Part 2: Indexing and Evaluation

Fashion E-Commerce Search Engine

IRWA Project
https://github.com/stanciuantonio/irwa-search-engine

October 31, 2025

1 Introduction

This report presents the implementation of Part 2 of the IRWA Final Project, building upon the preprocessing and exploratory analysis completed in Part 1. The primary objectives are twofold: constructing an efficient inverted index with TF-IDF ranking capabilities, and rigorously evaluating system performance using standard Information Retrieval metrics. We implement a search engine that processes conjunctive queries (AND semantics) and ranks results by relevance, then assess its effectiveness against ground truth judgments from validation queries. The dataset comprises 28,080 preprocessed fashion product documents with a vocabulary of 20,906 unique terms.

Our evaluation reveals significant performance variation across queries, with moderate results on one validation query (Precision@10 of 0.200) but complete failure on another (all metrics at 0.000). These results expose fundamental limitations of baseline TF-IDF ranking with strict conjunctive search, providing concrete evidence for algorithm improvements needed in subsequent project phases. This report documents our indexing implementation, evaluation methodology, quantitative results, and detailed analysis of system weaknesses with proposed solutions.

2 Inverted Index Construction

2.1 Data Structure and Design

The inverted index serves as the core data structure enabling efficient document retrieval. Our implementation adopts a hybrid approach combining numerical term identifiers with set-based posting lists. The structure consists of three primary components: a term index mapping vocabulary words to integer IDs, the inverted index itself mapping term IDs to sets of document indices, and bidirectional mappings between document indices and product IDs (PIDs).

The architecture prioritizes search efficiency over memory compactness. Each unique term receives a numerical ID during index construction, reducing memory overhead compared to string-based keys in the core index structure. The inverted index stores posting lists as Python set objects rather than lists, enabling O(n) set intersection operations during conjunctive search instead of $O(n^2)$ list-based approaches. Document indices use internal integers (0 through 28,079) for compact representation and fast set operations, with O(1) bidirectional conversion to PIDs maintained through auxiliary dictionaries.

Listing 1: Inverted index construction

```
class InvertedIndex:
    def __init__(self, corpus):
        self.corpus = corpus
        self.pid_list = list(corpus.keys())
```

```
        self.pid_to_idx = {pid: idx for idx, pid in enumerate(self.
            pid_list)}

        # Extract vocabulary
        all_tokens = []
        for doc in corpus.values():
            all_tokens.extend(doc['searchable_text'])
        vocabulary = set(all_tokens)

        # Create term index: {term: term_id}
        self.term_index = {term: i for i, term in enumerate(vocabulary)
            }

        # Build inverted index: {term_id: set(doc_indices)}
        self.index = defaultdict(set)
        for doc_idx, pid in enumerate(self.pid_list):
            doc = corpus[pid]
            tokens = doc['searchable_text']
            for term in tokens:
                term_id = self.term_index[term]
                self.index[term_id].add(doc_idx)
```

Index construction completes in approximately 0.17 to 0.19 seconds on our corpus, indexing all 20,906 terms across 28,080 documents. This construction time remains acceptable for batch indexing, though incremental updates would require index persistence and delta updates in production systems. The completed index resides entirely in memory during query processing, trading memory consumption for query speed.

2.2 Conjunctive Query Processing

Our search implementation enforces strict conjunctive (AND) semantics: only documents containing every query term qualify for ranking. This approach maximizes precision at the expense of recall, excluding documents with partial term matches regardless of their potential relevance. The algorithm performs iterative set intersection across query terms, terminating early if any term has an empty posting list.

Listing 2: Conjunctive search algorithm

```
def search_conjunctive(self, query_terms):
    docs = None
    for term in query_terms:
        term_docs = self.get_docs_with_term(term)
        if not term_docs:
            return set()  # Early termination if any term missing
        if docs is None:
            docs = term_docs.copy()
        else:
            docs.intersection_update(term_docs)
    return docs if docs else set()
```

The set intersection approach scales linearly with posting list sizes rather than quadratically. For a query with k terms and average posting list size p, complexity approximates $O(k \cdot p)$ rather than $O(k \cdot p^2)$ for naive list-based intersection. However, the strict AND semantics prove problematic for queries with many terms or rare term combinations, as demonstrated by our test query "winter jacket warm waterproof" returning zero results despite individually common terms.

