Information Retrieval
Assignment -1
Group-7

Submission id:
Roshan S
roshan20039@iiitd.ac.in

Submitted To:
Dr Rajiv Ratn shah

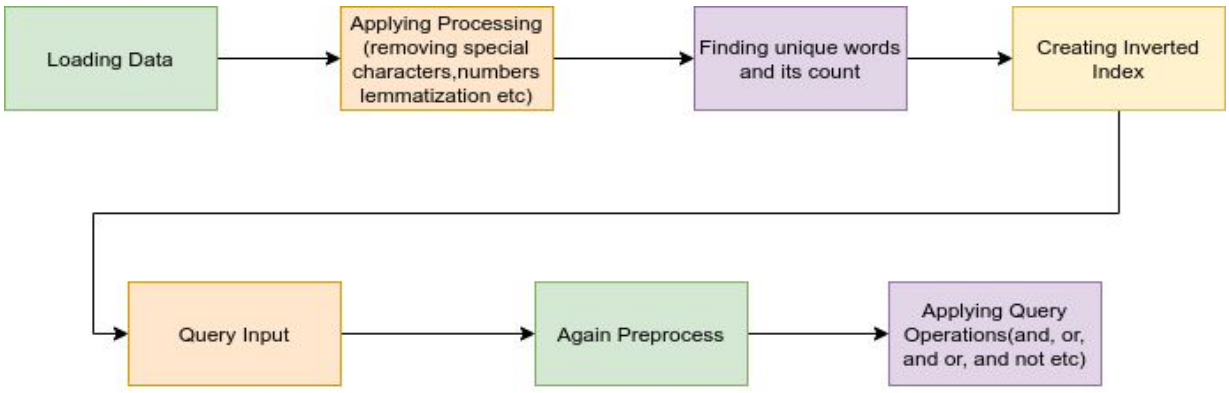
NOTE:

1. **The language used: Python.**
2. **We have used libraries pickle, glob, nltk, etc.**
3. **Dataset Link: http://archives.textfiles.com/stories.zip**
4. **Github Repository link: https://github.com/roshan-1/IR2021_A1_7**

PROBLEM STATEMENT:

a. Carry out the suitable preprocessing steps on the given dataset.
b. Implement the unigram inverted index data structure.
c. Provide support for the following queries-
    1. x OR y
    2. x AND y
    3. x AND NOT y
    4. x OR NOT Y

METHODOLOGY:



1. **Importing Libraries:**

Firstly, we need to import all the libraries, which are crucial in performing preprocessing steps, inverted index generation, query operations, etc.

- **glob:** Files matching specified patterns can be retrieved using glob.
- **re:** re is used to import Regular Expressions/RegEx in python.
- **nltk:** It is an essential library here which we are going to use for tokenization and lemmatization.
- **pickle:** used to convert data into byte streams, using pickling and unpickling make operations faster and save time.
- **Stopwords:** used at the time of preprocessing to remove English stopwords like a, an, the, he, etc.

```
import glob
import re
import pickle
import nltk
import os
from nltk.corpus import stopwords
from nltk.stem import WordNetLemmatizer
from nltk.tokenize import word_tokenize
stop_words = set(stopwords.words('english'))
```

2. Loading Dataset:

- Here we are downloading data from the given link.
- Unzipping that folder.
- We are storing the location of stories data in "path".
- Using glob.glob(path) recursively retrieves the path of all files/directories recursively from the stories dataset.
- There are 455 files in the stories folder, 16 files in the SRE directory, and one file in the FARNON directory.
- Total we need to consider only 467 files obtained by excluding all index.html files, FARNON directory, by including 452 files from stories folder and 15 files(.txt files) from the SRE folder.

```
for file1 in glob.glob(path):
    fpath = file1
    fname = file1.split("/")[1]
```

```
    fname = fname.split(".")[0]
```

3. Data Preprocessing:

As the loaded Stories data is in a very raw/unstructured format, so here we are performing several preprocessing steps to clean the data, which are as follows:

- ➢ Firstly we are checking if the obtained path is a file or directory.
- ➢ If the obtained path is the SRE directory or files excluding index files, we will perform all the mentioned preprocessing operations.

