IR Assignment-1

For the Dataset -

- I have made a Github repository for the dataset , basically I am cloning that dataset for retrieving the dataset. It is publicly available at https://github.com/tonythomasndm/CSE508_Winter2024_A1_Dataset.git
- I have also made a directory for storing the preprocessed files, name -preprocessed_files

Data PreProcessing

- I have used nltk python module for preprocessing and imported two specific functions word_tokenize and stopwords for data preprocessing
- I have also downloaded the punctuations and stopwords nltk resources, even in case they are not available in the system.
- I have also created a variable ‘additional_path’ to store the values - ‘/content/’ for google collab or some additional path values - so that I don’t have to make changes all over the python file
- I am looping from 1 to 999 formatting string for the path name, and performing the file read from the formatted string path name
- Lowercased all the text and then tokenised it using the word_tokenize function, then removed the stopwords from English only(with reference to the stopwords resource downloaded earlier), then removed punctuations and finally removed the blank space tokens
- Finally, I wrote back in the preprocessed_files directory with the same name as the read_file. I made a sentence out of all tokens by merging them and using space as a delimiter.
- Then all the file pointers were closed

Printing any 5 random sample files

- I selected any 5 random integers using the random module with the range(1,1000) and then showed them before preprocessing and after preprocessing the files.

Creating a Unigram Inverted Index

- I, first of all, created an empty dictionary, inverted_index, to store the inverted index with all the postings, here the dictionary key will be the tokens, and the dictionary values will be the posting list, each list having postings(file names)
- I then started reading the preprocessed files one by one in range(1,1000) and tokenised them using split and space as delimiter here.
- I also created a total_docs empty lista dn stored the names of the docs one by one.
- If the token were already present as a key, then I would add the doc_id(file name) as a value to the postings list(dictionary value -> list) else, if the token is not present, then I would create an key-empty list value pair in the dictionary (inverted_index) then add the doc_id(file name) to the newly created token

Use of pickle module to store and load the Inverted index

- It was a new experience, exploring with pickle, it will help me storing the inverted index by pickle.dump() and loading by pickle.load()

Query Operations for Inverted Index

- I created four functions for problems of AND, OR, AND NOT, OR NOT, and all of them implemented set logic. I also used the total_docs for finding the differences for the NOT operator
- I created operatorsInp and tokensInp empty list to store all the input values.
- Inputted the value of n(no. Of queries) and performed preprocessing on operators by converting to lowercase and splitting by “,” delimiter
- Performed QueryText Preprocessing in the same way as Data preprocessing, on queryText to derive tokens from it and stored the operators in the operatorsInp list and queryText(tokens) in the tokensInp list.
- Then i took one query at a time,stored them in the boolean query format in the variable “query”.

- Assumption - If some token is not there in the inverted_index, Iwould create a token with the empty set, in that list, because if we perform all other four operations, iot should return the same, otherwise the code would return keyError, so I handled the missing tokens.
- Then I performed the operations according to the operators provided.
- Assumption - I am assuming that the number of tokens = number of operators + 1, so for 5 tokens, there should be 4 operators. If we ran out of operators, the code will throw some error. But it wouldn't throw an error if the number of operators is more, it will consider the required ones in sequence with the priority from left to right.
- At last, it prints the number of documents and names of the documents retrieved.

Creating Positional Index

- We will create a positional index dictionary, where we will store token as a key and another dictionary as a value. This another dictionary will store the doc_id as key and a set of positions of the token as value for the next dictionary.
- We will use 1-based indexing for token positions in the doc.
- We will read the same preprocessed files one by one and read the tokens. If the token is not in the dictionary positional_index, we will add that token, then assign another dictionary with doc_id and a empty set, then we will add the token position to this empty set

```
positional_index[token] ={}
positional_index[token]={doc_id:set()}
positional_index[token][doc_id].add(wordPosition)
```

- If in case the token is present , but that doc_id is that present the positional_index[token], then we add a element doc_id as key and a empty set as value (a dictionary) and then add the wordPosition to that doc_id's empty set.

```
positional_index[token][doc_id] = set()
positional_index[token][doc_id].add(wordPosition)
```

Use of pickle module to store and load the Positional index

- Exploring with pickle, it heped me storing the positional index by pickle.dump() and loading by pickle.load()

Taking Queries Input -

- Initialised an empty tokensInp list to store the input tokens for all n queries.
- Taking the input of the value of “n” and then taking n lines of queries and then performing query text Preprocessing as same as data preprocessing and storing them in an empty tokensInp list.

Processing the queries for the Positional index

- Taking the n queries one by one and storing all the tokens in the variable “tokens” for the ith query
- Then taking one doc from the positional_index of the first token
- And doing a nested search for position from positional_index[tokens[0]][doc], then calling a recursive function with initialization token_index-> 1 (i.e the second token), current tokens query and then the current position of that word, with the current doc, that we are accessing

```
for doc in positional_index[tokens[0]]:
  for position in positional_index[tokens[0]][doc]:
     fn(1,tokens,position,doc)
```

- After retrieving all the relevant documents from the list, we would print their count and the names of the document
- After every query, we would initialise the global variable as an empty list

Recursive Function for retrieving the documents for positional_index

- Before answering the process/method, I would like to answer why recursion. I chose the recursion because it provided the smooth flow as of stack and also helped me find out the answer with easier logic and explore all the function possibilities and retrieve all the relevant documents

- Base Case: if the index(following 0-based indexing) becomes equal to tokens.size(), then we can see that all the tokens are present in the current Word document and have been placed in the adjacent positions hence, we will store the document's name in the docs list.
- We will then check if the token is in the positional_index.keys(), if it is not there, then we will do an immediate exit, return the function and move on to the next iteration. Here, we know that the current query will not be present, and it is not required to check the next iteration. But it makes our code a little more complex and not easy to understand.
- Then, if the currentDoc is there in the next token’s list, and the currentWordPos+1(i.e adjacency of the words) is also there in next token’s currentDoc’s list
- Then we will call the function again recursively with index+1 i.e next token, tokens list, currentWordPos+1 and the currentDoc till the function reaches the base case
- Else in all other cases - return nothing i.e(don’t add any doc name to the list)