**INFORMATION RETRIEVAL** TEAM MEMBER 1: Drishya Uniyal

**ASSIGNMENT 1 REPORT** TEAM MEMBER 2: Shashank Rustagi

## Preprocessing Steps:

- ★ Import necessary libraries like nltk, os, re, glob
- ★ For the English **STOPWORDS**, import them from nltk.corpus.
- ★ We will also be using the word.tokenize method of nltk in our preprocessing.
- ★ We will store the stop words as a set using the .words method of stopwords of the English language.
- ★ We will create an empty ***inverted index dictionary*** where we will store the words to document mapping.
- ★ We will create a dictionary for keeping the mapping of the document ID and Document name.
- ★ The next part will be ***dataset preparation*** where we will first store the current working directory path in some variable. After that, we will use glob in order to get the folder where the text files are stored.
- ★ We will then split the file names based on the last index whether it is txt etc.
- ★ The next step will be **indexing** each of **these files** which will internally contain the preprocessing and updating the index dictionary itself.
- ★ So for all the files in the files, we will call **update index** method which will **try** to open and read the file name and in the **catch** block, we will handle the **UNICODEDECODE exception** if it comes.
- ★ Then we will call the **preprocess** method on the files thus received and this method will lower case all the words in the text and then it will remove the useless symbols using the regular expressions library sub method. After that, we will tokenize the words and then we will remove the stop words from the text.
- ★ Now, in this preprocessed list, we will take each word, word by word and then we will create its **unigram inverted index.**
- ★ Now, after all this hassle, we will simply update the index dictionary and update the iterator i so that we can do the same thing for the next files.
- ★ We define a global power set that will sort this list and store it in the power set.

## Queries

- **OR query**
    - We will keep the words x and y as sorted and then on these sorted words, we will simply keep the comparisons variable for counting the number of comparisons. Initially, we will initialize this variable with 0 and then we will simply keep the iterators i and j and then we initialise them also to 0.
    - So, we will check if both i and j are within the length of x and y respectively, if they match, then we will simply append the result in the OR array and increment both i and j. And as we have compared these two variables, the number of comparisons will also increase by 1.
    - If that is not the case, then the comparisons count will increase by 2.
    - If the element at index i for x is less than an element of index j for y then in that case, we append the X[i] in the OR array and increment i by 1 and if not, we will append Y[j] in the OR Array and increase j by 1.
    - Now two things can happen, if the length of one of those gets finished, then we need to check outside the while loop for each of them, if they are still remaining, in that case, we will have to append the OR array likewise and increment the suitable iterator as required.
    - This function will simply return the OR array and the total number of comparisons.

- **AND query**
    - We will again keep all the x and y sorted initially and after sorting, we will simply store the words in the AND array.
    - We will compare x[i] and y[j] and as we have compared these two, then the number of comparisons will increase by 1 and if they are similar, in that case, we need to append x[i] in the AND Array.
    - Also, we will increment both i and j in that case if both of the words get matched.
    - Otherwise, x[i] and y[j] are not similar, so the number of comparisons will increase by 2. And if x[i]<y[j] then we will simply increment i by 1and otherwise we will increment j by 1.
    - In the end, we will simply return the AND array and the total number of comparisons.

- **SET DIFFERENCE**
    - **A or NOT B** is implemented in set theory as P-(B-A)
        - So, we will find the set difference between X and Y i.e. X-Y, inherently it means that the number of words which are in X but are not in Y. We will simply keep X as sorted and then we will have two iterators i and j both pre-initialized to 0.
        - We will traverse both the lists using i and j and i will go till the length of X and j will go till the length of Y. If x[i] and y[j] are matching, then, in this case, we will not include this in our answer, as we need X-Y. So the number of comparisons will increase by 1 and both i and j will get increased by 1.
        - Otherwise, increase the number of comparisons by 2. If x[i]>y[j], then, in that case, we will add the X[i] in the OR NOT array and increment i by 1. Also, if the array stops either when i surpasses len(X), then check if j has not surpassed Y then increment the j and also append the OR NOT array.
        - Now, we just have to call this utility function on A, B and A', U and sum up the number of comparisons we got in both of these.
    - **A and NOT B** is implemented in set theory as B-A
        - We will simply call setDifference on B,A
        - A AND NOT B is B-A simply