➢ Reading Text From Document:

After this, use open() to open the file, read contents of files using read(), and store them in the doc variable.

```
file1 = open(file1,"r",encoding='unicode_escape')
doc = file1.read() #reading contents of doc
```

➢ Deleting Special Characters:

- After reading the contents of files, Deleting special characters from like .,\,{,] &, etc.
- Matching all the input strings which are not character or numbers using [^a-zA-Z0-9\s] and storing them in the regex.
- Substituting regex with blank ‘’ in input and storing final result in output.

```
def delete_spec_chars(input): #function to delete
special characters
    regex = r'[^a-zA-Z0-9\s]'
    output = re.sub(regex,'',input)
    return output
```

➢ Deleting Numbers:

Checking if the doc contains digits using ‘\d+’ and substituting them with ‘’ in doc.

```
doc = re.sub(r'\d+','',doc) #deleting numbers
```

➢ Extracting Tokens:

- Tokenization is the process of splitting larger text into smaller lines or words.
- To perform Tokenization on the doc, using word_tokenize() module from nltk.tokenize and storing them in variable tokens.
- word_tokenize is splitting strings into tokens based on white spaces and punctuations.

```
tokens = word_tokenize(doc) #extracting tokens
```

➢ Converting into the Lower case:

Now iterating through tokens and converting every character from uppercase to lowercase using lower() of python to avoid ambiguity, and storing them in tokens_lower variable.

```
tokens_lower = [word.lower() for word in tokens]
```

➢ Lemmatization:

- Lemmatization referred to morphological analysis of words, which considers the context and returns the base word.
- To perform lemmatization, using the WordNetLemmatizer() module from nltk.stem.
- Here, iterating through words and performing lemmatization for that word then storing results in list words_lematized.

```
def lematize(words): #lemmatization
    lemmatizer = WordNetLemmatizer()
    words_lematized = [lemmatizer.lemmatize(word) for
word in words]
    return words_lematized
```

➢ Removing Stop words:

After Lemmatization iterating through list word_tokenize and removing all the stop words and storing final result in list tokens_final.

```
tokens_final = [word for word in tokens_lematized if
word not in stop_words and len(word) > 1]
```

➢ After performing all the preprocessing steps, creating a list **file_info** contains the id corresponding to each document.

```
file_info[doc_id] = os.path.basename(fpath1)
 doc_id += 1
```

➢ Finding Unique Words:

- Now the next step is to find all the available unique words from the corpus and its frequency.
- To perform this operation, firstly creating a list of unique words and using set() to convert the list of words into distinct elements.
- Now iterating through each word and checking if the word available is unique and also counting occurrences.
- Created dictionary word_freq to store the frequency of each word in (word, frequency) form.

```
def find_unique(words): #function to find unique words
along with its frequency in the doc
    unique_words = list(set(words))
    word_freq = {}
    for word in unique_words:
        word_freq[word] = words.count(word)
        #word_freq[word] += 1
    return word_freq

uq_dict = find_unique(tokens_final)
 unique_words_dict.update(uq_dict)
```

➢ Creating list unique_words which is a combined list of unique words from all the docs.

```
unique_words = list(set(unique_words_dict)) #list of
unique words in all docs combined
```

➢ Storing all the obtained tokens frequency, unique words, and document id in respective pickle files to save time from long-running operations.

```
unique_words,unique_words_dict,file_info
=process('stories/*')
f = open('tokens.pkl','wb')
pickle.dump(unique_words,f)
f.close()
f1 = open('token_freq.pkl','wb')
pickle.dump(unique_words_dict,f1)
f1.close()
f2 = open('file_info.pkl','wb')
pickle.dump(file_info,f2)
f2.close()
```

4. Implementing Unigram Inverted Index Data Structure:

- After performing all the preprocessing steps, now we have to create an inverted index data structure.
- The inverted index is a data structure that contains the word and its corresponding document id (i.e., in which it is present) and frequency(count of a word in that document).
- Here we are using a linked list to implement the inverted index (creating a dictionary of the linked list), and steps to create that are as follows:

➢ Creating Linked List:

- Creating a class node i.e node for each word contains docId, and frequency of that word in doc.
- Using next to append the next document in which that word is available and its count.