3 TF-IDF Ranking

3.1 Algorithm Implementation

After identifying candidate documents through conjunctive search, we rank results using the classical TF-IDF scoring scheme. Our implementation computes term frequency as raw occurrence count within the document, inverse document frequency as the logarithm of total documents divided by document frequency, and final score as the sum of TF-IDF products across query terms. The formulation follows:

$$\text{TF}(t, d) = \text{count}(t, d) \tag{1}$$

$$\text{IDF}(t) = \log\left(\frac{N}{\text{df}(t)}\right) \tag{2}$$

$$\text{Score}(q, d) = \sum_{t \in q} \text{TF}(t, d) \times \text{IDF}(t) \tag{3}$$

where $N = 28,080$ represents total documents, $\text{df}(t)$ counts documents containing term t, and $\text{count}(t, d)$ gives raw frequency of term t in document d.

Listing 3: TF-IDF ranker implementation

```python
class TFIDFRanker:
    def calculate_tf(self, term, doc_tokens):
        return doc_tokens.count(term)

    def calculate_idf(self, term):
        docs_with_term = len(self.index.get_docs_with_term(term))
        if docs_with_term == 0:
            return 0
        return math.log(self.total_docs / docs_with_term)

    def rank_documents(self, query_terms, candidate_doc_indices):
        scores = []
        for doc_idx in candidate_doc_indices:
            pid = self.pid_list[doc_idx]
            doc = self.corpus[pid]
            doc_tokens = doc['searchable_text']
            doc_score = sum(self.calculate_tfidf(term, doc_tokens)
                            for term in query_terms)
            scores.append((pid, doc_score))
        scores.sort(key=lambda x: x[1], reverse=True)
        return scores
```

This implementation deviates from the theoretically optimal cosine similarity formulation by omitting document length normalization. Consequently, longer documents accumulate higher raw TF-IDF scores, potentially dominating shorter but more focused documents. Additionally, we apply no field-specific weighting, treating matches in product titles identically to matches buried in lengthy descriptions. These design simplifications, while enabling straightforward implementation, contribute to suboptimal ranking quality as revealed by our evaluation results.

3.2 Test Queries

We defined five test queries representing diverse product search scenarios across categories, attributes, and specificity levels. Table 1 summarizes query performance, revealing the restrictiveness of conjunctive search.

Query	Results	Avg Score	Max Score
women cotton dress summer	273	13.672	13.672
men leather shoes formal	15	26.121	44.504
kids blue jeans comfortable	1	21.347	21.347
sports running shoes lightweight	3	30.068	32.287
winter jacket warm waterproof	0	0.000	0.000

Table 1: Test queries performance summary

The first query "women cotton dress summer" returns a substantial 273 documents, providing sufficient candidates for meaningful ranking. However, the next three queries return progressively fewer results (15, 1, and 3 documents respectively), limiting our ability to evaluate ranking quality when result sets shrink. Most critically, query five "winter jacket warm waterproof" returns zero results despite each term appearing frequently in the corpus individually. This failure illustrates how conjunctive search eliminates potentially relevant documents: a "winter jacket" with "warm" insulation but lacking "waterproof" in its description would be excluded entirely, even if highly relevant to user intent.

4 Evaluation Metrics Implementation

4.1 Metric Definitions

We implemented seven standard Information Retrieval metrics to assess search effectiveness: Precision at K, Recall at K, Average Precision at K, F1-Score at K, Mean Reciprocal Rank, Normalized Discounted Cumulative Gain, and Mean Average Precision. Each metric captures different aspects of retrieval quality, from binary relevance assessment to position-aware ranking evaluation.

4.2 Validation Query Results

We evaluated our system on two predefined queries from the provided ground truth file. Each query includes 20 manually labeled documents with binary relevance judgments (1 for relevant, 0 for non-relevant), enabling quantitative performance assessment.

4.2.1 Query 1: "women full sleeve sweatshirt cotton"

The first validation query seeks women's cotton sweatshirts with full sleeves, representing a typical fashion search with multiple descriptive attributes. Our system retrieved 500 documents matching all five query terms (women, full, sleeve, sweatshirt, cotton after stemming). The ground truth contains 20 labeled documents, of which 13 are marked relevant.

Table 2 presents evaluation results across cutoff thresholds K=5, 10, and 20. The system achieves modest performance with Precision@10 of 0.200, indicating only 2 of the top 10 retrieved documents are relevant according to ground truth labels. Recall@10 reaches 0.154, meaning the system recovers just 2 of the 13 relevant documents (15.4%) in the top 10 results. The Mean Reciprocal Rank of 0.333 reveals the first relevant document appears at position 3, providing moderate navigational quality.

