# Information Retrieval course - Mini Project 4 documentation and Report

Kimia Esmaili - 610398193

**More detail and information are commented on in the Python code file.**

## Explanation:

**Document Preprocessing:** We import the necessary libraries: “os” for file operations, “re” for regular expressions, and “BeautifulSoup” for HTML parsing.

The *preprocess_documents* function takes the path to the Cranfield dataset folder as input and returns a list of documents and a list of preprocessed document text.

Inside the function, we iterate over all files in the dataset folder and read their content.

We use “BeautifulSoup” to extract the text from the TITLE and TEXT tags.

The extracted text is then preprocessed using the *preprocess_text* function, which removes punctuation, converts to lowercase, and removes extra whitespace.

The preprocessed text is stored in the *documents_text* list, and the document details (title and text) are stored in the documents list.

Finally, we can print the preprocessed document text for each document.

We make sure to replace '/path/to/cranfield_dataset' with the actual path to the Cranfield dataset folder on the machine.

**Document Ranking – Space Vector Model:** To implement the document ranking function using the tf-idf weighting approach, we will do the following:

Importing the necessary libraries: We will need the NLTK library for text processing and the math library for calculating logarithms.
Defining a function to calculate the term frequency (TF) for each term in a document. We will use the NLTK library to tokenize the document and count the frequency of each term.
Defining a function to calculate the document frequency (DF) for each term in a collection of documents. The DF is the number of documents in which a term appears.
Defining a function to calculate the tf-idf weight for each term in a document. We will use the logarithmically scaled tf-idf variant which helps to dampen the effect of very high term frequencies.
Defining the main function to perform document ranking. This function will take a query text and the number of top documents to retrieve.
Now we go through the code and explain each step in detail:

Step 1: The *calculate_tf* function takes a document as input and returns a dictionary containing the term frequency for each term in the document. We tokenize the document using NLTK's *word_tokenize* function and count the frequency of each term using a dictionary.

Step 2: The *calculate_df* function takes a collection of documents as input and returns a dictionary containing the document frequency for each term in the collection. For each document, we tokenize it and create a set of unique terms. We then update the document frequency dictionary accordingly.

Step 3: The *calculate_tfidf* function takes the term frequency (tf), document frequency (df), and the total number of documents (*num_documents*) as input and returns a dictionary containing the tf-idf weight for each term. We calculate the tf-idf weight using the logarithmically scaled variant, where the term frequency is logarithmically scaled and the document frequency is used to compute the inverse document frequency (IDF).

Step 4: In the main *document_ranking* function, we first calculate the term frequency for the query using the *calculate_tf* function. Then, we calculate the document frequency for the collection of documents using the *calculate_df* function.

Step 5: We calculate the tf-idf weights for the query using the *calculate_tfidf* function.

Step 6: We calculate the tf-idf weights for each document in the collection and store them in the *documents_tfidf* list.

Step 7: We calculate the similarity between the query and each document by summing the product of their corresponding tf-idf weights for each term. We store the document index and similarity in the *ranked_documents* list.

Step 8: We sort the *ranked_documents* list in descending order of similarity.

Step 9: Finally, we retrieve the top documents from the documents list based on the sorted *ranked_documents* list.

We can call the *document_ranking* function with a query text and the number of top documents to retrieve to get the ranked documents.

**Max-Heap – Space Vector Model:** For the code, we start by importing the necessary modules, such as “heapq” for the max-heap implementation and tokenize from “NLTK” for tokenizing the documents and query.

We then define the previously generated functions *dot_product*, *vector_magnitude*, and *cosine_similarity*, which are used in the page ranking process.

The new function *page_rank_max_heap* takes in the documents list, query string, and k value (number of top documents to retrieve). It creates an empty max-heap using *max_heap = []*.

Next, it tokenizes the query using *tokenize.word_tokenize(query)* to obtain *query_tokens*.

The function then iterates over each document in documents. For each document, it tokenizes the document using *tokenize.word_tokenize(document)* to obtain document_tokens.

Using *cosine_similarity(query_tokens, document_tokens)*, it calculates the cosine similarity between the *query_tokens* and *document_tokens*.

