IR Assignment-1

Samyak Jain (2019098) and Sarthak Johari (2019099)

https://github.com/samyak19098/IR2022_A1_62

Question 1

We have used the ‘os’ library of python to traverse and pick files from the data directory. The text of each file is read and is preprocessed and finally, we have tokens corresponding to each file.

Preprocessing

After extracting the text from the files we performed the following preprocessing steps on the text to convert them into valid tokens which would be later used in making the unigram inverted index data structure.

1. First we converted the text to lowercase.
2. Text tends to have the shortened form of a combination of two words For Example “I will” is written as “I’ll” to fix this issue we have fixed the contractions from the text. We expand the contractions for eg. convert “don’t” to “do not”
3. We then performed word tokenization on the text.
4. For all tokens, non-alphanumeric characters were removed from them.
5. We then removed all the stopwords from our tokens.
6. Punctuations and blank spaces were removed from the tokens and finally we had our valid sets of tokens.

Methodology

Creating the inverted index data structure

After creating the valid tokens for each file we began to implement the unigram inverted index data structure.

For this we maintained two dictionaries, one for postings and other for maintaining the number of files where that token can be found. We iterated over the tokens of each file.

First we checked if the token was present in the index,

- If absent we added the token to the index and initialized it with the document Id of the file we are in.
- If present we checked if the document Id was present in the posting of that token.
    - If absent we added the document Id in the posting of the token and incremented the frequency count by 1.

After this we first sorted the entire data structures by the term in alphabetical order.

Then, we sorted the each individual posting list in ascending order of the document ids and finally our inverted index was ready

Creating the logical operators query

- AND
  We wanted to perform the logical AND operator to perform the query between the postings of two tokens. We made a function which took the posting list of both the tokens. Then we performed the intersection of the two by performing the merge algorithm. The fact that the posting lists are already sorted makes the merge operation much more efficient and simpler.

  The first step was to check if any of the postings were empty. i.e the token itself did not exist in our inverted index. If any of the two token's postings were empty we returned an empty array. Only if both were non empty we went ahead with merging.

  For the merge operation we kept two pointers one for each posting list and then traversed through the posting lists and compared the document ids where the pointers are during their traversal.

    1. If the document ids are the same that means the tokens are present in both the files we append this document id in our answer array that we maintain. Then the pointers are incremented by 1.
    2. If the value at first pointer is smaller than the value at second pointer we increment the first pointer by 1 and vice versa.

- OR

  To perform the logical OR operator. We checked the posting of both the tokens.

  1. If both the postings were empty we returned an empty array as the answer.
  2. If any one of the postings were empty, we returned the other token's posting list as the answer to the OR operation.

  3. For the merge operation we kept two pointers one for each posting list and then traversed through the posting lists and compared the document ids where the pointers are during their traversal.

     - If the document ids are the same that means the tokens are present in both the files we append this document id in our answer array that we maintain. Then the pointers are incremented by 1.
     - If the value at first pointer is smaller than the value at second pointer we increment the first pointer by 1 and add the document ID at the pointer in our answer array and vice versa.

     Finally we add all the remaining document IDs from the pointer that has not yet reached the end of its posting list

- NOT
  Here we simply returned all the document IDs that are not present in the posting list that was given as the input. If the posting list is empty then all the document ids would be returned.

- AND_NOT

  For implementing AND NOT we checked if the first posting was empty or not, if yes we returned an empty array as the answer. If not, we then implemented a merge algorithm.

  For the merge operation we kept two pointers one for each posting list and then traversed through the posting lists and compared the document ids where the pointers are during their traversal.
  1. If the document IDs are the same we just incremented the pointers.

2. If the pointer in the first posting list is at a value less than the other pointer in the second posting list. We added the document ID of the first posting list to our answer and incremented the first pointer by 1.
3. Finally if the pointer of the first posting list has not yet reached its end, and the other pointer has. We add all the remaining document IDs of the first posting list to the answer array.

- OR_ NOT

  Here we first perform the NOT operator on the second posting list. We then perform the OR operation on the first posting list and output obtained after the NOT operator of the second posting list.

Before performing the logical operations. The input query was preprocessed in the similar way we pre-processed the text that was present in the files. The query was converted into a list of tokens and then the operations are performed between these tokens.

Also if on preprocessing the query and its conversions into query tokens, if the number of operators is not suitable with the number of query tokens(i.e. no. of operators != no. query tokens - 1), we do not process anything in that case and return an error message.

Assumptions

1. We perform our query in a sequential order i.e from left to right. Also, no precedence is given to operators and the query execution order is not decided by any optimization.
2. We assume that the number of operators given are in accordance with the number of query tokens generated after the preprocessing of the query. If the number of operations given is not equal to the number of preprocessed query tokens minus 1, then the program terminates.
3. We have only counted the comparisons that take place during the merging operation of the postings i.e only where merging algorithm is perform, which is not including the not operation.
4. We do not perform any lemmatization or stemming of the file's tokens or the query's tokens.

