Albert Jané Lardiés - U215114
Marc de los Aires Tello - U198732
Jordi Esteve Claramunt - U215108
https://github.com/jordiestevee/irwa-search-engine-g_019
TAG: IRWA-2025-part-3

IRWA Project - Part 3 Report

Introduction

In Parts 1 and 2 of the IRWA project we:

- cleaned and standardized the **Fashion Products Dataset**, producing a consistent DataFrame suitable for search, and
- implemented a **TF–IDF + cosine similarity** retrieval system with an inverted index and a basic evaluation framework.

Part 3 builds directly on that work. The objective is to turn the existing prototype into a more realistic **ranking and filtering module** for an e-commerce fashion search engine by:

1. Implementing and comparing several ranking strategies:
    - TF–IDF + cosine similarity
    - BM25
    - a custom e-commerce–aware ranking function
    - a Word2Vec-based semantic ranker
2. Adding **filtering logic** (conjunctive matching) to restrict the candidate set to relevant products.
3. Evaluating and comparing these methods on labeled queries using standard IR metrics.

Data and Pre-Processing

As in previous parts, we use the **Fashion Products Dataset**, where each product contains:

- textual fields: title, description, product_details
- categorical fields: brand, category, sub_category, seller
- numeric fields: selling_price, actual_price, discount, average_rating, out_of_stock
- a unique identifier: pid

From Part 1, we reuse the preprocessing pipeline (tokenization, lowercasing, stopword removal, punctuation filtering, stemming, and removal of very short tokens) to create a cleaned token list per product stored in a tokens column. Product attributes in product_details are converted to text and integrated into the searchable content, while numeric fields are kept as numbers for later use in ranking (e.g. price, discount, rating).

From Part 2, we reuse the **inverted index** and **TF–IDF statistics**: for each term we store the list of documents that contain it and the term frequency within each document, as well as its document frequency and IDF weight.

In Part 3 we further compute:

- doc_length: number of tokens per product
- avgdl: average document length (used by BM25)
- a term-frequency dictionary tf of the form {term: {doc_id: freq}}

Query Pre-Processing

User queries are normalized with a dedicated function:

```
def query_normalizer(line):

    # 1) tokenize

    # 2) lowercase

    # 3) remove stopwords

    # 4) keep only alphanumeric tokens

    # 5) remove very short tokens

    # 6) apply Porter stemming
```

This guarantees that query terms are represented in the same space as document tokens (same stemming and stopwords), which is crucial for consistent matching across TF–IDF, BM25 and the custom ranker.

Ranking Methods

1. TF–IDF + Cosine Similarity

As a first ranking method we extend the TF–IDF baseline from Part 2 into a more general **cosine similarity ranker**:

- search_tf_cos(query, index, idf, tf, top_k=20)
- tf_cos_ranker(terms, docs, index, idf, tf)

Pipeline:

1. Normalize the input query with query_normalizer.
2. Find a **candidate set** of documents that contain all query terms by intersecting postings lists.
3. Build a TF–IDF vector for the query and for each candidate document:
   - tf comes from the inverted index.
   - idf was precomputed from document frequencies.
4. Compute the cosine similarity between the query vector and each document vector.
5. Return the top-k products sorted by similarity.

2. BM25

To improve retrieval quality for longer documents and handle term saturation more explicitly, we implement BM25:

- search_bm25(query, index, idf, tf, df, top_k=20)
- bm25_rank_documents(query, docs, index, idf, tf, df, k1=1.5, b=0.75)

Advantages over pure TF–IDF:

- **Term saturation**: repeating a term many times in the same description gives diminishing returns.
- **Length normalization**: prevents long product descriptions from dominating rankings just because they contain more words.

3. Custom E-commerce Ranking

To incorporate business signals relevant for an e-commerce scenario, we define:

- MySearch(query, index, idf, tf, df, top_k=20)
- MyRanking(terms, docs, index, idf, tf, df, TIW=0.9)

The idea is to combine standard textual relevance (cosine similarity over TF–IDF) with a **boost** derived from product metadata:

- **Average rating** (higher rating → better)
- **Discount** (strong promotions can be pushed up)
- **Recency** (newer items may be more attractive / relevant)

4. Semantic Ranking with Word2Vec

The final method uses Word2Vec to capture semantic similarity between queries and product descriptions.

Steps:

1. Training data: we build a corpus from the tokens of all products, optionally focusing on informative fields such as title, brand and category.
2. Model: we train a gensim.models.Word2Vec model (Skip-gram, chosen hyper-parameters for window size, dimensionality and min_count), using the fashion catalog as domain data.
3. Product vectors:
   - For each product, we take the average of its word embeddings, obtaining a dense vector vdv_dvd.
4. Query vectors:
   - A separate tokenizer w2v_tokenize_for_query normalizes the query.
   - w2v_vector_for_tokens averages Word2Vec vectors for the query tokens to get vqv_qvq.
5. Filtering:
   - We use conjunctive_docs(query, index) to restrict candidates to documents containing all query terms in the inverted index. This prevents semantic similarity from drifting into unrelated products.

6. Ranking:
    - For each candidate, we compute cosine similarity between vqv_qvq and vdv_dvd (implemented in w2v_search).
    - We return the top-k documents sorted by this similarity.

Filtering Strategy

All ranking methods in Part 3 use conjunctive filtering before scoring:

- For a query with terms {t1,t2,… }\{t_1, t_2, \dots\}{t1,t2,…}, we intersect the postings lists of each term.
- Only documents that contain all query terms are kept as candidates.

This design choice:

- significantly reduces computational cost (fewer documents to score),
- enforces strong topical relevance, and
- keeps semantic methods (Word2Vec) grounded in exact term occurrences.

Evaluation

Labeled Data and Queries

We focus on a small set of realistic fashion queries, for example:

- “women full sleeve sweatshirt cotton”
- “men slim jeans blue”

For each query:

1. The system retrieves ranked lists of products using each ranking method.
2. The lists are aligned with the validation labels to compute metrics.

Metrics

We reuse and extend the evaluation from Part 2:

- Precision@K
- Recall@K
- F1@K
- AP@K (Average Precision)
- MAP (Mean Average Precision)
- MRR (Mean Reciprocal Rank)
- NDCG@K (Normalized Discounted Cumulative Gain)

These are implemented by:

- precision_at_k, recall_at_k, ap_at_k, f1_at_k
- map_score, mrr, ndcg_at_k
- evaluate_labeled_query and run_one_query as helper functions.

Qualitative Results

BM25 typically improves the ranking of documents that match the intent but have moderately long descriptions, thanks to length normalization and term saturation.

When several products have similar lexical relevance, the custom score tends to push highly rated and discounted items slightly higher.

For queries using synonyms or slightly different phrasing, Word2Vec occasionally surfaces relevant items that lexical methods rank lower or miss entirely.

On the other hand, when the embeddings are trained only on the catalog (without large external corpora), performance can degrade for rare or very specific terms

AI Usage Statement

ChatGPT (GPT-5.1) was used to help structure and formulate this report and the accompanying README for Part 3. All code, experiments, and final evaluation runs were implemented, executed, and verified by the authors.