# Final Project - Part 2

Indexing and Evaluation

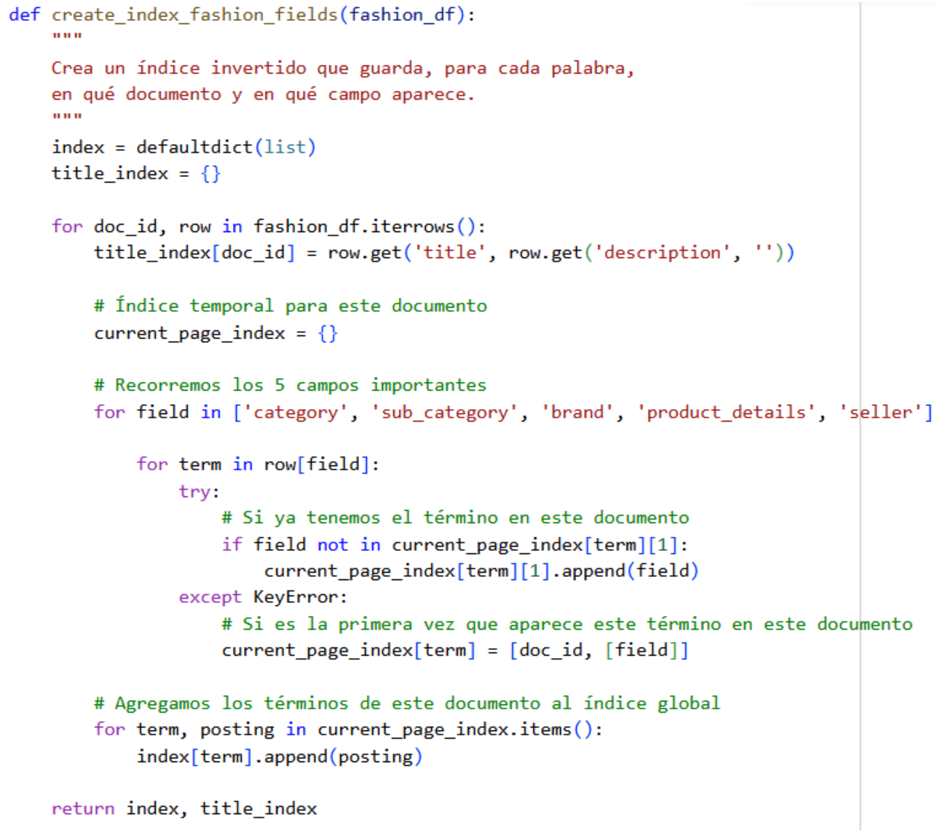
**GitHub:** https://github.com/juditvribe/IRWA_2025_G14_FinalProject.git
**TAG:** IRWA-2025-part-2

## PART 1: Indexing

### 1.1 Build inverted index

After preprocessing the dataset, the next step was to build an **inverted index**. The goal of this step is to construct an index that links each unique term from the fashion dataset to the *doc_id* in which it occurs, and the specific fields where the term appears (*e.g., category, brand, seller, …*).

In our data structure, the **main index** is implemented as a **defaultdict(list)**, where each key is a term and the value is a list of postings, each of which contain the doc_id and the list of fields where the term appears, as explained before.
A second dictionary **title_index** stores a mapping between each document's ID and its title (or description, if the title is missing). This mapping will be used to display readable search results.



```
def create_index_fashion_fields(fashion_df):
    """
    Crea un índice invertido que guarda, para cada palabra,
    en qué documento y en qué campo aparece.
    """
    index = defaultdict(list)
    title_index = {}

    for doc_id, row in fashion_df.iterrows():
        title_index[doc_id] = row.get('title', row.get('description', ''))

        # Índice temporal para este documento
        current_page_index = {}

        # Recorremos los 5 campos importantes
        for field in ['category', 'sub_category', 'brand', 'product_details', 'seller']

            for term in row[field]:
                try:
                    # Si ya tenemos el término en este documento
                    if field not in current_page_index[term][1]:
                        current_page_index[term][1].append(field)
                except KeyError:
                    # Si es la primera vez que aparece este término en este documento
                    current_page_index[term] = [doc_id, [field]]

        # Agregamos los términos de este documento al índice global
        for term, posting in current_page_index.items():
            index[term].append(posting)

    return index, title_index
```

*→ Iterate over all documents so that every record is processed individually.*
*→ current_page_index avoids duplicate entries before merging with the global index.*

