My Information Retrieval System for the Cranfield Collection: How I Built and Evaluated It

Aaqid Masoodi
Dublin City University
aaqid.masoodi2@mail.dcu.ie

March 14, 2025

Abstract

In this report, I share my journey of building an Information Retrieval (IR) system for the Cranfield collection, a classic dataset with 1400 aeronautical documents, 225 queries and relevance judgments [6]. I designed a modular system in Python, featuring parsing, indexing, ranking and evaluation components. I opted for an inverted index for fast retrieval [4], tailored preprocessing to keep aeronautical terms [1] and implemented three ranking models: Vector Space Model (VSM) with cosine similarity [13], BM25 with probabilistic scoring [12] and a Dirichlet-smoothed Language Model (LM) [7]. Using multiprocessing and `trec_eval`, I measured performance with Mean Average Precision (MAP), Precision at 5 (P@5) and Normalized Discounted Cumulative Gain (NDCG) [5]. BM25 performed best, but my results were lower than expected, prompting future improvements. This report details my implementation, choices and reflections, leaning on code snippets and ideas from referenced works.

1 Introduction

When I set out to tackle the Cranfield collection—a dataset from the 1960s with 1400 aeronautical documents, 225 queries and relevance judgments [6]—I wanted to create an IR system that was both efficient and tailored to its unique challenges. My goal was to parse, index, rank and evaluate this collection, learning from the process and testing different approaches.

I built my system in Python with a modular architecture, breaking it into four main parts:

1. **Parsing**: I extracted data from XML files (`cran.all.1400.xml`, `cran.qry.xml`) and TREC-formatted relevance judgments (`cranqrel.trec.txt`) using `xml.etree.ElementTree` and custom cleaning functions [17].

2. **Indexing**: I created an inverted index and precomputed statistics like TF-IDF vectors, saving them to disk for speed [20].

3. **Ranking**: I implemented VSM, BM25 and LM, each with its own scoring method and data needs [10].

4. **Evaluation**: I used `multiprocessing.Pool` to run queries in parallel, outputting results in TREC format for `trec_eval` analysis [19].

I relied on libraries like NLTK for preprocessing [16] and NumPy for calculations, aiming for a system that's easy to maintain and extend. I made choices like preserving domain-specific terms and tuning ranking parameters based on the collection's nature and IR best practices [10]. Throughout, I adapted code snippets and concepts from references like Wu [20] and Spot Intelligence [13], which I cite in the bibliography. Here's how I did it, step by step.

2 Indexing

Building the index was my first big task, turning the Cranfield documents into something searchable. I handled this in `indexing.py` and `preprocess.py`, making choices about analysis, preprocessing and data structures.

2.1 Document Analysis and Preprocessing

I started by parsing the documents. In `parser.py`, my `parse_documents()` function pulled fields like `docno`, `title` and `text` from `cran.all.1400.xml`. Take Doc 1:

```
<doc>
<docno>1</docno>
<title>experimental investigation of the aerodynamics of a wing in a
    slipstream .</title>
<text>experimental investigation of the aerodynamics of a wing in a
    slipstream . an experimental study...</text>
</doc>
```

I hit a snag with a missing `<root>` tag, but I fixed it manually since the dataset won't change [17].

For cleaning, my `clean_text()` function normalized spaces and kept alphanumeric characters plus basic punctuation (I wanted to put the regex here but Latex was going crazy). Doc 1's title became:

"experimental investigation of the aerodynamics of a wing in a slipstream." I wanted to preserve numbers like "Mach 2" for aeronautical relevance [3].

In `preprocess_text()`, I tokenized with NLTK's `word_tokenize`, filtered stopwords (but kept terms like "wing" and "flow") and stemmed words over four characters with Snowball. For Doc 1's title: - Tokenized: ["experimental", "investigation", "aerodynamics", "wing", "slipstream"] - Filtered: Same, since "wing" stayed. - Stemmed: ["experiment", "investig", "aerodynam", "wing", "slipstream"] I picked Snowball over Porter for speed [16] and tweaked stopwords for the domain [21].

