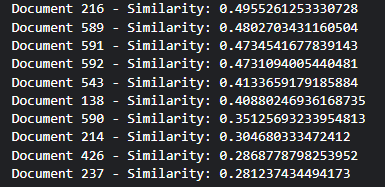
Information Retrieval

Building a Mini Search Engine using the Vector Space Model

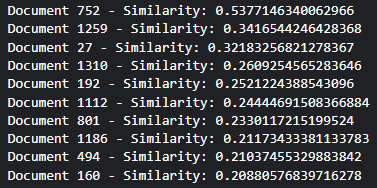
SARA Alsanajleh 163418 Ghada Abu Shaqra 164188

**Q1 : Presents at least 5 test cases for your search engine.**

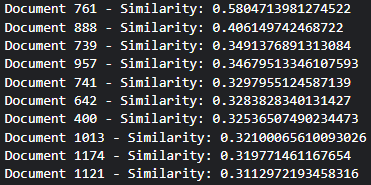
1. Testing query: has anyone explained the kink in the surge line of a multi-stage axial compressor .



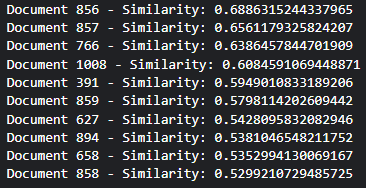
1. Testing query: to find an approximate correction for thickness in slender thin-wing theory .



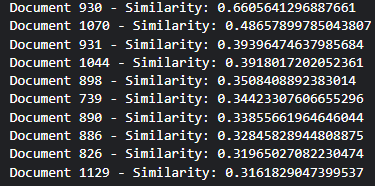
1. Testing query: is the information on the buckling of sandwich sphere available .



1. Testing query: have the effects of an elastic edge restraint been considered in previous papers on panel flutter .



1. Testing query: have non-linear large deflection analyses been conducted for shell shapes other than conical .



**Q2 : Documentation for Our code.**

* Part 1 : (PreProcessing)

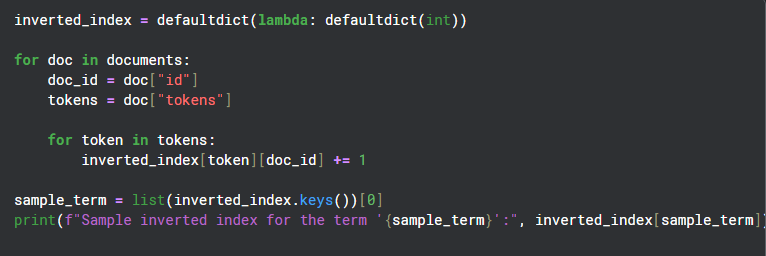
****

**Explanation:**

* stop\_words: a set containing common English stop words that usually don’t add meaning, such as "the", "is", "and".
* PorterStemmer: a tool to stem words by reducing them to their root (e.g., "running" becomes "run", "connected" becomes "connect").
* preprocess function steps:
  1. Convert the entire text to lowercase for consistency.
  2. Tokenize the text into individual words.
  3. Filter out any tokens that are not purely alphabetic (removing punctuation and numbers).
  4. Remove stop words that add little semantic value.
  5. Stem the tokens to their root form to unify similar words.
* The preprocessing function is applied to each document’s text in the documents list, and the resulting tokens are saved in a new field "tokens".

**Python Libraries Used:**

* nltk.corpus.stopwords
* nltk.stem.PorterStemmer
* nltk.tokenize.word\_tokenize
* Part 2 : (Indexing)