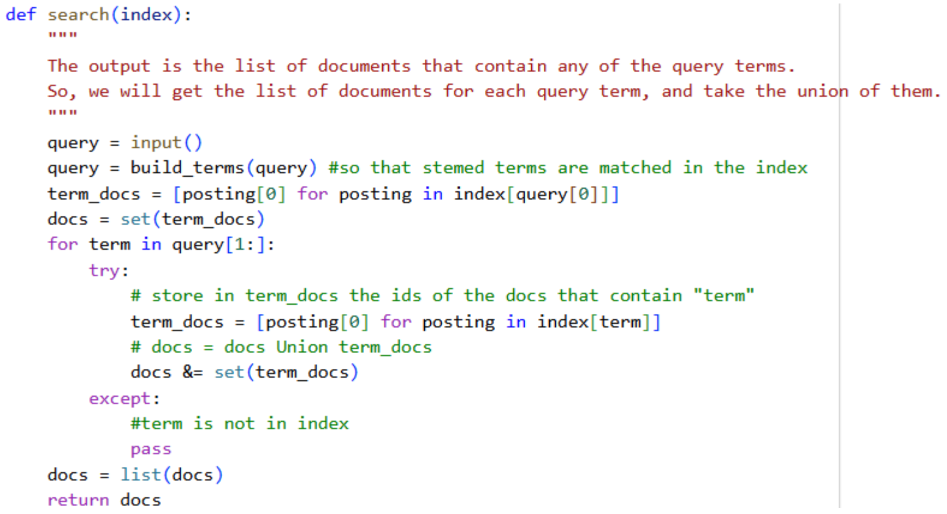
*→ Only semantically relevant fields are used.*

*→ If a term already exists, it appends the field name to its list.*
*Otherwise, it creates a new entry.*

*→ Merge with the global index.*

## 1.2 Propose test queries

Once the inverted index is built, the next step is to test and evaluate the retrieval system through several queries. The retrieval process for these queries is handled by the **search()** function, which implements **conjunctive (AND) query logic**. This means that only documents containing all query terms will be returned.



```
def search(index):
    """
    The output is the list of documents that contain any of the query terms.
    So, we will get the list of documents for each query term, and take the union of them.
    """
    query = input()
    query = build_terms(query) #so that stemed terms are matched in the index
    term_docs = [posting[0] for posting in index[query[0]]]
    docs = set(term_docs)
    for term in query[1:]:
        try:
            # store in term_docs the ids of the docs that contain "term"
            term_docs = [posting[0] for posting in index[term]]
            # docs = docs Union term_docs
            docs &= set(term_docs)
        except:
            #term is not in index
            pass
    docs = list(docs)
    return docs
```

→ *Query is entered by the user and processed by the* ***build_terms()*** *function.*
→ *Retrieve candidate docs for the first term.*
→ *Iterative intersection (AND logic)*

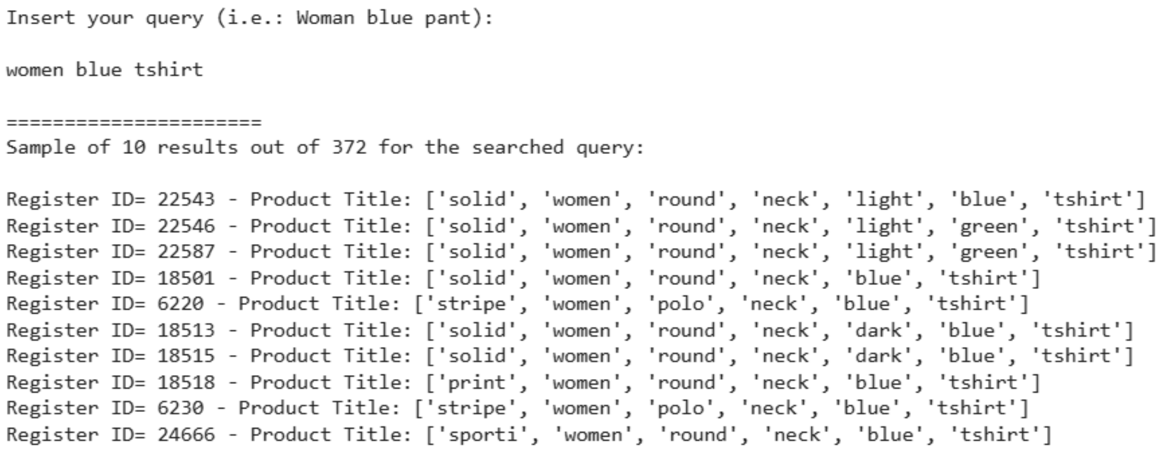
→ *Handling missing terms.*

→ *Return matching docs.*

Notice that both the dataset and the user query passed through the same preprocessing pipeline (*build_terms()*), which ensures same preprocessing (stemming, stopword removal, etc.) of the query and document terms.