- So the format of Linked List for inverted index is :
- word- (doc_id1,frequency)->(doc_id2,frequency).....and so on.

```
class node:
    def __init__(self,doc_id,freq=0):
        self.doc_id = doc_id
        self.freq = freq
        self.next = None

class linked_list:
    def __init__(self, head=None, tail=None):
        self.head = head
        self.tail = tail
        self.len = 0
```

- Also creating an empty linked list “universal _list” to store the linked list of all the document ids..

```
universal_list = linked_list()
```

➢ Creating Dictionary to Store inverted Index:

Firstly here traversing through each token and creating a dictionary that contains (key, value) pair where the key is the word and value is the linked list for that word

```
inverted_index = {}
for word in tokens:
        inverted_index[word] = linked_list()
```

➢ Creating Inverted Index:

- Firstly we need to perform all the above preprocessing operations on the document.

- Then iterating through each word in created uq_dict (that contains a unique word and it’s frequency), For each unique word creating a linked list and appending a (doc_id, frequency).

```
for word in uq_dict.keys():

inverted_index[word].append(doc_id,uq_dict[word])
doc_id += 1
```

➢ **Function to append the new node (doc_id, frequency):**

- This is a simple linked list operation where if no node exists then creating a new node otherwise appending a new node in the linked list corresponding to that word in O(1) time.

- Using the len variable which is counting the length of the linked list for each word i.e. len is the count of documents in which the word is present.

```
def append(self,id,freq):
        new_node = node(id,freq)
        if self.head is None:
            self.head = self.tail = new_node
        else:
            self.tail.next = new_node
            self.tail = new_node
        self.len += 1
```

➢ **Function to display the linked list:**

This function is used to display the linked list of inverted indices.

```
 def display(self,file_info):
        temp = self.head
        print(self.len)
        while(temp):

print(temp.doc_id,temp.freq,file_info[temp.doc_id])
            temp = temp.next
```

5. **Query Operations:**

➢ Taking the query as an input from the user and applying processing operations on that query.

```
query = input('Enter the Query')
processed_query = process_query(query)
process_word = processed_query.split()
```

➢ Taking the operations as input from the user and converting those into the list of operations.

```
query_op = input('Enter the Operations')
query_op = query_op.replace('[','')
query_op = query_op.replace(']','')
query_op = query_op.split(',')
```

➢ Now For each word in the process word (i.e. preprocessed query), firstly checking if the word is available in tokens or not if yes then appending the available linked list (from the inverted index) into the total, where the total is the list of a linked list (i.e. linked list of words).

```
total = []
i=0
for word in process_word:
   if word in tokens:
      total.append(inverted_index[word])
```

➢ Here, for each word from the list of processed words, taking the first two words as word1 and word2 and applying a query on those two words.

- Removing the first two words from the linked list.
- After that checking the type of operation.
- Adding the result of the query (i.e. word 3) again in the first position of the total list.
- Now again a query will be performed between the result and the next word.
- Returning the count of comparisons.

```
final_count = 0
```

```
    for i in range(len(process_word)-1):
        word1 = total[0]
        word2 = total[1]
        if query_op[i] == 'OR':
            word3,count1 = query_or(word1,word2)
        elif query_op[i] == 'AND':
            word3,count1 = query_and(word1,word2)
        elif query_op[i] == 'OR NOT':
            word3,count1 = query_ornot(word1,word2)
        elif query_op[i] == 'AND NOT':
            word3,count1 =query_andnot(word1,word2)
        final_count += count1
        total.remove(word1)
        total.remove(word2)
        total.insert(0,word3)
    resultant = total[0]
```

➢ **x OR y:**

- Storing the head node of both the query lists in p1 and p2 respectively.
- Traversing both the lists simultaneously and performing and operations.
- Here count variable is used to keep track of the number of comparisons.
- When p1 and p2 are not null, then comparing the doc_id of both the lists :

1. If they are equal then adding that respective doc_id and frequency in the resultant linked list and incrementing the pointer by 1.