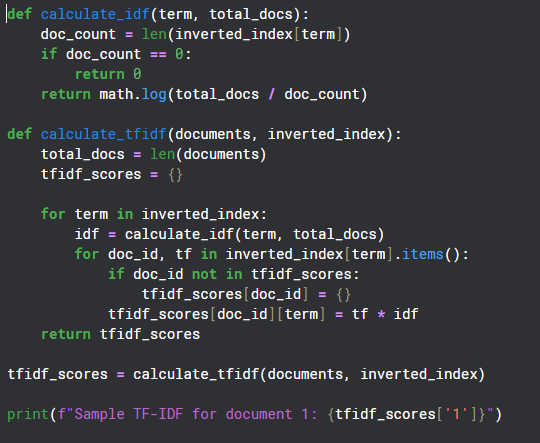
****

**Explanation**

* We build an **inverted index** data structure that links each token (word) to the documents containing it, along with the frequency of that token in each document.
* The inverted\_index is implemented as a nested dictionary:
  + The **keys** of the outer dictionary are the tokens (words).
  + The **values** are inner dictionaries where:
    - The keys are doc\_ids (unique identifiers of documents).
    - The values are the count of how many times the token appears in the corresponding document.
* The code processes each document in the dataset by:
  + Extracting the document’s unique doc\_id.
  + Retrieving the list of preprocessed tokens (tokens) for that document.
  + For each token, incrementing its frequency count in the inverted index under the respective doc\_id.
* Finally, the code prints a sample term from the inverted index with its document frequency distribution to verify the index was built correctly.

**Python Libraries Used**

* 1. collections.defaultdict
* Part 3 : (Weighting)

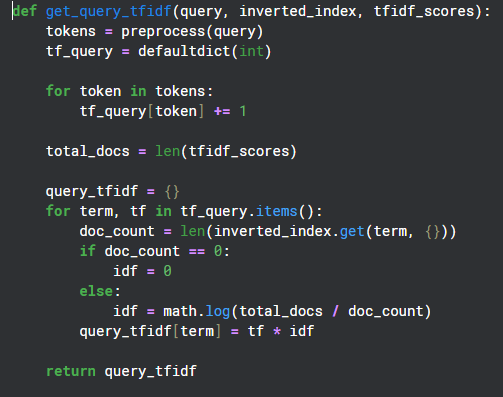
****

**Explanation:**

* The function calculate\_idf computes the **Inverse Document Frequency (IDF)** for a given term:
  + It counts in how many documents the term appears (doc\_count).
  + Returns 0 if the term doesn't appear in any document (to avoid division by zero).
  + Otherwise, it calculates IDF as the logarithm of the ratio between the total number of documents and doc\_count.
* The function calculate\_tfidf calculates **TF-IDF scores** for all terms in all documents:
  + It gets the total number of documents (total\_docs).
  + For each term in the inverted index, it computes the term’s IDF.
  + Then, for each document containing that term, it multiplies the term frequency (TF) by the IDF to get the TF-IDF score.
  + These scores are stored in a nested dictionary tfidf\_scores where the outer keys are document IDs, and the inner keys are terms with their TF-IDF values.
* Finally, a sample TF-IDF dictionary for document with ID '1' is printed.

**Python Libraries Used:**

* math
* Part 4 : (Query Processing)

****

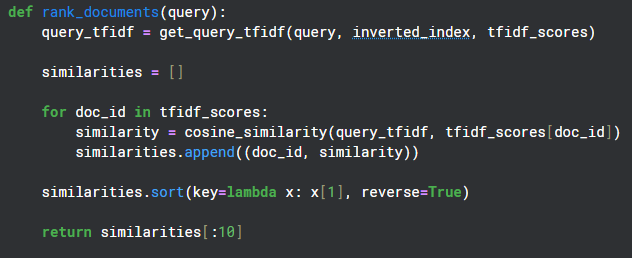
**Explanation:**

This function takes a user’s search query and converts it into a numerical representation called TF-IDF, which helps the search system understand how important each word in the query is.

* First, the query is cleaned and broken down into simple words by removing common words (like “the” or “and”), turning everything into lowercase, and reducing words to their root form (for example, “running” becomes “run”).
* Then, the function counts how many times each word appears in the query itself. Words that appear more often are considered more important for this specific query.
* After that, the function checks how common or rare each word is across all documents in the collection. If a word appears in many documents, it’s less helpful for distinguishing relevant documents; if it’s rare, it’s more important.
* By combining these two ideas — how often a word appears in the query and how rare it is in the document collection — the function assigns a weight (score) to each word.
* Finally, the function returns these weights so that the system can compare the query to the documents and find the best matches.

**Python Libraries Used:**

* collections.defaultdict
* math
* Part 5 : (Ranking)



**Explanation:**

* The function takes a **query string** as input.
* It uses get\_query\_tfidf to convert the query into a TF-IDF weighted vector.
* It then computes the **cosine similarity** between this query vector and the TF-IDF vector of every document.
* All document IDs and their similarity scores are stored in a list.
* The list is sorted so that the documents most similar to the query come first.
* Finally, it returns the **top 10 documents** that best match the query.

**Python Libraries Used:**

* Uses your previously defined get\_query\_tfidf and cosine\_similarity functions.
* No new libraries are needed here.