Based on the exploratory results (Final Project - Part 1), the following five example queries were designed. Each query combines high-frequency terms that reflect user-like information needs.

1. Comfort Women Dark Blue T-Shirt
2. Green Stripe Men Cotton Polo
3. Solid Track Black Pants for Men
4. Print stylish pyjama
5. Cycling Clothes in Black



```
Insert your query (i.e.: Woman blue pant):

women blue tshirt

======================
Sample of 10 results out of 372 for the searched query:

Register ID= 22543 - Product Title: ['solid', 'women', 'round', 'neck', 'light', 'blue', 'tshirt']
Register ID= 22546 - Product Title: ['solid', 'women', 'round', 'neck', 'light', 'green', 'tshirt']
Register ID= 22587 - Product Title: ['solid', 'women', 'round', 'neck', 'light', 'green', 'tshirt']
Register ID= 18501 - Product Title: ['solid', 'women', 'round', 'neck', 'blue', 'tshirt']
Register ID= 6220 - Product Title: ['stripe', 'women', 'polo', 'neck', 'blue', 'tshirt']
Register ID= 18513 - Product Title: ['solid', 'women', 'round', 'neck', 'dark', 'blue', 'tshirt']
Register ID= 18515 - Product Title: ['solid', 'women', 'round', 'neck', 'dark', 'blue', 'tshirt']
Register ID= 18518 - Product Title: ['print', 'women', 'round', 'neck', 'blue', 'tshirt']
Register ID= 6230 - Product Title: ['stripe', 'women', 'polo', 'neck', 'blue', 'tshirt']
Register ID= 24666 - Product Title: ['sporti', 'women', 'round', 'neck', 'blue', 'tshirt']
```

**RESULTS:** Despite the strict conjunctive condition, the system retrieved 372 matching products, showing that the indexed collection contains sufficient lexical variation. The retrieved titles indicate that matches may come from different fields, for example, the color from “product_details”, product type from “category” or gender from “sub_category”.

At this stage, documents are not ordered by relevance. All results are considered equally valid matches.

### 1.3 Rank your results

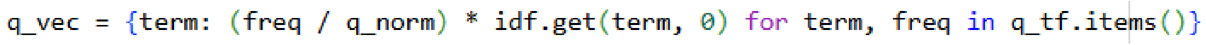
The next step is clearly to rank the retrieved results by relevance. In information retrieval, **TF-IDF algorithm** is a widely used metric that balances two opposing tendencies:

- TF (Term Frequency): A term is **more** relevant to a document if it appears frequently in it.
- IDF (Inverse Document Frequency): A term is **less** relevant if it appears in many documents across the corpus.

By combining both, TF-IDF algorithm gives higher scores to terms that are frequent within a document but rare across the whole collection.

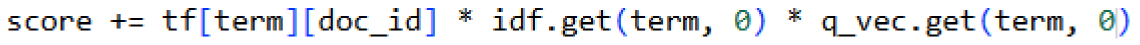
**create_index_tfidf()** function builds a TF-IDF based index from the preprocessed dataset. The variable index is a defaultdict(list) that stores all term occurrences across the dataset. Each term maps to a list of postings containing the document id and the positions of the term within that document. Then we compute the term frequency and normalize it by the document length to avoid bias toward longer documents that naturally contain more terms. The document frequency (DF) counts how many different documents contain a specific term. And finally, after iterating over all records, the inverse document frequency (IDF) is computed. The function then stores a title_index, mapping each document’s pid to its title. By combining TF, DF, and IDF information in separate dictionaries, the system can efficiently compute relevance scores for new queries.

**rank_documents()** uses the previously built statistics to rank candidate documents based on their TF-IDF similarity. A **query vector** is built by computing the TF-IDF value for each term in the query:



```
q_vec = {term: (freq / q_norm) * idf.get(term, 0) for term, freq in q_tf.items()}
```

where q_norm normalizes the query vector to unit length.
Then for every candidate document, the system calculates a **cosine similarity** score between the query and the document TF-IDF vectors:



```
score += tf[term][doc_id] * idf.get(term, 0) * q_vec.get(term, 0)
```

Each term’s weight contributes proportionally to how strongly it appears in both the query and the document. Finally documents are sorted in **descending order** of their scores.