The Average Precision values (0.058 at K=10) significantly trail Precision values (0.200 at K=10), indicating relevant documents appear scattered throughout results rather than concentrated at top positions. Similarly, the NDCG@10 score of 0.179 remains far below the ideal 1.0, confirming suboptimal ranking quality. These position-aware metrics demonstrate that while our system identifies some relevant documents, it fails to prioritize them effectively.

Metric	K=5	K=10	K=20
Precision@K	0.200	0.200	0.150
Recall@K	0.077	0.154	0.231
Average Precision@K	0.067	0.058	0.056
F1-Score@K	0.111	0.174	0.182
NDCG@K	0.170	0.179	0.195
MRR		0.333	

Table 2: Query 1 evaluation results: "women full sleeve sweatshirt cotton"

4.2.2 Query 2: "men slim jeans blue"

The second validation query seeks men's slim-fit blue jeans, another common fashion search pattern. Our system retrieved 222 documents matching all four query terms. The ground truth contains 20 labeled documents with 10 marked relevant.

Table 3 reveals catastrophic performance: all metrics equal exactly 0.000 across all cutoff thresholds. This result indicates not a single relevant document appears in the top 20 retrieved results. The system found 222 candidate documents satisfying the conjunctive query but ranked every single ground truth relevant document below position 20.

Metric	K=5	K=10	K=20
Precision@K	0.000	0.000	0.000
Recall@K	0.000	0.000	0.000
Average Precision@K	0.000	0.000	0.000
F1-Score@K	0.000	0.000	0.000
NDCG@K	0.000	0.000	0.000
MRR		0.000	

Table 3: Query 2 evaluation results: "men slim jeans blue"

Investigation reveals the disconnect between our rankings and ground truth. Ground truth relevant documents include PIDs like `JEAFVXG4GGZH9VFA` ("Slim Men Dark Blue Jeans") and `JEAFTGSGTYKZGAEZ` ("Slim Men Blue Jeans"). Our top-ranked results include `JEAFDWG9K7KPKHS8` ("Slim Men Blue Jeans") and `JEAFEC2GEMGBWHA5` ("Slim Men Light Blue Jeans"). Despite semantic similarity and identical query term matches, the TF-IDF scoring produces completely non-overlapping result sets between our rankings and ground truth labels.

This failure is not a software bug but rather evidence of fundamental algorithmic limitations. The TF-IDF scoring treats all term occurrences equally regardless of field location (title versus description), document quality signals (ratings, brands), or semantic variations ("dark blue" versus "blue"). The ground truth labelers presumably considered factors beyond term frequency (perhaps brand reputation, price reasonableness, or rating quality)that our baseline algorithm completely ignores.

4.3 Aggregate Performance

Across both validation queries, the system achieves Mean Average Precision of 0.028 and Mean Reciprocal Rank of 0.167. These aggregate metrics fall substantially below acceptable thresholds: typical production search systems target MAP above 0.30 and MRR above 0.50. The extreme difference between queries (Query 1 modest, Query 2 failed) indicates inconsistent performance dependent on query characteristics rather than reliable effectiveness.

5 Analysis and Proposed Improvements

5.1 Semantic Limitations of TF-IDF

The Query 2 failure exposes TF-IDF's inability to capture semantic relationships beyond exact term matching. Ground truth documents labeled as relevant for "men slim jeans blue" include products titled "Slim Men Dark Blue Jeans" where "dark blue" is treated as distinct from "blue" by term-based indexing. After stemming, "dark" becomes a separate token with its own IDF weight, altering document scores compared to documents containing only "blue". Users searching for "blue jeans" almost certainly intend to match "dark blue jeans", "light blue jeans", and "navy blue jeans", but TF-IDF has no mechanism to recognize these semantic equivalences.

Furthermore, our system processes product details attributes where "Color: Dark Blue" appears as a structured field value. While our preprocessing extracts "dark" and "blue" separately, it loses the semantic unit "dark blue". Query expansion or synonym handling could address this: a query for "blue" should automatically expand to include "dark blue", "navy", "denim blue", and similar color variants common in fashion. Alternatively, word embeddings (Word2Vec, GloVe) or contextual models (BERT) could capture that "blue" and "dark blue" are semantically proximate in vector space, enabling soft matching beyond exact tokens.