2.2 Indexing Construction and Data Structures

In `build_inverted_index()`, I gave title terms double weight (e.g., `title_terms * 2 + text_terms`) since they summarize documents [10]. Here's a snippet I adapted from Wu [20]:

```
inverted_index = defaultdict(dict)
for doc_id, doc in documents.items():
    all_terms = title_terms * 2 + text_terms
    term_freq = defaultdict(int)
    for term in all_terms:
        term_freq[term] += 1
    for term, freq in term_freq.items():
        inverted_index[term][doc_id] = freq
```

For Doc 1, "wing" got a boosted frequency. I calculated IDF with a BM25-inspired formula: $\log((N - df + 0.5)/(df + 0.5) + 1)$ [12] and built normalized TF-IDF vectors with sublinear TF $(1 + \log(1 + \text{freq}))$ [18].

My data structures included: - `inverted_index`: A `defaultdict(dict)` for O(1) term lookups [20]. - `doc_vectors`: Normalized vectors for VSM. - `idf_values`, `doc_lengths`, `term_frequencies`: For BM25 and LM. I saved everything to `index_data.pkl` with `pickle` to avoid recomputing [9]. The inverted index felt right for speed [4] and my runs confirmed frequent terms like "wing" were well-handled.

3 Ranking

Next, I tackled ranking in `ranking.py`, implementing VSM, BM25 and LM. Each had its own flavor and I made choices based on their strengths.

3.1 Vector Space Model (VSM)

For VSM, I built query vectors with sublinear TF and IDF, then used cosine similarity with `doc_vectors`. I borrowed this from Spot Intelligence [13]:

```
similarity = sum(weight * doc_vector.get(term, 0) for term, weight in
    query_vector.items())
```

For Query 1 ("what similarity laws must be obeyed..."), terms like "similar" matched well. I liked VSM's simplicity and its focus on term overlap [10] and precomputed vectors made it fast.

3.2 BM25

BM25 was trickier but promising. I used this formula:

$$\text{score} = \sum_{\text{term}} \text{idf} \times \frac{\text{tf} \times (k_1 + 1)}{\text{tf} + k_1 \times (1 - b + b \times \frac{\text{doc\_length}}{\text{avg\_doc\_length}})}$$

with $k_1 = 1.2$, $b = 0.5$, adapting code from Lison [8]. I only scored documents in the inverted index for each query term, which saved time. I chose BM25 for its proven relevance ranking [12] and the parameters felt like a safe starting point [15].

3.3 Language Model (LM)

For LM, I went with Dirichlet smoothing ($\mu = 500$):

$$P(\text{term}|\text{doc}) = \frac{\text{tf} + \mu \times P(\text{term}|\text{collection})}{\text{doc\_length} + \mu}$$

I summed log probabilities for stability, inspired by Jurafsky and Martin [7]. It scored all documents, which was slower, but I liked how it handled sparse data. I picked $\mu = 500$ as a default [10], though I later wondered if it needed tuning.

3.4 Design Choices

I made all models output sorted `(doc_id, score)` lists and printed top-10 results for debugging. VSM was fast with vectors, BM25 leaned on the index and LM's exhaustive approach showed in my run times (e.g., 0.01-0.02s per query). The top-10 outputs helped me spot trends—like Doc 51 ranking high for Query 1 across models.

4 Evaluation

In `main.py`, I put everything together to test the Cranfield collection. Here's how I handled it and what I found.

4.1 Processing the Cranfield Collection

I parsed 1400 documents, 225 queries and 1837 relevance judgments, as my run showed:

```
Parsed 1400 documents from datasets/cran.all.1400.xml
Parsed 225 queries from datasets/cran.qry.xml
Parsed 1837 relevance judgments from datasets/cranqrel.trec.txt
```

For Query 1, preprocessing gave me ["similar", "law", "obey", "construct", "aeroelast", "model", "heat", "speed", "aircraft"], aligning with documents via my custom stopword list [1].