**search_tf_idf()** is called for each query and integrates both the retrieval and ranking processes.

**RESULTS:** For the query *“Comfort Women Dark Blue T-Shirt”* the system returned the following top-10 results:

1. Solid Women Dark Blue Track Pants
2. Striped Men Dark Blue Track Pants
3. Printed Women Dark Blue Track Pants
4. Slim Women Dark Blue Jeans
5. Solid Men Dark Blue Track Pants
6. Solid Men Dark Blue Track Pants

7. Solid Women Dark Blue Track Pants
8. Skinny Women Dark Blue Jeans
9. Self Design Women Dark Blue Track Pants
10. Solid Women Dark Blue Track Pants

We observe that the system correctly identifies the color term “dark blue” and matches it across cproducts. However, The system fails to retrieve T-shirts, even though this is the most important semantic element of the query.

# PART 2: Evaluation

## 2.1 Evaluation Metrics

After implementing the indexing and ranking components, the next step was to evaluate how effectively the system retrieves relevant documents. Each metric operates on two inputs **y_true**. ground truth relevance labels (1 = relevant, 0 = not relevant), and **y_score**, predicted relevance scores from the TF-IDF ranking.

1. **Precision@K**: *“Of the top K retrieved documents, how many are actually relevant?”*
2. **Recall@K**: *“Of all relevant documents in the dataset, how many did the system retrieve in the top K?”*
3. **Average Precision@K**: *“How well are the relevant documents distributed among the top K results?”*
4. **F1-Score@K**: *“The harmonic mean of Precision and Recall.”*
5. **Mean Average Precision (MAP)**: *“Average of the Average Precision across all queries.”*
6. **Mean Reciprocal Rank (MRR)**: *“How high does the first relevant document appear?”*
7. **Normalized Discounted Cumulative Gain (NDCG)**: *“How well-ordered are the relevant results in the ranking?”*

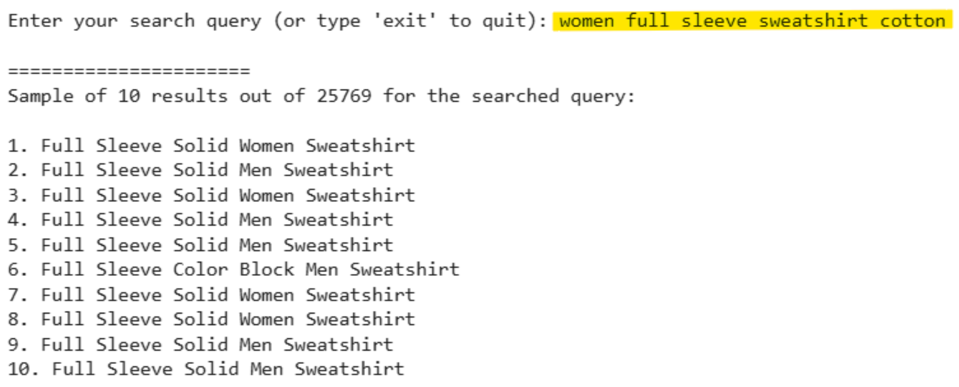
## 2.2 Evaluation on Validation Data

Two validation queries were defined in the provided dataset (validation_labels.csv):

Query 1: “women full sleeve sweatshirt cotton”
Query 2: “men slim jeans blue”

For each query we retrieved the top-ranked results using the search_tf_idf() function, the corresponding ground-truth labels extracted from validation_labels.csv and evaluation metrics that were computed at different K thresholds (4, 8, 12, 16, 20).

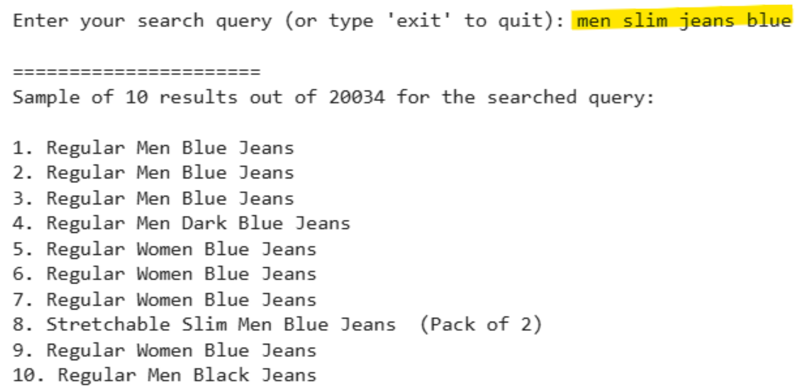
**RESULTS:**



```
Enter your search query (or type 'exit' to quit): women full sleeve sweatshirt cotton

======================
Sample of 10 results out of 25769 for the searched query:

1. Full Sleeve Solid Women Sweatshirt
2. Full Sleeve Solid Men Sweatshirt
3. Full Sleeve Solid Women Sweatshirt
4. Full Sleeve Solid Men Sweatshirt
5. Full Sleeve Solid Men Sweatshirt
6. Full Sleeve Color Block Men Sweatshirt
7. Full Sleeve Solid Women Sweatshirt
8. Full Sleeve Solid Women Sweatshirt
9. Full Sleeve Solid Men Sweatshirt
10. Full Sleeve Solid Men Sweatshirt
```



```
Enter your search query (or type 'exit' to quit): men slim jeans blue

======================
Sample of 10 results out of 20034 for the searched query:

1. Regular Men Blue Jeans
2. Regular Men Blue Jeans
3. Regular Men Blue Jeans
4. Regular Men Dark Blue Jeans
5. Regular Women Blue Jeans
6. Regular Women Blue Jeans
7. Regular Women Blue Jeans
8. Stretchable Slim Men Blue Jeans  (Pack of 2)
9. Regular Women Blue Jeans
10. Regular Men Black Jeans
```

