**Project: Design and implement a basic search engine for a collection of text**

**documents.**

1. **Data Collection: (1%)**

My data collection: <https://huggingface.co/datasets/BeIR/scidocs-qrels> At the end in a table of datasets I choose “TREC-COVID” It contains a set of text documents to serve as a corpus and has its queries and test files as a zip file to be downloaded.

2. **Preprocessing: (1%)**

For preprocessing my data I apply a function to do tokenization that splits documents into individual tokens, lowercasing that converts all text to lowercase for case insensitivity that makes the word back to the normal case, stopword removal that eliminates common words like "and", "the", "is" that do not contribute much to the meaning of the document, and stemming or lemmatization that reduces words to their base or root form like "running" to "run".

3. **Indexing: (1%)**

In the indexing phase, we build an inverted index, a critical structure for efficient document retrieval. This index maps each unique word to the documents containing it, along with the frequency of occurrence. By organizing the corpus in this way we calculate the term frequency (TF) for each term in a collection of processed documents, also iterate over the lexicon of the index, and print out each term along with its associated statistics, we enable fast and accurate search queries, enhancing the search engine's efficiency and user experience.

4. **Query Processing: (2%)**

On the corpus data preprocessing steps (tokenization, lowercase, etc.) as used during indexing, also retrieve information about the term "review"(ex) from an index using PyTerrier. It loads the index and accesses its components: the meta index, inverted index, and lexicon. Then, it retrieves the lexicon entry for the term "review" (ex) and extracts details such as its frequency and postings. If the term exists, it retrieves postings and iterates through them to calculate the total document length for the term. Finally, it prints out this total document length, offering insights into the term's distribution across the indexed documents and printed as a document ID then its frequency in this document. Also, a model is set up using the TF-IDF weighting, which assigns weights to terms based on their frequency in a document and inverse frequency across the entire collection, the search prioritizes terms that are frequent in the query but rare in the overall collection, aiming to retrieve documents that are most relevant to the query. Additionally, the model is configured to return a maximum of 10 documents as search results for each query, ensuring concise and focused retrieval outcomes.

5. **Query expansion: (3%)**

Apply relevant feedback by analyzing top-ranked documents for the initial

Queries by demonstrating query expansion using the RM3 model within PyTerrier. Initially, the document collection is indexed using the BM25 retrieval model. The query "name" (ex) is preprocessed and then used to retrieve search results with BM25. Subsequently, the RM3 model is employed to expand the query, incorporating additional terms based on their relevance to the initial query and the retrieved documents. The expanded query is then utilized to retrieve search results again using BM25 and in the end the code compares the search results before and after query expansion, showing the potential improvement in retrieval effectiveness. Also, it retrieves and displays the text of the top 5 tweets corresponding to the search results, providing insight into the relevance and context of the retrieved documents. It begins by tokenizing input text and creating a dictionary of documents. it trains a Word2Vec model to generate word embeddings, enabling semantic understanding. Query expansion is then demonstrated using the trained model to find similar terms. Additionally, the code employs ELMo to generate contextualized word embeddings for specific words within example sentences. Overall, it showcases a versatile approach to NLP tasks, integrating tokenization, word embedding training, query expansion, and contextualized embedding generation using popular libraries like NLTK, Gensim, TensorFlow, and TensorFlow Hub.

6. **User Interface: (1%)**

The UI that was used in this application serves as a basic search engine UI with an input field and a search button. The UI is designed using HTML and CSS, providing a simple and clean layout. The input field allows users to enter their search queries, while the search button triggers the search functionality. Upon clicking the search button, a JavaScript function is invoked to fetch search results asynchronously from the server. The results are then displayed dynamically on the web page without the need for page reloads, offering a seamless user experience. The UI's color scheme and styling are carefully chosen to ensure readability and aesthetic appeal, with a dark header contrasting against a light background and interactive elements styled for visual feedback on hover and focus.

7. **Evaluation: (1%)**

The evaluation code segment performs an evaluation of a retrieval system using the Vaswani dataset in PyTerrier. Initially, the dataset's topics are retrieved and stored in a data frame. The 'query' column is renamed to 'Text', and the index is assigned to the 'docno' column. Similarly, relevance judgments (qrels) are obtained from the dataset and processed to ensure consistency in data types. The retrieval system is set up using TF-IDF weighting. A search is conducted for the query term "mathematical" using the retrieval system. Finally, the search results are evaluated against the relevance judgments using the Evaluate function, providing comprehensive insights into the retrieval system's performance, including metrics such as Precision, Recall, and F1-score.