Project Part 2 Report

Information Retrieval And Web Analysis 2025

Date: 3rd November 2025
Group Identifier: G_007
Group Member 1: Marc Bosch Manzano u215231
Group Member 2: Chris Matienzo Chilo u198726>
Group Member 3: Àlex Roger Moya u199765
This file was edited with Markdown language.

GitHub repository link

The GitHub repository with the code of the labs is the following: https://github.com/u215231/Information-Retrieval-and-Web-Analytics-Project.git. Here, you have to navigate to the folder where the Part 2 solution is located: `IRWA-2025-part-2` . The Python notebook with the solution's code is called `IRWA-2025-part-2-final-solution.ipynb` .

Introductory Section

For this Part 2 of the project, we have implemented several functions and classes in order to solve all the questions. We have structured them into some python source code files described below.

- **processing.py**: It contains functions to process the lines of text. In particular, it contains the `build_terms` function and the definitions of the stemmer object and stop words.
- **index.py**: It contains the class called *Indexing* that controls the data structure of the index. It is used in section `1.1` and `2.3` .
- **ranking.py**: It contains the *Ranking* class that is used to create the ranking of documents given the queries. There are several functions, including those that create the term frequency, the inverse document frequency, the weights for documents or queries, etc. The most relevant function is `rank_tfidf_dataframe` , which returns a dataframe with the queries, the top K documents with the best

scores, and the scores for each document and query. It is used in sections `1.3` and following.

- **evaluation.py**: It contains the *Evaluation* class to perform the analysis of the documents predicted as relevant for the ranking. Here are all the functions requested for sections `2.1`, `2.2` and `2.3`: precision at K, recall at K, average precision at K, etc.

All the explanations and the documentation of the classes and functions can be found on the respective files, since we have used python docstrings to comment the codes.

Part 1: Indexing

1.1. Build inverted index: After having pre-processed the data, you can then create the inverted index.

Previously, we have generated the processed version of the Fashion Products Dataset as CSV file. We have added a code to exported it at the bottom part of project 1 solution notebook (see: IRWA-2025-part-1-final-solution.ipynb).

Then, we have imported the processed dataframe as `df_fashion_products`. We focused on the columns `pid` and `document`, which contain the document identifiers and document texts. The column *document* is defined in the project part 1 notebook and contains the merged contents of the title, description, and categorical variables.

Here, we have applied the `build_inverted_index` function to these documents, and we printed the results with the `print_inverted_index` function (see: indexing.py). We then sent an example query to the index, "women cotton dress", using the `search_by_conjunctive_queries` function. It retrieved all the documents that match exactly the words of the query.

1.2. Propose test queries: Define five queries that will be used to evaluate your search engine.

Here, we have proposed five queries and we have arranged them as a Pandas series, identifying each query by an integer.

1.3. Rank your results: Implement the TF-IDF algorithm and provide ranking-based results.

Theoretical TF-IDF Model

Here, we provide the mathematical formulation of the TF-IDF model we have learned in theory lessons.

Let the following definitions of our inputs:

- $V = k_1, k_2, ..., k_t$ is the vocabulary, the set of terms, where t is the number of terms in the vocabulary.
- $d_1, d_2, ..., d_N$ is the documents of the collection, where N is the number of documents in the collection.
- $i \in \{1, 2, ..., t\}$ the index for a term in the vocabulary.
- $j \in \{1, 2, ..., N\}$ the index for a document in the collection.
- q is a query.

Let the following definitions of the metrics of the vector model:

- $f_{i,j}$ be the frequency of occurrence of term i in document j.
- df_i be the document frequency for term i, that is, the number of documents where term i occurs, defined as $df_i = |\{j : f_{i,j} > 0\}|$.
- $tf_{i,j}$ be the term frequency for term i in document j, defined as

$$tf_{i,j} = 1 + log_2(f_{i,j})$$

- idf_i be the inverse document frequency defined for term i, defined as

$$idf_i = log \frac{N}{df_i}$$

- Let $w_{i,j}$ be the term weights or TF-IDF weights for term i and document j, defined as

$$w_{i,j} = \begin{cases} tf_{i,j} \cdot idf_i & \text{if } f_{i,j} > 0, \\ 0 & \text{otherwise} \end{cases}$$

Similarly, we define the frequency $f_{i,q}$, term frequency $t_{i,q}$, and weights $w_{i,q}$ of term i for query q.