When looking at the results of the evaluation metrics in the notebook, we notice that Precision@4 was high for Query 1 (0.75) but very low for Query 2 (0.25). This means that the top results for “women sweatshirt” were mostly relevant, while “men slim jeans” had more noise. As K increases, Recall@K steadily improves, reaching 1.0 for both queries at K = 20. This indicates that all relevant items eventually appear in the retrieved set, although not necessarily at the top.

MAP values rise from 0.444 at K = 4 to 0.568 at K = 20, showing that retrieval improves when more results are considered. However, the relatively modest MAP values (< 0.6) suggest that relevant documents are not consistently ranked near the top, confirming that the TF-IDF model lacks fine-grained ranking precision.

F1 increases with K (from approximately 0.35 to 0.78 for Query 1) showing that while early results may miss some relevant items, broader retrieval compensates by including them later. The harmonic mean emphasizes that both precision and recall need to be strong for a good balance, which occurs around K = 12–20.

NDCG follows a similar trend. It grows from 0.610 at K = 4 to 0.814 at K = 20 for Query 1. This demonstrates that relevant documents increasingly occupy higher positions in the ranked list. The lower NDCG values for Query 2 ( approximately 0.16 to 0.66) indicate poorer ranking quality for jeans-related searches.

The MRR remains constant at 0.375, meaning that on average, the first relevant document appears around position 3 in the ranking. This shows that while the system finds relevant items relatively early, it does not always rank them first.

The evaluation demonstrates that the system is functionally correct and retrieves relevant documents, ranking quality is moderate, and that the performance depends heavily on query phrasing.

## 2.3 Evaluation on our Queries

After validating the retrieval system on predefined queries, we conducted additional tests using five custom queries that reflect realistic user search behavior and product variety. Each query was evaluated using the same set of metrics, for larger K values, given the large size of the product corpus. Our queries were:

Query 1: “comfort women dark blue t-shirt”
Query 2: “green stripe men cotton polo”
Query 3: “solid track black pants for men”

Query 4: “print stylish pyjama”
Query 5: “cycling clothes in black”

**RESULTS:**
Precision remained 1.0 for nearly all queries and K values, indicating that every retrieved document was considered relevant according to the validation labels. Similarly, MAP and NDCG achieved a perfect score of 1.0, confirming that all relevant products were ranked at the top and the ranking order perfectly matched the ground truth.

Recall started very low (around 0.04 at K=1000) and increased steadily with higher K, reaching almost full recall at K=20000 for most queries. This pattern indicates that although relevant items were ranked highly, there were many relevant documents distributed deeper in the index, requiring a larger retrieval depth to be captured.

The F1-Score improved from about 0.08 at K=1000 to over 0.9 at K=20000. This consistent increase shows that the system balances recall and precision well as the retrieved set grows, with no drop in precision due to irrelevant items.

All NDCG and MRR values were 1.000, implying that the first retrieved document was always relevant, and the relevance scores decayed ideally. This reinforces that the ranking order was perfect for these test queries.

To sum up, we have observed that the current TF-IDF based search system shows strong lexical matching but several limitations. It relies purely on keyword overlap, leading to poor retrieval accuracy when users use synonyms or slightly different phrasing. Moreover, the ranking quality is limited because all fields contribute equally, even though titles and categories should be more influential than long descriptions. Additionally, it lacks semantic understanding, so context and intent are ignored. Query formulation is also simple, with no expansion or spelling correction, and the indexing strategy treats all text as a flat bag of words.

To improve performance, we suggest the following: integrating semantic models, applying field weighting in TF-IDF or BM25, implementing query expansion using synonyms and spell-checking, and refining the indexing structure to store field-level importance and phrase information.