4.2 Evaluation Process

I ran queries in parallel with 8 CPU cores:

```
args = [(query_id, query_text, vsm, bm25, lm, output_dir) for query_id,
    query_text in queries.items()]
with Pool(processes=8) as pool:
    pool.map(evaluate_single_query, args)
```

I limited my debug run to 5 queries, saving results in TREC format (e.g., "1 Q0 51 1 0.1872 vsm") and evaluated them with `trec_eval` for MAP, P@5 and NDCG [5].

4.3 Results

Here's what I got from my actual run:

Model	MAP	P@5	NDCG
VSM	0.0804	0.2000	0.3328
BM25	0.0807	0.2400	0.3261
LM	0.0785	0.2400	0.3241

Table 1: My evaluation metrics for ranking models on the first 5 queries of the Cranfield collection.

4.4 Discussion

Looking at my results, BM25 edged out with a MAP of 0.0807 and P@5 of 0.2400, which makes sense given its index efficiency and parameter tuning [12]. For Query 1, Doc 51 scored high (24.9838), matching its relevance. VSM's P@5 (0.2000) showed decent early precision, like Doc 51 at 0.1872, thanks to cosine similarity [13]. LM lagged with a MAP of 0.0785, possibly because $\mu = 500$ didn't fit the collection's sparsity [7].

But honestly, these numbers—MAP around 0.08—are lower than I expected. Studies like PyTerrier Team [11] report MAPs closer to 0.2-0.3 for Cranfield. My debug run only covered 5 queries, which might skew things and I suspect my preprocessing or parameter choices (e.g., k_1, b, μ) need tweaking. The top-10 outputs look reasonable, though—Doc 12 topped Query 2 across models, fitting its aeroelastic focus. In the future, I'll run all 225 queries, tune parameters and maybe refine my stopword list to boost performance.

5 Conclusions

Building this IR system was a blast and I'm proud of how it handles the Cranfield collection. BM25 stood out, thanks to its smart use of the inverted index and probabilistic scoring [12]. My domain-specific preprocessing—like keeping "wing"—worked well [21] and the modular setup (e.g., separate `parser.py`, `ranking.py`) made development smooth. Too smooth.

Not everything clicked, though. LM's weaker performance and the low MAP scores across models tell me I've got room to grow. The manual XML fix was a hassle and LM's slow scoring bogged things down. Looking ahead, I'd: - Tune μ for LM with a grid search [10]. - Test query expansion (e.g., "aircraft" to "plane") [10]. - Use PyTerrier for better indexing and evaluation [11]. With my results lower than hoped, I'm motivated to refine this system further, building on this solid base.

6 Githuh Repository

MOSearchEngine on GitHub

A search engine implementation for the Cranfield dataset using Vector Space Model (VSM), BM25, and Language Model (LM) ranking algorithms. This project evaluates the performance of these models using the `trec_eval` tool.

References

[1] Analytics Vidhya. 2021. *Text Preprocessing in NLP with Python Codes*. Analytics Vidhya Blog. Retrieved March 14, 2025, from https://www.analyticsvidhya.com/blog/2021/06/text-preprocessing-in-nlp-with-python-codes/

[2] Chen, X. 2023. *BM25 for Python: Achieving High Performance While Simplifying Dependencies with BM25S*. Hugging Face Blog. Retrieved March 14, 2025, from https://huggingface.co/blog/xhluca/bm25s

[3] GeeksforGeeks. 2018. *Text Preprocessing in Python — Set – 1*. GeeksforGeeks. Retrieved March 14, 2025, from https://www.geeksforgeeks.org/text-preprocessing-in-python-set-1/

[4] GeeksforGeeks. 2020. *Inverted Index*. GeeksforGeeks. Retrieved March 14, 2025, from https://www.geeksforgeeks.org/inverted-index/