5. During the input when the operators are entered they must be enclosed by ‘[]’ and are case sensitive (should be enter in capital letters).
6. The input query sequence should not be an empty string and the operator sequence should not be empty.

Question 2

We have used the ‘os’ library of python to traverse and pick files from the data directory. The text of each file is read and is preprocessed and finally, we have tokens corresponding to each file.

Preprocessing

1. The text is converted to lowercase.
2. Word tokenization is performed on the text.
3. Stopwords are removed from the tokens.
4. Punctuation marks have been removed from the tokens.
5. Blank space has been removed from the tokens.

These operations are performed in order for all the text of all the files to get a list of tokens for each file.

Methodology

Creating positional index

Two dictionaries have been maintained, one which contains the details of the document IDs and the respective positions in which they occur in the document; and the other one maintains the document frequency for each term i.e - the number of documents containing that term.

The method followed for positional index construction is as follows:
Iterate over all the documents and for each document, further, iterate over the list of tokens of that document.

For each token in a document, if it does not already exist in the index terms, we add the token as a term in our index and also add the corresponding document ID to the dictionary associated with each term and term's position to the list associated with each documentID in a term's dictionary.
If the token already exists in the index terms, we simply check that if the current document ID exists in the dictionary associated with the term. If not, we add it to that and also add the position to the document's associated list of positions. If it exists, we simply add the term's position to the document's position list.
Finally, we sort this in alphabetical order wrt terms and also sort the respective document list and each document's position list. We convert this whole structure to a nested list format where there is a dictionary with the terms as keys and the nested list of document IDs and positions.
The frequency for each term has been separately maintained in a dictionary throughout this process.
The structure of the positional index list is like : {'term1' : [[doc1, [pos11, pos21, pos31]], [doc2, [pos12, pos22, pos32]]], 'term2' : [[doc3, [pos13, pos23, pos33]]], 'term3' : …….}
And the frequency dict: {'term1':freq1, 'term2':freq2, ……..}

Phrase Query

- For performing the phrase query, we have to implement two levels of searching. Firstly we have to search such documents wherein the query tokens occur and then, in the common document's positions lists we have to check if all query tokens occur consecutively together in the proper specified order.

- For arriving at the common document wherein all the query tokens occur, firstly we will make 'n' outer pointers (where n = no. of query tokens). These outer pointers iterate over the nested list associated with each term and find the common document ID where all the terms occur. In each iteration, we check whether each outer_pointer points to the same document ID, if so then that document may be a document of interest for us and we go into searching the positions list of that document ID for each term and after doing that, we increment each outer_pointer by one. Otherwise, we increment that outer_pointer which corresponds to the lowest document ID magnitude wise.

- Now for inner level searching of the positions list of the common document ID for a term (which is a document of interest for us according to the previous point), we maintain 'n' inner_pointers which we'll use to iterate over the positions list of each term's common document ID. At every iteration, we check if position values pointed to by every inner pointer (starting from the second pointer) is one more than position values pointed to by the previous inner_pointer i.e - we check if the positions values are consecutive and in an increasing order starting from the position value of the first term in the query tokens. If so, then the current document ID whose positions list we are searching in is a valid document ID for our answer.

- Otherwise, we follow an algorithm in which we compare the jth inner_pointer's position value with the (j -1)th and while it is lesser, we keep on incrementing the jth inner pointer. Once it is greater than the (j-1)th inner_pointer's value, we check if it is just one more than the (j-1)th inner_pointer's value. If so, we increment the value of j and otherwise, we set the value of j = 1 and move the 0th inner_pointer ahead by one.

This way, we build our set of valid document IDs wherein the query phrase occurs.

General Implementation details

We traverse the data directory and read the individual file's texts. Also, we have made a mapping for document ID to Name for our use.
We first preprocess the given data, by preprocessing each file's text and converting it to a list of tokens corresponding to each file. Using these tokens, a positional index is built which includes the building of both the index structure holding the documentIDs, term positions for all the terms and also, the frequency dictionary which contains the document frequency of each term.

The phrase query input is firstly preprocessed every time using the same set of preprocessing operations used to preprocess the file text. The operations are also performed in the same order as they are for the file texts. The phrase query string is converted into query tokens and they are used for phrase query searching purposes.

Note
- As we are removing the stop words from the text, then if our query is "good morning" and if suppose document with document ID 10 contains a sentence "Good is morning". Then as 'is' is a stopword it will be removed and document ID 10 will be a valid answer for our query.

Assumptions

1. If we enter a query with more than 5 tokens, our program returns with a message and no query is performed.
2. In the case where there is a token in the phrase which is not present in the index, the program returns with a message and returns result 0, []. (0 no. of docs and [] empty list of doc names).
3. Input phrase string should not be empty.