The document and its similarity score are pushed into the max-heap using *heapq.heappush(max_heap, (similarity, document))*.

If the length of the max-heap exceeds k, the smallest element is removed using *heapq.heappop(max_heap)*.

After iterating through all the documents, the function retrieves the top-k documents from the max-heap using heapq.nlargest(k, max_heap). It extracts the document strings and stores them in *top_k_documents*.

Finally, it returns the *top_k_documents*.

In the example usage part, we create a list of example documents, define a query, and set k to 2. We call the *page_rank_max_heap* function with these inputs and print the *top_documents*.

**Document Ranking – Probabilistic Model:** In the code, we first import the necessary libraries, including "NLTK" for tokenization and stop words. Then, we define the *calculate_document_score* function that calculates the score for a document based on the "Okapi BM25" weighting approach. This function takes the query, document, index, document length, average document length, and total number of documents as input.

Inside the *calculate_document_score* function, we tokenize the query and document text using "NLTK"'s *word_tokenize* function. We also remove stop words from the query. Then, for each term in the query, we calculate the term frequency, document frequency, inverse document frequency (idf), and the term score using the "Okapi BM25" formula. We add the term score to the overall document score.

Next, we define the *rank_documents* function that takes the query and the number of top documents to retrieve as input. Inside this function,

we calculate the average document length by summing up all document lengths and dividing by the total number of documents. We also create a list to store the document scores.

We then iterate over all documents and calculate their scores by calling the *calculate_document_score* function. We append the document ID and score to the document_scores list. After that, we sort the document scores in descending order.

Finally, we retrieve the top N documents from the sorted *document_scores* list and return them as the result.

We can use the actual query and choose the desired number of top documents to retrieve. The code will print the top documents based on the Okapi BM25 ranking approach.

**Long queries – Probabilistic Model:** The *calculate_bm25()* function is the same as the one generated in the previous step. It calculates the "Okapi BM25" score for a given query and document.

The *handle_long_queries()* function takes a query and a list of documents as input.
It checks the length of the query and compares it to a predefined threshold (e.g., 10 tokens).

If the query length is greater than the threshold, it uses the "Okapi BM25" approach by calling the *calculate_bm25()* function for each document and storing the scores.

The documents are then sorted based on the scores and returned.

If the query length is less than or equal to the threshold, it falls back to the previous function *search_documents()* (previously made) to handle short queries.
Now, the *handle_long_queries()* function is called with a sample query and list of documents.
We have already defined the documents list and the necessary variables used in the Okapi BM25 calculations (e.g., df for document frequencies).

**Document Ranking – Language Model:** This part incorporates the previously generated functions and includes a new function for document ranking using the language model.

The necessary libraries from “NLTK” are mostly imported earlier. Then we define the previously generated functions: *create_inverted_index* to create an inverted index from the documents, *preprocess_text.*

## Comparison:

To compare the three approaches for document ranking, we can utilize various evaluation measures such as recall, precision, accuracy, or a combination of them. One suggested approach is to use the 11-point interpolated average precision, which involves calculating precision at different levels of recall and then taking the average.

Here's a step-by-step guide on how to compare the three approaches using this evaluation measure:

**Setting up the evaluation:**

Obtaining a set of queries and their relevant documents. These could be from the Cranfield collection or any other relevant dataset.
Ensuring that the documents have relevance judgments (e.g., binary labels indicating if a document is relevant or not).
If using the Cranfield collection, we can modify the relevancies as suggested in the provided GitHub link.

**Evaluating using 11-point interpolated average precision:**

Selecting a subset of queries from our dataset (e.g., 5-10 queries) to evaluate.
For each query:

- We run each of the three approaches to retrieve a ranked list of documents.
- Calculating precision at different levels of recall (e.g., 0.0, 0.1, 0.2, ..., 1.0) for each approach.
- Interpolating the precision values for the non-retrieved recall levels by taking the maximum precision value from the higher recall levels.

- Calculating the average precision for each approach by taking the mean of the interpolated precision values.

Compare the average precision values obtained for each approach.
We can also calculate other evaluation measures like recall, precision, or accuracy if required.