Let $\overrightarrow{d_j} = [w_{1,j}, w_{2,j}, ..., w_{t,j}]^T$ be the weighted vector for document j
Let $\overrightarrow{q} = [w_{1,q}, w_{1,q}, ..., w_{t,q}]^T$ be the weighted vector for query q.

We define the similarity between document j and query q as the cosine for the angle between vectors $\overrightarrow{d_j}$ and $\overrightarrow{q}$:

$$\text{sim}(\overrightarrow{d_j}, \overrightarrow{q}) = \frac{\langle \overrightarrow{d_j}, \overrightarrow{q} \rangle}{\left\| \overrightarrow{d_j} \right\| \cdot \left\| \overrightarrow{q} \right\|}$$

where $\langle \cdot, \cdot \rangle$ is the inner product of two vectors and $\| \cdot \|$ is the Euclidean norm of a vector, both in space $\mathbb{R}^t$. This formula is also called the cosine similarity between vectors.

We define our ranking for our query q as the first top K documents retrieved by ordering our similarities in a descending way. If our sorted similarities are

$$\text{sim}(\overrightarrow{d}_{(1)}, \overrightarrow{q}) \geq \text{sim}(\overrightarrow{d}_{(2)}, \overrightarrow{q}) \geq ... \geq \text{sim}(\overrightarrow{d}_{(K)}, \overrightarrow{q}) \geq ... \geq \text{sim}(\overrightarrow{d}_{(N)}, \overrightarrow{q})$$

Therefore, our retrieved documents are those in the $K \leq N$ ordered greater scores, that is, the tuple of document indices $(j : j = (r))_{r=1}^{K}$, where $(\cdot)$ is a function that assigns a rank number $r \in 1, ..., N$ to the real document index number $j \in 1, ..., N$.

Implemented TF-IDF Model

Our implementation has consisted of a class called *Ranking* (see: *ranking.py* file.). It uses dictionaries in order to optimize the memory consumption and make more efficient the computations.

The class on its constructor receives two series: a documents' series, with identifier and document text content, and a queries' series, with identifier and query text content, too. This makes the ranking process more generalized: it can manage several queries with the same collection of documents. We have furthermore two setters, *set_documents* and *set_queries*, whether the user wants to change the collection of the ranking object.

The framework is managed by several methods and attributes document. Each method computes one of the metrics (frequencies, weights, etc.), and it is able to return it as well as modify the value of the corresponding attribute. For example, the *Ranking.get_df* function returns a dictionary of document frequencies of the terms, but sets the value to the attribute *self.df*, which stores the document frequencies. Each of

these methods, when it has already computed one of these metrics and both the documents and the queries have not changed, directly returns the value of the corresponding attribute. This optimizes the execution time of document ranking calculations. In order to recalculate metrics, new documents or queries must be entered into the setters, otherwise they will not be recalculated.

The attributes are mainly dictionaries or compositions of dictionaries, as we have said, in order to save memory and improve the speed of calculation. For example, in the case of occurrence frequencies *f*, if they are computed for all the documents in the collection, they form a dictionary with the document identifier as key and, as value, another dictionary with the term and its frequency as another.

The calculation of the similarity scores for documents as queries is done in *rank_tfidf_dict*. Here, we retrieve a dictionary of the top K scored documents for each query. It has as keys the query identifier, and as values a dictionary of documents identifier and each score, respectively. The similarity it is computed by the terms of the query that appear on the document. This helps optimize the calculation of similarities, avoiding having to do the dot products of vectors with as many dimensions as the terms in the vocabulary. So, the formula we have used for the calculations of the scores can be stated as following:

$$\text{sim}(d_j, q) = \frac{\sum_{i:k_i \in d_j \cap q} w_{i,j} w_{i,q}}{\sqrt{\sum_{i:k_i \in d_j} w_{i,j}^2} \sqrt{\sum_{i:k_i \in q} w_{i,q}^2}}$$

We have not eliminated the normalization of scores by query length, as we will evaluate the scores in the next sections. We need all scores in the range $[0, 1]$ in order to facilitate the analysis.

