

calculates the score using three parts:

Part 1: IDF (Inverse Document Frequency). `p1 = (((ri + 0.5)/ (R - ri + 0.5)) / ((ni - ri + 0.5) / (N - ni - R + ri + 0.5)))`|

This measures how rare or common a term is across all documents. The rarer the term, the more valuable it is for matching. In our simplified version, ri and R are relevance parameters that are set to 0 (since we don't have relevance feedback), so it simplifies to:

IDF log((N - ni + 0.5) / (ni + 0.5))

Where:

- N = total number of documents
- ni = number of documents containing the term

Part 2: Term Frequency Normalization. Copy`p2 = (((k1 + 1) * fi)/ (kval + fi))`

This balances how many times a term appears in a document (fi). It uses a diminishing returns approach - seeing a term 10 times isn't 10 times better than seeing it once. The formula gives a score that increases with term frequency but levels off.

Where:

- fi = frequency of term in document
- kval = the normalization factor from our first function
- k1 = controls how quickly the term frequency value saturates

Part 3: Query Term Frequency. `p3 = (((k2 + 1) * qfi)/` (k2 + qfi))

This accounts for how many times a term appears in our query (qfi). Terms that appear multiple times in the query are probably more important.

Where:

- qfi = frequency of the term in the query
- k2 = controls the impact of query term frequency (100 means it has minimal impact)

4.2.4 Putting It All Together. The final BM25 score for each term is:

Copy`score = math.log(p1+1) * p2 * p3`

And our search function adds these scores to all matching terms between the query and the document.

4.2.5 The Search Process. In `search_bm25()` function:

(1) We go through each query
(2) Count how many times each term appears in the query (qtermdict)
(3) For each term in the query, find all documents containing that term
(4) Calculate the BM25 score contribution for that term
(5) Add up all the scores for each document
(6) Sort documents by their total scores
(7) Return the ranked results

This gives us a ranked list of the most relevant documents for each query, based on both the importance of the matching terms and how well they match.

4.3 Query Likelihood Model (QLM)

The **Query Likelihood Model (QLM)** helps determine how likely it is that a document could have "generated" a user's query. Unlike basic search models that simply match terms, QLM takes a **probabilistic approach**, estimating how well a document explains the query based on term distributions.

How It Works. When a user submits a query, the system:

(1) **Processes the query** – breaking it down into standardized terms.
(2) **Calculates a likelihood score** for each document, measuring how well it aligns with the query.
(3) **Applies Dirichlet smoothing** to account for missing terms, ensuring a more balanced probability distribution.
(4) **Ranks and returns the top 100 most relevant documents** based on their computed scores.

The **Dirichlet smoothing parameter** (default: **0.1**) plays a key role. It helps balance the importance of terms found in a document versus those commonly used across all documents. A **higher value** gives more weight to general language trends, reducing overfitting but potentially making results less specific for unique queries.

How QLM Compares to Other Models. QLM **improves upon traditional models** by providing a **more dynamic and context-aware approach** to ranking search results:

- **Boolean Model:** Only checks if a term exists in a document (**binary decision**) without ranking results.
- **Vector Space Model (VSM):** Uses **geometric similarity** (like cosine similarity) but doesn't fully consider how terms naturally appear in text.
- **QLM:** Treats documents as **probability distributions**, making it more effective at understanding term importance and handling missing words.

Why Use QLM?. I chose QLM because it better captures the **real-world intent** behind a search query. Instead of simply counting term matches, it evaluates how well a document **explains** the query, making search results more natural and relevant. Additionally, its built-in **smoothing techniques** help address vocabulary mismatches, ensuring robust and reliable ranking—especially in cases where queries are vague or complex.

5 Evaluation

Working with the Cranfield collection was a challenging yet rewarding experience. Initially, I struggled to understand how the dataset was structured and how to extract the necessary information. The dataset is stored in an XML format, but it does not have a root tag, which made it difficult to parse directly. When I first tried reading the XML files, I kept running into errors. After searching online, I found some GitHub links and used ChatGPT to learn how to properly parse XML files in Python. Eventually, I was able to modify the structure, add a root tag, and successfully read both documents and queries from the dataset.