2. If the doc_id of the first posting list i.e p1 is greater than the doc_id of the 2nd posting list i.e p2 then incrementing the pointer of p2 by 1, as by default inverted indices are in ascending order, and appending the (doc_id, frequency) of p2.

3. Otherwise incrementing the pointer of p1 by 1, and appending the (doc_id, frequency) of p2.

```
def query_or(list1, list2):
    p1 = list1.head
    p2 = list2.head
    resultant = linked_list()
    count = 0
    while p1 is not None and p2 is not None:
        if p1.doc_id == p2.doc_id:
            count += 1
            resultant.append(p1.doc_id, p1.freq)
            p1 = p1.next
            p2 = p2.next

        elif p1.doc_id > p2.doc_id:
            resultant.append(p2.doc_id, p2.freq)
            count+= 1
            p2 = p2.next

        else:
            resultant.append(p1.doc_id, p1.freq)
            count+= 1
            p1 = p1.next

    while p1 is not None:
        resultant.append(p1.doc_id, p1.freq)
        p1 = p1.next

    while p2 is not None:
        resultant.append(p2.doc_id, p2.freq)
        p2 = p2.next

    return resultant, count
```

➢ **x AND y:**

- Storing the head node of both the query lists in p1 and p2 respectively.
- Traversing both the lists simultaneously and performing and operations.
- Here count variable is used to keep track of the number of comparisons.
- When p1 and p2 are not null, then comparing the doc_id of both the lists :

    1. If they are equal then adding that respective (doc_id, frequency) in the resultant linked list and incrementing the pointer by 1.

    2. If the doc_id of the first posting list i.e p1 is greater than the doc_id of the 2nd posting list i.e p2 then incrementing the pointer of p2 by 1, as by default inverted indices are in ascending order.

    3. Otherwise incrementing the pointer of p1 by 1.

```
def query_and(list1, list2):
    p1 = list1.head
    p2 = list2.head
    resultant = linked_list()
    count = 0
    while p1 is not None and p2 is not None:
        if p1.doc_id == p2.doc_id:
            count += 1
            resultant.append(p1.doc_id, p1.freq)
            p1 = p1.next
            p2 = p2.next

        elif p1.doc_id > p2.doc_id:
            count+= 1
            p2 = p2.next

        else:
            count+= 1
            p1 = p1.next

    return resultant, count
```

➢ x AND NOT y:

AND NOT operation is equivalent to subtracting the posting list of y from the posting list of x.

```
def query_andnot(list1,list2):
    resultant, count1 = subtract(list1, list2)
    return resultant, count1
```

➢ x OR NOT y:

OR NOT operation is the combination of:

1. Difference between posting list 2 and 1 and storing its result in list3.
2. Subtracting the posting list 3 from univarsal_list.

```
def query_ornot(list1, list2):
    list3, count1 = subtract(list2, list1)
    resultant, count2 = subtract(universal_list, list3)
    return resultant, count1+count2
```

➢ x Subtract y:

- Storing the head node of both the query lists in p1 and p2 respectively.
- Traversing both the lists simultaneously and performing and operations.
- Here count variable is used to keep track of the number of comparisons.
- When p1 and p2 are not null, then comparing the doc_id of both the lists:

    1. If they are equal then incrementing both pointers by 1.

    2. If the doc_id of the first posting list i.e p1 is greater than the doc_id of the 2nd posting list i.e. p2 then here we are creating a new node with the doc_id of p2 and its frequency.