5.2 Field Weighting Absence

Our current implementation treats all text fields identically, summing TF-IDF contributions from title, description, and product details without differentiation. This approach ignores the intuition that title matches should weigh more heavily than description matches: a product titled "Men Slim Blue Jeans" more directly satisfies the query than one mentioning "slim" and "blue" separately within a lengthy description. Field-specific weighting would multiply scores by field importance factors, perhaps 3.0 for title, 1.0 for description, and 0.5 for product details.

Additionally, term position within fields carries semantic weight. Title-initial terms ("Blue Slim Men Jeans") typically identify primary product attributes, while title-final terms may indicate secondary features. Similarly, description-initial sentences often summarize key selling points, while later paragraphs detail care instructions or manufacturer information. Position-based weighting would decay term contributions by their offset within fields, prioritizing early-appearing matches.

The BM25 ranking algorithm provides principled field weighting through document length normalization and saturation effects. BM25 incorporates parameters k_1 (term frequency saturation) and b (length normalization) that can be tuned per field, naturally handling both field importance and within-field term significance. Implementing BM25 would address multiple current limitations simultaneously.

5.3 Metadata Integration

Our ranking completely ignores structured metadata that correlates with relevance. Consider two jeans products with identical text matches but different attributes: one from a premium brand (Levi's) with 4.5-star rating at $80, another from an unknown brand with 2.8-star rating at $15. Query-independent quality signals suggest the first product likely provides higher user satisfaction despite identical text relevance scores. Similarly, the category and subcategory fields directly indicate product type, yet we do not boost documents where category="Jeans" for a jeans query.

Integration approaches include additive boosting, multiplicative boosting, or learning-to-rank. Additive boosting adds fixed bonuses: $\text{Score}_{\text{final}} = \text{TF-IDF} + \beta_1 \cdot \mathbf{1}_{\text{category match}} + \beta_2 \cdot \text{rating}$. Multiplicative boosting scales base scores: $\text{Score}_{\text{final}} = \text{TF-IDF} \times (1 + \alpha_1 \cdot \text{rating}) \times (1 + \alpha_2 \cdot \text{discount})$. Learning-to-rank approaches like LambdaMART or RankNet train models on

features including text scores, metadata, and interaction patterns to predict optimal rankings from labeled data.

Even simple heuristics could improve results. For Query 2, filtering candidates to require category="Jeans" or subcategory="Bottomwear" before ranking would eliminate irrelevant document types. Using average rating as a tiebreaker when TF-IDF scores are equal (as they are for the 20 documents scoring 13.672 in Query 1) would promote higher-quality products. These straightforward enhancements require minimal implementation complexity while addressing obvious ranking deficiencies.

5.4 Conjunctive Search Restrictions

The zero-result outcome for test query "winter jacket warm waterproof" demonstrates how conjunctive search eliminates plausible matches. A winter jacket described as "warm and insulated" lacks the exact term "waterproof" despite potentially meeting user needs. Similarly, a jacket labeled "water-resistant" rather than "waterproof" would be excluded despite functional similarity. Strict AND semantics maximize precision at severe recall cost.

Relaxing to disjunctive (OR) search with scoring adjustments would retrieve documents matching any query term, then penalize scores based on missing term counts. A simple approach computes TF-IDF for present terms only, naturally yielding lower scores for partial matches. More sophisticated formulations explicitly penalize missing terms: $\text{Score}(q, d) = \sum_{t \in q \cap d} \text{TF-IDF}(t, d) - \lambda \cdot |q \setminus d|$ where λ controls the penalty for absent terms and $q \setminus d$ represents query terms missing from the document.

BM25 naturally handles missing terms by contributing zero to the score sum rather than eliminating documents entirely. This behavior provides a principled middle ground between strict AND and unpenalized OR, making BM25 migration particularly attractive for addressing multiple issues simultaneously.

6 Structural Modifications from Part 1

While Part 2 builds directly upon Part 1's preprocessing work, we introduced three minimal structural modifications to support efficient indexing and ranking operations. These changes preserve all preprocessing logic (tokenization, stemming, stop word removal) while enhancing data access patterns for search engine requirements. Importantly, no reprocessing of the corpus was necessary, and Part 1 results remain fully valid and reproducible.