5.0.1 Challenges Faced in Running Trec-Eval. One of the biggest hurdles in this project was compiling and running TrecEval to evaluate the search engine's performance. I initially tried running it on my system, but I kept facing compilation issues. I even tried using Google Colab, but that didn't work either. After hours of research and troubleshooting, I finally managed to run it and generate evaluation reports for my models.

Another challenge was to get the evaluation scores (MAP, P5 and NDCG) in the expected range (0.1 to 0.2). My results were initially far from this range, and after experimenting, I realized the issue was with query IDs. The Cranfield dataset's queries were not in sequential order, which seemed to affect the evaluation metrics. I fixed this by renumbering the query IDs from 1 to 255, which improved the results. However, this process was not easy, as I also encountered duplicate document ID issues in the trecEval output, which required additional debugging.

Evaluation Results Below is a table showing the evaluation results for different models based on Mean Average Precision (MAP), Precision@5 (P@5), and Normalized Discounted Cumulative Gain (NDCG):

The evaluation shows **significant differences** in how each model ranks and retrieves relevant documents.

1 Vector Space Model (VSM).

- **Best performing model** in terms of **MAP (0.1877)**, **P@5 (0.2560), and P@10 (0.2204)**.

	VSM	BM25	QLM
Map	0.1877	0.0954	0.1367
GM_MAP	0.0276	0.0506	0.0142
P_5	0.2560	0.0480	0.1991
P_10	0.2204	0.0533	0.1791

- **Why?**
  - VSM ranks documents based on **cosine similarity**, which works well when query terms appear frequently in relevant documents.
  - Since Cranfield is a **smaller dataset**, TF-IDF-based ranking gives good precision.

2 BM25 (Best Probabilistic Model).

- **MAP (0.0954) is lower** than VSM but **GM_MAP (0.0506) is the highest**, indicating better performance across all queries.
- **Why?**
  - BM25 **penalizes long documents**, which might not always be ideal in **Cranfield's short abstracts**.
  - The **low P@5 (0.0480) and P@10 (0.0533)** suggest that BM25 retrieves **some relevant documents but ranks them lower**, affecting precision.

3 Query Likelihood Model (QLM).

- **Performs better than BM25 in P@5 (0.1991) and P@10 (0.1791), but lower in GM_MAP (0.0142)**.
- **Why?**
  - QLM **models queries as probability distributions**, which helps in some cases but may not be as effective as TF-IDF for **short queries**.
  - Since the Cranfield dataset is **not large**, QLM's reliance on **language modeling** might not be as effective.

6 Conclusion

This project provided valuable hands-on experience in the building and evaluation of a search engine at a fundamental level. Although parsing the data set was straightforward, other aspects of the project, especially implementing trecEval for evaluation, proved to be quite challenging. Despite these difficulties, working on this project deepened my understanding of information retrieval (IR) techniques and how modern search engines function.

One of the biggest takeaways was realizing how the vector space model (VSM), BM25, and the query likelihood model (QLM) share similar core components, such as term frequency, document frequency, and inverted indexes, but apply different formulas to rank documents. This helped me to appreciate the strengths and weaknesses of different ranking models and why certain approaches work better depending on the data set.

Beyond the implementation, FutureLearn's content provided valuable insights into how large-scale search engines operate. Concepts like web crawling, metadata tags, robots.txt files, and indexing strategies helped me connect the theoretical foundations of this project to real-world applications.

Future Scope and Extensions Moving forward, I would like to expand beyond text retrieval and explore areas like image processing and image recognition. This would allow me to apply similar IR principles to visual data, broadening my understanding of search engines that handle multimedia content. Additionally, incorporating deep learning models could further enhance the ranking process, making search results even more accurate and context-aware.

This project has been a great starting point, and I look forward to diving deeper into more advanced IR techniques and real-world applications in the future (e.g., BERT for semantic search).

