Information Retreival Assignment 1

Ayush Sharma (2020042); Aditya Jain (2020554); Rupin Oberoi (2020571)

Methodology:

First of all from the original dataset we have created another folder with similar structure and identical file names but with concatenated ‘TITLE’ and ‘TEXT’ sections.

For preprocessing we have implemented all the steps mentioned in the assignment. For tokenization and stop word removal we haved used the NLTK library.

Unigram Inverted Index

In the second question we were asked to implement the inverted index from scratch and then save it using pickle library of python and then demonstrate to load it using pickle itself

Here is a breakdown of the functions:

- **create_index(tokenized_docs):** This function takes in a dictionary of tokenized documents, where the keys are document IDs and the values are lists of tokens. It creates an inverted index, which is a dictionary where the keys are tokens and the values are lists of document IDs that contain that token. The function loops through each token in each document and adds the document ID to the list of IDs for that token in the inverted index. If the token is not already in the inverted index, it creates a new key with the token and a list containing the current document ID. The function returns the inverted index.

- **save_index(inverted_index)**: This function takes in an inverted index and saves it to disk as a binary file using the Python pickle module. It opens the file inverted_index.pickle in write binary mode, writes the inverted index to the file using pickle.dump(), and then closes the file.

- **load_index()**: This function loads the inverted index from disk by reading the binary file inverted_index.pickle. It opens the file in read binary mode, reads the inverted index using pickle.load(), and then closes the file. The function returns the loaded inverted index.

Next we were asked to implement the support for generalized queries and the following operations:

a. T1 AND T2
b. T1 AND NOT T2
c. T1 OR T2
d. T1 OR NOT T2

We were also required to compute the minimum number of comparisons done to execute the query. To implement all this we created some helper functions which are given below:

- **intersection(set_1, set_2):** This function takes in two sets and returns their intersection as a set, along with the number of comparisons made during the intersection operation. The function first initializes an empty set to hold the result, then checks if set_1 is smaller than set_2. If so, it swaps the two sets. The function then iterates over the elements of set_2, checking if each element is also in set_1. If it is, the element is added to the result set. The number of comparisons made during the intersection operation is returned along with the result set.
- **union(set_1, set_2)**: This function takes in two sets and returns their union as a set, along with the number of comparisons made during the union operation. The function first initializes a new set by copying set_1, then iterates over the elements of set_2, adding each element to the new set if it is not already present. The number of comparisons made during the union operation is returned along with the new set.

- **difference(set_1, set_2)**: This function takes in two sets and returns the difference between the two sets as a set, along with the number of comparisons made during the difference operation. The function first initializes a new set by copying set_1, then iterates over the elements of set_2, removing each element from the new set if it is present. The number of comparisons made during the difference operation is returned along with the new set.

- **evaluate_query(index, doc_ids, query):** This function takes in an inverted index, a list of document IDs, and a boolean query, and returns the set of documents that satisfy the query, along with the number of comparisons made during the evaluation. The function first splits the query into a list of words, then iterates over each word. If the word is "AND", "OR", or "NOT", the corresponding operation is saved in the variable operation. If the word is a term in the index, the corresponding posting list is retrieved, and the function adds the posting list to a list of sets doc_ids_sets. If the not_operation flag is set, the function applies the difference function to each set in doc_ids_sets and the set of all document IDs, and sets the result to res. Otherwise, the function applies the operation function to res and the new posting list, and sets the result to the new set. The function keeps track of the number of comparisons made during the evaluation and returns the result set and the number of comparisons. For processing boolean queries we process from left to right.

Bigram inverted index

For the bigram inverted index we iterate each file in the updated dataset and pass the file data and file_id to the bigram_inv_index() . In this function the file data is splitted in a list and then another iteration is done on this list to get two consecutive terms that forms a bigram.
Then following conditions are checked and terms are added in the bigram index dictionary accordingly:

- If the bigram is not present in the dictionary then a new (key:value) pair of bigram term and a new list containing the corresponding file_id is added to the dictionary.
- Else if bigram is present in dictionary then:
    - If the file_id is also present then we can skip this iteration
    - Else we add that file_id in the list of the corresponding bigram key in the dictionary.

In this way our bigram inverted index dictionary is created . For the query processing:

- We first preprocess the query in the same way as we did for the dataset. We get a list of bigrams in the query.
- Now we take a list of all the file_ids ans our initial answer .
- Then we iterate through this query bigrams list and for each bigram ‘b’ we do:
  intersection(ans , bigram_inv_index[b]) .

In this way we get our resultant list of documents containing all the bigrams of the query text.

Positional index

For positional index, for every unique word/token seen we add it as a key in a dictionary, and its corresponding value consists of the total number of occurrences for this word, and a dictionary with keys as names of documents and within the value for each document the positions of the word are stored.

When processing a query (after preprocessing), first a possible set of documents is initialized which are the documents in which first query term exists, then from there on we keep on deleting those documents which have do not satisfy the relative position requirements.

The results from positional and bigram inverted indices are guaranteed to be identical for 2 word long queries as they both will look for documents which satisfy word 1 and word 2 being together but for 3 word or longer queries, the results from bigram inverted index can be larger in size. This is because for a query ‘a b c’, bigram index will look for documents where ‘a b’ and ‘b c’ exist, but positional index will impose a stricter condition that ‘a b c’ should exist ie all 3 words together.

Individual contributions:
Unigram inverted index: Ayush
Bigram inverted index: Aditya
Preprocessing and positional index: Rupin