[5] Glater, Rafael. 2018. *Learn How to Use trec_eval to Evaluate Your Information Retrieval System*. Rafael Glater Blog. Retrieved March 14, 2025, from http://www.rafaelglater.com/en/post/learn-how-to-use-trec_eval-to-evaluate-your-information-retrieval-system

[6] ir_datasets. 2023. *Cranfield Dataset Documentation*. ir_datasets. Retrieved March 14, 2025, from https://ir-datasets.com/cranfield.html

[7] Jurafsky, Dan and Martin, James H. 2008. *Language Models for Information Retrieval*. In *Speech and Language Processing*. Stanford University. Retrieved March 14, 2025, from https://nlp.stanford.edu/IR-book/html/htmledition/language-models-for-information-retrieval-1.html

[8] Lison, Pierre. 2015. *rank-bm25: A Collection of BM25 Algorithms in Python*. PyPI. Retrieved March 14, 2025, from https://pypi.org/project/rank-bm25/

[9] Logic, Zxz. 2016. *ir-python: Information Retrieval in Python*. GitHub Repository. Retrieved March 14, 2025, from https://github.com/zxzlogic/ir-python

[10] Manning, Christopher D., Raghavan, Prabhakar and Schütze, Hinrich. 2008. *Introduction to Information Retrieval*. Cambridge University Press. Retrieved March 14, 2025, from https://nlp.stanford.edu/IR-book/

[11] PyTerrier Team. 2023. *Running Experiments*. PyTerrier Documentation. Retrieved March 14, 2025, from https://pyterrier.readthedocs.io/en/latest/experiments.html

[12] Robertson, Stephen and Zaragoza, Hugo. 2009. *The Probabilistic Relevance Framework: BM25 and Beyond*. Foundations and Trends in Information Retrieval 3, 4 (April 2009), 333–389. DOI:https://doi.org/10.1561/1500000019

[13] Spot Intelligence. 2023. *Vector Space Model Made Simple With Examples & Tutorial In Python*. Spot Intelligence Blog. Retrieved March 14, 2025, from https://spotintelligence.com/2023/09/07/vector-space-model/

[14] Stack Overflow Contributors. 2017. *BM-25 Search Algorithm Implementation in Python*. Stack Overflow. Retrieved March 14, 2025, from https://stackoverflow.com/questions/32537117/bm-25-search-algorithm-implementation-in-python

[15] Stack Overflow Contributors. 2020. *Implementation of Okapi BM25 in Python*. Stack Overflow. Retrieved March 14, 2025, from https://stackoverflow.com/questions/61877065/implementation-of-okapi-bm25-in-python

[16] Stack Overflow Contributors. 2017. *How Can I Make My Python NLTK Pre-Processing Code More Efficient?* Stack Overflow. Retrieved March 14, 2025, from https://stackoverflow.com/questions/46203023/how-can-i-make-my-python-nltk-pre-processing-code-more-efficient

[17] Stack Overflow Contributors. 2017. *Parsing Large Dataset with Python*. Stack Overflow. Retrieved March 14, 2025, from https://stackoverflow.com/questions/42725454/parsing-large-dataset-with-python

[18] Stack Overflow Contributors. 2018. *Python TF-IDF Algorithm*. Stack Overflow. Retrieved March 14, 2025, from https://stackoverflow.com/questions/49277926/python-tf-idf-algorithm

[19] TREC. 2023. *Text Retrieval Conference (TREC) Overview*. National Institute of Standards and Technology. Retrieved March 14, 2025, from https://trec.nist.gov/overview.html

[20] Wu, Fro. 2019. *Writing a Simple Inverted Index in Python*. Medium. Retrieved March 14, 2025, from https://medium.com/@fro_g/writing-a-simple-inverted-index-in-python-3c8bcb52169a

[21] Yadav, Vishal. 2021. *Indexing in Natural Language Processing for Information Retrieval*. Analytics Vidhya Blog. Retrieved March 14, 2025, from https://www.analyticsvidhya.com/blog/2021/07/indexing-in-natural-language-processing-for-information-retrieval