We have also defined *rank_tfidf_dataframe*, which converts the scores to a dataframe of query identifiers, document identifiers, and scores. This method is used in the evaluation section to compute the ranked documents for the *validation_labels.csv* file.

Finally, we have a *print_rankings* method to visualize the ranked document for each query one by one. We have other static methods an auxiliary methods in the class that are self explained in the *ranking.py* file.

Object methods of *Ranking* class.

```
def _set_items(items, type]) -> None:
def set_documents(documents) -> None:
def set_queries(queries) -> None:
def get_df() -> dict:
def get_idf() -> dict:
def get_f(type) -> dict:
def get_tf(type) -> dict:
def get_tfidf(type) -> dict:
def rank_tfidf_dict(topK) -> dict:
def rank_tfidf_dataframe(topK) -> pd.DataFrame:
def print_rankings(topK) -> None:
```

Static methods or *Ranking* class.

```
def get_frequencies_from_text(text) -> dict:
def get_tf_from_frequency(frequencies) -> dict:
def get_weights_from_tf_idf(tf, idf) -> dict:
def get_cosine_similarity(document_vector, query_vector) -> float:
def get_norm(vector) -> float:
def get_top_scores(document_scores, topK) -> dict:
```

Results

We can see the ranked documents we have obtained for our five queries are quite relevant, as they contain many of the words in the queries most of the time.

Part 2: Evaluation

2.1 Implement the following evaluation metrics to assess the effectiveness of your retrieval solutions. These metrics will help you measure how well your system retrieves relevant documents for each query:

- **i. Precision@K (P@K)**
- **ii. Recall@K (R@K)**
- **iii. Average Precision@K (P@K)**
- **iv. F1-Score@K**
- **v. Mean Average Precision (MAP)**
- **vi. Mean Reciprocal Rank (MRR)**
- **vii. Normalized Discounted Cumulative Gain (NDCG)**

Implementation of Evaluation Metrics

Our implementation of these metrics has been carried out in the class *Evaluation* (see: *evaluation.py* file.). This source code file groups all of these metrics.

The class constructor recieves a *search_results*, which is a Pandas dataframe containing a query identifier (a query category, which in our case is 1 or 2 for validations.csv), the labels of the documents (true or false), and the scores of the documents (a nonnegative value). The search results dataframe is stored on the evaluation object. We have also other attributes of the class, which are obtained from search results and the chosen *query_id* of the user: *labels*, the values of the column of labels, and *scores*, the values of the column of scores. We have an optional attribute called *query_text*, which is can contain the text associated to a query.

The methods for the evaluation metrics have been mainly taken from the professor's Evaluation_prof.ipynb, and have been slightly modified and adapted to be object methods. Furthermore, we have created a print function for all of these evaluations.

Object methods of *Evaluation* class.

```
def _select_labels(query_id):
def _select_scores(query_id):
def precision_at_k(k) -> float:
def recall_at_k(k) -> float:
def avg_precision_at_k(k) -> float:
def f1_score_at_k(k) -> float:
def map_at_k(k) -> float:
def rr_at_k(k) -> float:
def mrr_at_k(k) -> float:
def dcg_at_k(labels, scores, k) -> float:
def ndcg_at_k(k) -> float:
def print_evaluation(k) -> float:
```