- **MAIN FUNCTION**
    - We are simply asking the user to **enter** the **query**
    - We will then **preprocess** the user query and **remove all the stop words** and **convert them into lower case**, and do everything that was mentioned in the preprocess report.
    - Then we will **print the number of documents matched and the minimum number of comparisons will also be printed.**

```
Enter no of queries : 1
========== Preparing Dataset ========== [OK]
Vocabulary : 65185 #Documents : 1133

You are requested to enter your query : the lion stood thoughtfully for a moment
Enter the operation sequence (Only 3 operations is needed) : [OR,OR,OR]
Running query :  lion [OR stood OR thoughtfully OR] moment
96
Document ID :  [116, 135, 138, 218, 241, 248, 291, 295, 318, 322, 378, 418, 533, 595, 596, 597, 602, 635, 688, 689, 753, 791, 8
42, 977, 980, 1027, 1039]
Document Names (10 max) :  ['bitnet.txt', 'boneles2.txt', 'booze1.fun', 'christop.int', 'collected_quotes.txt', 'computer.txt',
'deep.txt', 'devils.jok', 'dromes.txt', 'dthought.txt']
Number of documents matched:  27
Minimum Operation :  96
```

Q2.

## Assumptions:

1. Given the phrase query, after pre-processing, the tokens generated should be of length <=5.

   Example: If the query is “Well Costello, I'm going”

   After pre-processing the tokens are generated and then we check if the length is <=5 or not.

   Output: ['well', 'costello', "'m", 'going']

   And its length is less than 5 so the further process will take place.

## Initial necessary steps:

★ Import the important libraries.

a. Nltk
b. os
c. re
d. string
e. NumPy
f. pandas
g. pickle

★ Give the path to the folder where the files are kept. Use os.listdir() to check the files in the folder and append their paths finally so that they can be used for further processing.

## (a) Carry out the following preprocessing steps on the given dataset

### Convert the text to lower case

★ For each file in the folder, convert the text in files to lower using the .lower() function.

### Remove punctuation marks from tokens

★ Loop through the text and check if text in !.,:@#$%^&?<>*()[}{]-=;/\"\\\t\n", then remove the punctions.

### Remove blank space tokens

★ Replace the blank spaces from the text.

### Perform word tokenization

★ Use word_tokenize() from nltk and process the data and convert to tokens.
★ Perform Lemmatization

### Remove stopwords from tokens

★ From nltk download English stopwords and use them.
★ Loop and check if a stopword is present in tokenized words then remove it.

Input: This will be done for both text and phrase queries.

```
Enter the phrase query:  Well Costello, I'm going
```

Output:

```
: ['well', 'costello', "'m", 'going']
```

### Implement the positional index data structure.

★ After the preprocessing steps, we get the tokens of text and we will work on them now.
★ Initialize a dictionary vocab {} which will be storing the tokens and their corresponding positions in the document.
★ Check the tokens and if the word is already present in the dictionary, add the document and position it is stored at.
★ If the word is not present in the dictionary, create a new entry.
★ Update the frequency of the word for every document and the number of documents it is present in.
★ Finally, print the number of documents and the name of the document.

### (c) Provide support for the searching of phrase queries. You may assume query length to be

★ Take the phrase query as input and handle it using try and catch so that if a word is not present in any document it prints “Exception Occurred.”
★ Taken the assumption that the length of the query should be less than or equal to 5.
★ Do the required preprocessing steps.
★ Get the postings list, the dictionary which contains the word and the position at which the word is present in a document. Update the frequency as a single word may be present in many documents.
★ If we find it, print the documents and the name of the document.

Output

```
Number of Documents retrived are : 3
List of Documents retrived are : {'abbott.txt': [11], 'aclamt.txt': [168], 'a_fish_c.apo': [263]}
```