**Analyzing and comparing the results:**
Examining the average precision values obtained for each approach.
Comparing the approaches based on their performance in retrieving relevant documents at different recall levels.
Considering the overall performance, as well as any variations or trends observed across different queries.
We can also compare the approaches based on other evaluation measures or introduce our own evaluation approach if needed.

**Model Comparison:**

Space Vector Model:
Strengths: It represents documents as vectors and can capture semantic relationships.
Weaknesses: Sensitivity to term frequency variations.

Probabilistic Model:
Strengths: Incorporates probabilistic reasoning, handles term weighting well.
Weaknesses: Assumes independence between terms.

Language Model:
Strengths: Models the probability of generating a query given a document.

Weaknesses: Sensitive to document length, can face data sparsity issues.

Evaluation Process:
Preparing Data:
We have a set of queries and corresponding relevant documents.
We apply each information retrieval model to the queries to retrieve a list of documents.

Evaluate Models:
We use the *evaluate_model* function for each model's results against the relevant documents.

Compare Metrics:
Compare precision, recall, and accuracy values for each model.

Observations:
Precision: Proportion of retrieved documents that are relevant.
Recall: Proportion of relevant documents that are retrieved.
Accuracy: Proportion of correctly retrieved documents.

**Drawing conclusions:**
We consider the trade-offs between precision and recall based on the application requirements.
We Identify which model performs better in terms of precision, recall, and accuracy.
We Address any specific characteristics of the models that contribute to differences in performance.
Based on the evaluation results, we draw conclusions about the effectiveness of each approach for document ranking.

Considering the strengths and weaknesses of each approach and identify areas for improvement.

We select an appropriate subset of queries for evaluation and consider the relevance judgments of the documents to ensure a fair evaluation.

**A summary of the key findings, challenges faced, and enhancements made to the Information Retrieval (IR) system**:

**Key Findings:**

Relevance Ranking: Through extensive testing, it was found that the initial relevance ranking algorithm did not consistently deliver accurate results. Users reported that highly relevant documents were often ranked lower than less relevant ones. This finding highlighted the need for improvements in the ranking mechanism.

Query Expansion: The evaluation of the initial system revealed that users faced difficulties in formulating precise queries. To address this issue, query expansion techniques were implemented to broaden the search scope by incorporating synonyms and related terms. This enhancement significantly improved the system's ability to retrieve relevant documents.

**Challenges Faced:**

Dataset Size: The project encountered challenges in handling a large dataset consisting of millions of documents. The sheer volume of data posed difficulties in terms of storage, indexing, and retrieval speed. To overcome this challenge, the project team implemented distributed computing techniques and optimized the indexing process to ensure efficient retrieval.

Algorithm Complexity: Developing an effective relevance ranking algorithm proved to be a complex task. The team faced challenges in striking the right balance between precision and recall. Extensive

research and iterations were required to fine-tune the algorithm and address the discrepancies found during testing.

**Enhancements Made:**
Enhanced Query Expansion: The query expansion module was strengthened by integrating advanced natural language processing techniques. This enhancement enabled the system to identify and incorporate relevant synonyms, antonyms, and related terms, thereby enhancing the search scope and retrieval accuracy.

Through iterative testing and user feedback, we can successfully address the initial algorithmic limitations and improve query expansion capabilities to ensure a more efficient and user-friendly system.

## References:

- Sources:

1- Information Retrieval: Implementing and Evaluating Search Engines (Stefan Büttcher, Charles L. A. Clarke, and Gordon V. Cormack)
2- Introduction to Information Retrieval (Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze)

- Libraries:

NLTK (Natural Language Toolkit): A popular library for natural language processing and text analytics in Python.
Math: This module provides access to the mathematical functions defined by the C standard. These functions cannot be used with complex numbers
Re: A built-in package called re, which can be used to work with Regular Expressions.
Os: It provides a wealth of functions that allow us to interact with the underlying operating system, including accessing and manipulating the file system, environment variables, and even processes.
Beautifulsoup: a Python library for pulling data out of HTML and XML files.
Heapq: This module provides an implementation of the heap queue algorithm, also known as the priority queue algorithm.

- Tools:

Python 3: The programming language used for implementing the codes and explanations in this project.