Commentary on Evaluation Metrics

- `Precision@K (P@K)` : This metric (from precision_at_k) measures accuracy. It measures how many documents in the top K were actually relevant.
- `Recall@K (R@K)` : This metric (recall_at_k) measures completeness. It measures all the possible relevant documents in the entire collection, how many did I find in the top K.
- `F1-Score@K` : This is the harmonic mean of Precision and Recall (f1_score_at_k). It provides a single, balanced score, which is useful when both false positives (low precision) and false negatives (low recall) are equally bad.
- `Average Precision (AP) / MAP` : AP (avg_precision_at_k) is a powerful, rank-sensitive metric for a single query. It rewards systems that not only find relevant documents but also rank them higher. A relevant item at rank 1 is weighted much more heavily than one at rank 10. MAP (Mean Average Precision) is just the average of these AP scores across all my queries.
- `Reciprocal Rank (RR) / MRR` : This metric (rr_at_k) is very simple: it measures the rank of the first relevant document found. If the first relevant item is at rank 3, the RR is 1/3. MRR is the average of these scores. It's excellent for "known-item" searches where the user just wants one correct answer.
- `Normalized Discounted Cumulative Gain (nDCG)` : This (ndcg_at_k) is the most robust, rank-sensitive metric. It assumes relevant items are more useful at the top (the "discount"). It is "normalized" by comparing my system's score to the score of a perfect ranking. This makes it the industry standard for evaluating ranking quality.

2.2 Apply the evaluation metrics you have implemented to the search results and relevance judgments provided in validation_labels.csv for the predefined queries. When reporting evaluation results, provide only numeric values, rounded to three decimal places. Do not include textual explanations or additional statistics in this section.

- **a. Query 1: women full sleeve sweatshirt cotton**
- **b. Query 2: men slim jeans blue**

In this section, in the notebook, we have imported the Validation Labels file as a dataframe `df_validation_labels` . We have applied to its queries the *build_terms* function. Then, we have created the dataframe `df_predicted_labels` using the documents from the Fashion Products Dataset file, the queries from the Validation Labels file, and the *rank_tfidf_dataframe* method. Next, we have merged both dataframes to a single one called `df_validation_predicted_labels` , wich contains query identifiers, query texts, labels, and scores. Finally, we have printed an evaluation for each category of query.

2.3 You will act as expert judges by establishing the ground truth for each document and query.

- **2.3.a. For the test queries you defined in Part 1, Step 2 during indexing, assign a binary relevance label to each document: 1 if the document is relevant to the query, or 0 if it is not.**
- **2.3.b. Comment on each of the evaluation metrics, stating how they differ, and which information gives each of them. Analyze your results.**

Based on the evaluation results:

Good Performance:

- Queries 3, 4, and 5 show strong performance with perfect recall (1.000), meaning all relevant documents were retrieved in the top 10 results
- Query 5 performed exceptionally well with perfect precision, average precision, and nDCG

Poor Performance:

- Queries 6 and 7 failed completely (all metrics = 0.000), indicating no relevant documents were found

- Recall is NaN for queries 6 and 7 because there were no relevant documents in the entire collection for these queries

Key Insights:

- The system handles clothing-related queries well but struggles with accessories (bags, shoes)
- There may be vocabulary mismatch or insufficient product coverage for bags and shoes
- The ranking effectively places relevant documents at the top for successful queries (high nDCG and reciprocal rank)

- **2.3.c. Analyze the current search system and identify its main problems or limitations. For each issue you find, propose possible ways to resolve it. Consider aspects such as retrieval accuracy, ranking quality, handling of different field types, query formulation, and indexing strategies.**

The current system shows promising potential, achieving a MAP@5 of 0.513, though there's still room for improvement in consistency across queries. The analysis highlights three clear opportunities to strengthen the model and achieve higher performance:

- Conjunctive Query Handling: The strict "AND" query (*search_by_conjunctive_queries* in indexing.py) currently limits retrieval for multi-term queries such as "adidas jacket" and "discount sport shoes." By switching to a more flexible disjunctive "OR" query, the system can significantly broaden its search results and improve recall.
- Term-Frequency Issue: A small adjustment in processing.py — removing the line `line = list(set(line))` — will restore proper TF-IDF calculations. This simple fix will ensure that term frequency is accurately represented, greatly enhancing ranking quality.
- Field Weighting Enhancement: The model currently treats all fields equally, blending product_name, brand, and description. Introducing field-based weighting will allow stronger matches in the product_name to take priority, delivering more relevant and precise results.

Overall, these insights reveal straightforward, high-impact improvements that can elevate the system's performance and reliability.