3. After that appending that new node in the resultant linked list, and incrementing pointer of p2 by 1.
4. Otherwise appending the doc_id of p1 and its frequency in the resultant list and incrementing the pointer of p1 by 1.

```
def subtract(list1, list2):
    p1 = list1.head
    p2 = list2.head
    resultant = linked_list()
    count = 0
    while p1 is not None and p2 is not None:
        if p1.doc_id == p2.doc_id:
            count += 1
            p1 = p1.next
            p2 = p2.next

        elif p1.doc_id > p2.doc_id:
            resultant.append(p2.doc_id, p2.freq)
            count+= 1
            p2 = p2.next

        elif p2.doc_id > p1.doc_id:
            resultant.append(p1.doc_id, p1.freq)
            count+= 1
            p1 = p1.next

    while p1 is not None:
        resultant.append(p1.doc_id, p1.freq)
        p1 = p1.next

    return resultant, count
```

6. Query Results:

```
Enter the Number of Queries 2
Enter the Query lion stood thoughtfully for a moment
Enter the Operations [OR,OR,OR]
Processed Query  lion OR stood OR thoughtfully OR moment
Number of Documents Matched  270
Number of Comparisons Required  673
Documents Retrieved
Doc Id  1 Doc Name  buggy.txt
Doc Id  2 Doc Name  clon
Doc Id  3 Doc Name  shulk.txt
Doc Id  5 Doc Name  sight.txt
Doc Id  8 Doc Name  piracy.sto
Doc Id  9 Doc Name  nigel.10
Doc Id  10 Doc Name  igiv
Doc Id  11 Doc Name  lionwar.txt
Doc Id  13 Doc Name  aluminum.hum
Doc Id  15 Doc Name  dskool.txt
Doc Id  16 Doc Name  bulphrek.txt
Doc Id  18 Doc Name  ab40thv.txt
Doc Id  19 Doc Name  16.lws
Doc Id  20 Doc Name  horswolf.txt
Doc Id  21 Doc Name  beast.asc
Doc Id  22 Doc Name  rocket.sf
Doc Id  24 Doc Name  consumdr.hum
Doc Id  25 Doc Name  domain.poe
Doc Id  26 Doc Name  hound-b.txt
Doc Id  27 Doc Name  lmtchgrl.txt
Doc Id  28 Doc Name  redragon.txt
Doc Id  31 Doc Name  floc
```

```
Enter the Query telephone,paved, roads
Enter the Operations [OR NOT,AND NOT]
Processed Query  telephone OR NOT paved AND NOT road
Number of Documents Matched  345
Number of Comparisons Required  918
Documents Retrieved
Doc Id  1 Doc Name  buggy.txt
Doc Id  2 Doc Name  clon
Doc Id  3 Doc Name  shulk.txt
Doc Id  4 Doc Name  mario.txt
Doc Id  6 Doc Name  tao3.dos
Doc Id  7 Doc Name  crabhern.txt
Doc Id  8 Doc Name  piracy.sto
Doc Id  9 Doc Name  nigel.10
Doc Id  10 Doc Name  igiv
Doc Id  11 Doc Name  lionwar.txt
Doc Id  12 Doc Name  blossom.pom
Doc Id  14 Doc Name  lionbird
Doc Id  15 Doc Name  dskool.txt
Doc Id  16 Doc Name  bulphrek.txt
Doc Id  18 Doc Name  ab40thv.txt
Doc Id  20 Doc Name  horswolf.txt
Doc Id  21 Doc Name  beast.asc
Doc Id  22 Doc Name  rocket.sf
Doc Id  23 Doc Name  poem-1.txt
Doc Id  25 Doc Name  domain.poe
Doc Id  28 Doc Name  redragon.txt
Doc Id  29 Doc Name  panama.txt
```

7. **Assumptions:**

- ➢ The query does not contain more than one continuous special character.
- ➢ The query does not contain more than one digits together.