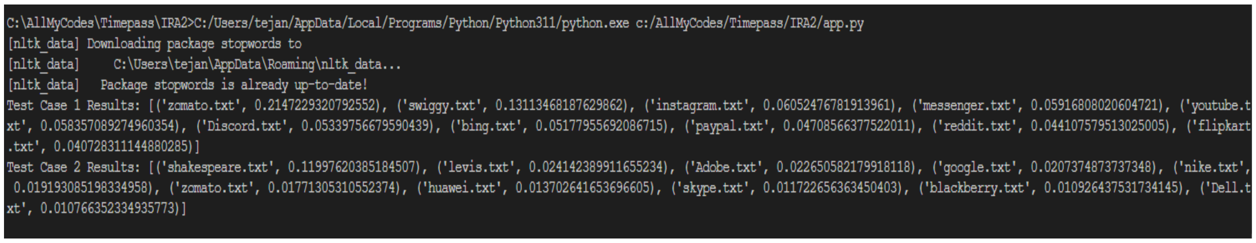
Information Retrieval Lab. (CSD358) Assignment 2

By: Tejansh Sachdeva (2110110555) and Mitaali Singhal (2110110883)

GitHub: https://github.com/tejanshsachdeva/Ranked-Retrieval

Output:



```
C:\AllMyCodes\Timepass\IRA2>C:/Users/tejan/AppData/Local/Programs/Python/Python311/python.exe c:/AllMyCodes/Timepass/IRA2/app.py
[nltk_data] Downloading package stopwords to
[nltk_data]     C:\Users\tejan\AppData\Roaming\nltk_data...
[nltk_data]   Package stopwords is already up-to-date!
Test Case 1 Results: [('zomato.txt', 0.2147229320792552), ('swiggy.txt', 0.13113468187629862), ('instagram.txt', 0.06052476781913961), ('messenger.txt', 0.05916808020604721), ('youtube.t
xt', 0.058357089274960354), ('Discord.txt', 0.05339756679590439), ('bing.txt', 0.05177955692086715), ('paypal.txt', 0.04708566377522011), ('reddit.txt', 0.04410757951302505), ('flipkart
.txt', 0.040728311144880285)]
Test Case 2 Results: [('shakespeare.txt', 0.11997620385184507), ('levis.txt', 0.024142389911655234), ('Adobe.txt', 0.022650582179918118), ('google.txt', 0.0207374873737348), ('nike.txt',
 0.019193085198334958), ('zomato.txt', 0.01771305310552374), ('huawei.txt', 0.013702641653696605), ('skype.txt', 0.011722656363450403), ('blackberry.txt', 0.010926437531734145), ('Dell.t
xt', 0.010766352334935773)]
```

Test Case 1 Results: [('zomato.txt', 0.2147229320792552), ('swiggy.txt', 0.13113468187629862), ('instagram.txt', 0.06052476781913961), ('messenger.txt', 0.05916808020604721), ('youtube.txt', 0.058357089274960354), ('Discord.txt', 0.05339756679590439), ('bing.txt', 0.05177955692086715), ('paypal.txt', 0.04708566377522011), ('reddit.txt', 0.04410757951302505), ('flipkart.txt', 0.040728311144880285)]

Test Case 2 Results: [('shakespeare.txt', 0.11997620385184507), ('levis.txt', 0.024142389911655234), ('Adobe.txt', 0.022650582179918118), ('google.txt', 0.0207374873737348), ('nike.txt', 0.019193085198334958), ('zomato.txt', 0.01771305310552374), ('huawei.txt', 0.013702641653696605), ('skype.txt', 0.011722656363450403), ('blackberry.txt', 0.010926437531734145), ('Dell.txt', 0.010766352334935773)]

For:
query1 = "Developing your Zomato business account and profile is a great way to boost your restaurant's online reputation"

And

query2 = "Warwickshire, came from an ancient family and was the heiress to some land"

Note: The output we got is quite different from what we expected, but that's not surprising given how much the results depend on pre-processing steps and indexing techniques. We used standard methods like tokenization, removing stop words, stemming, and applied **TF-IDF** weighting with **lnc** for documents and **lnt** for queries, combined with cosine similarity to rank the results. We also applied debugging techniques along the way to ensure everything was working properly. While we tried to refine the process as much as possible, the pre-processing and weighting methods we followed explain the results we're seeing.

Code Logic

1. **Tokenization:**
   - `tokenize(text)`:
     - The function tokenizes the input text by converting it to lowercase, splitting it into words, removing stopwords, and applying stemming (reducing words to their base form).
2. **Index Caching:**
   - `build_index_if_needed(corpus_path)`:
     - This function checks if the index is already built (`index_cache` is used for caching).
     - If not, it calls `build_index()` to generate the inverted index and document statistics.
3. **Index Building:**
   - `build_index(corpus_path)`:
     - This function parses all the text files in the `corpus_path` directory.
     - For each document, it tokenizes the content, counts term frequencies, and updates the dictionary and postings list (inverted index).
     - It also calculates document vector lengths (for cosine similarity) and stores document IDs.
4. **Query Processing (TF-IDF Calculation):**
   - `process_query(query, dictionary, N)`:
     - The input query is tokenized, and term frequencies are calculated for the query terms.
     - TF-IDF weights are computed for each query term using `log_tf` (logarithm of term frequency) and `idf` (inverse document frequency).
     - The weights are then normalized to avoid bias from longer queries.
5. **Query Normalization:**
   - `normalize_query_weights(query_weights)`:
     - The query weights are normalized by dividing each weight by the vector length, ensuring that the magnitude of the query vector is 1.
6. **Document Ranking (Cosine Similarity):**
   - `rank_documents(query_weights, dictionary, doc_lengths)`:
     - For each term in the query, the function calculates a score for every document that contains the term, using cosine similarity (based on `log_tf` for documents).
     - The scores are normalized by the document length to account for document size.

  - The top 10 documents are returned, ranked by their cosine similarity scores.

7. **Search Function:**

   – `search(query, corpus_path):`

     - This is the main function that handles the entire search process.
     - It builds the index if not already done, processes the query, and ranks the relevant documents.
     - The top 10 ranked documents are returned with their file names and scores.