7 Citations and Bibliographies

My code base for the retrieval model, [7]. To build a solid understanding of information retrieval models, I first explored the concepts explained on FutureLearn [3] and then deepened my knowledge with additional resources like GeeksforGeeks [4]. When it came to implementing core retrieval techniques such as inverted indexing, TF-IDF computation, and cosine similarity, I followed the code from a GitHub repository [6] and supplemented it with a video tutorial that helped clarify these concepts. I also referenced another GitHub repository to incorporate some aspects of the implementation [2].

At first, I implemented a Boolean retrieval model to get a feel for how basic document retrieval works. Later, to integrate BM25 with the Cranfield dataset, so I referred to an existing implementation [1]. However, since my code already included vector space and query likelihood models, adding BM25 caused some compatibility issues. Rather than force it into the same codebase, I decided to implement it separately. For the query likelihood model, I used another code reference [10] to refine the implementation.

To guide my approach, I reviewed a research paper that provided a broad overview of information retrieval systems [11]. I also looked into more focused studies on query likelihood models [9], vector space models [8], and BM25 models [12] to better understand their theoretical foundations. My dataset the Cranfield dataset, which I referenced from [13].Reading from XML files, I found a useful implementation on GeeksforGeeks that helped streamline the process [5].

Let me know if you'd like any further refinements!

References

[1] Kousthubh Belur. 2018. Simple-Search-Engine/Phase1_Task3/BM25_A.py at master · kousthub93/Simple-Search-Engine — github.com. https://github.com/kousthub93/Simple-Search-Engine/blob/master/Phase1_Task3/BM25_A.py#L51. [Accessed 10-03-2025].

[2] Muhammad Burhan. 2019. Vector-Space-Model/Vector_Space_Model.ipynb at master · peermohtaram/Vector-Space-Model — github.com. https://github.com/peermohtaram/Vector-Space-Model/blob/master/Vector_Space_Model.ipynb. [Accessed 10-03-2025].

[3] FutureLearn. 2025. Sign in - FutureLearn — futurelearn.com. https://www.futurelearn.com/your-programs/mechanics-of-search/4. [Accessed 11-03-2025].

[4] GeeksforGeeks. 2013. What is Information Retrieval? - GeeksforGeeks — geeksforgeeks.org. https://www.geeksforgeeks.org/what-is-information-retrieval/. [Accessed 11-03-2025].

[5] Geeksforgeeks. 2025. Reading and Writing XML Files in Python - GeeksforGeeks — geeksforgeeks.org. https://www.geeksforgeeks.org/reading-and-writing-xml-files-in-python/.

[6] Aadarsha Ghimire. 2023. GitHub - cyberghimire/search-engine-udemy — github.com. https://github.com/cyberghimire/search-engine-udemy.

[7] Bhakti Khot. 10-03-2025. GitHub - Sabha95/CSC1121-Bhakti_Khot_10542: CSC1121-Bhakti_Khot_10542 Search Engine — github.com. https://github.com/Sabha95/CSC1121-Bhakti_Khot_10542/tree/master.

[8] D.L. Lee, Huei Chuang, and K. Seamons. 1997. Document ranking and the vector-space model. *IEEE Software* 14, 2 (1997), 67–75. doi:10.1109/52.582976

[9] Yuanhua Lv and ChengXiang Zhai. 2012. Query likelihood with negative query generation. In *Proceedings of the 21st ACM international conference on Information and knowledge management*. 1799–1803.

[10] QLM. 2013. QueryLikelihood/src/query.py at master · nhirakawa/QueryLikelihood — github.com. https://github.com/nhirakawa/QueryLikelihood/blob/master/src/query.py. [Accessed 11-03-2025].

[11] Amit Singhal et al. 2001. Modern information retrieval: A brief overview. *IEEE Data Eng. Bull.* 24, 4 (2001), 35–43.

[12] Krysta M Svore and Christopher JC Burges. 2009. A machine learning approach for improved BM25 retrieval. In *Proceedings of the 18th ACM conference on Information and knowledge management*. 1811–1814.

[13] Terrierteam. 2025. GitHub - terrierteam/jtreceval: A wrapper on trec_eval — github.com. https://github.com/terrierteam/jtreceval?tab=readme-ov-file#readme.