6.1 PID at Root Level

The most significant structural change elevates the Product ID (PID) from nested within the original document object to the root level of the preprocessed document structure. Part 1's preprocessing returned documents structured as:

```
{
    "searchable_text": [...],
    "metadata": {...},
    "original": {
        "pid": "TKPFCZ9EA7H5FYZH",
        # other original fields
    }
}
```

Part 2 modifies this to:

```
{
    "pid": "TKPFCZ9EA7H5FYZH",  # Now at root level
    "searchable_text": [...],
```

```
    "metadata": {...},
    "original": {
        "pid": "TKPFCZ9EA7H5FYZH",  # Also preserved here
        # other original fields
    }
}
```

This modification addresses a critical indexing performance requirement. During inverted index construction, we iterate through all 28,080 documents, extracting PIDs thousands of times to build term-to-document mappings. Accessing `doc['pid']` requires a single dictionary lookup, while accessing `doc['original']['pid']` requires two nested lookups. At scale across millions of index construction operations, this difference becomes measurable. The PID remains duplicated in the original structure for backward compatibility with Part 1 code that expects it there.

6.2 Dictionary-Based Corpus Structure

Part 1 stored the preprocessed corpus as a list of documents, suitable for sequential processing during exploratory analysis. Part 2 introduces a helper function `load_preprocessed_corpus()` that converts this list to a dictionary indexed by PID:

```
def load_preprocessed_corpus(cache_path):
    with open(cache_path, 'rb') as f:
        corpus_list = pickle.load(f)

    corpus_dict = {}
    for doc in corpus_list:
        pid = doc.get('pid') or doc['original']['pid']
        corpus_dict[pid] = doc

    return corpus_dict
```

This transformation enables O(1) document retrieval by PID rather than O(n) linear search through the list. During ranking, we convert document indices back to PIDs to retrieve full document content for score calculation. With dictionary structure, `corpus[pid]` executes in constant time regardless of corpus size. For our 28,080 document corpus, the performance difference remains modest, but the pattern scales to million-document corpora where linear search becomes prohibitive.

The function includes fallback logic (`doc.get('pid') or doc['original']['pid']`) to handle both old and new document structures, ensuring compatibility if Part 1's preprocessed corpus is loaded directly without the PID elevation modification.

6.3 No Changes to Preprocessing Logic

Critically, these modifications affect only data structure organization, not preprocessing algorithms. The tokenization, stemming, stop word removal, and field extraction logic remains byte-identical to Part 1. We do not reprocess the raw JSON dataset; instead, we load Part 1's cached `preprocessed_corpus.pkl` file and apply structural transformations at load time or simply work with the existing structure where the PID is accessible from both locations.

This design choice reflects software engineering best practices: Part 1 focused on text processing correctness, while Part 2 optimizes data access patterns for its specific use case. The preprocessing module (`myapp/preprocessing/text_processing.py`) remains the single source of truth for how documents are processed, and no divergence exists between Part 1 and Part 2 in terms of what constitutes a "preprocessed document" semantically. The structural modifications simply reorganize this data for efficient algorithmic access during indexing and search operations.

7 Conclusion

This report documents our implementation of inverted indexing, TF-IDF ranking, and comprehensive evaluation for a fashion e-commerce search engine. We successfully constructed an efficient index covering 20,906 unique terms across 28,080 documents with sub-second build times, implemented conjunctive query processing with set-based intersection, and developed TF-IDF ranking with classical term frequency and inverse document frequency calculations. Our evaluation framework implements all seven required metrics (Precision, Recall, Average Precision, F1-Score, MRR, NDCG, and MAP) with mathematically correct formulations validated against ground truth labels.

The evaluation results reveal substantial system weaknesses. Query 1 achieves moderate performance with 20% precision and 15.4% recall in top-10 results, indicating the system identifies some relevant documents but fails to prioritize them effectively. Query 2 exhibits complete failure with all metrics at zero, retrieving 222 candidate documents but ranking every ground truth relevant document outside the top 20. This performance dichotomy exposes fundamental limitations rather than implementation bugs: TF-IDF with strict conjunctive search cannot capture semantic variations, lacks field-aware weighting, ignores quality metadata, and eliminates plausible partial matches.

The aggregate MAP of 0.028 and MRR of 0.167 fall far below production system expectations, requiring 5-10× improvement to reach acceptable effectiveness. Our analysis identifies four primary improvement directions: semantic matching through embeddings or query expansion, field-specific weighting with algorithms like BM25, metadata integration through quality signals, and relaxed boolean semantics balancing precision and recall. These enhancements will form the foundation of Part 3 development, where we implement BM25 ranking, custom scoring with quality signals, and word embedding-based similarity to address the limitations empirically demonstrated by our rigorous evaluation methodology.