Information Retrieval
Assignment - 2
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**
   https://drive.google.com/file/d/1okxas8RjrsGuKKSpEDRfxEq6aSa6CaDk/view

---

Question 1 :

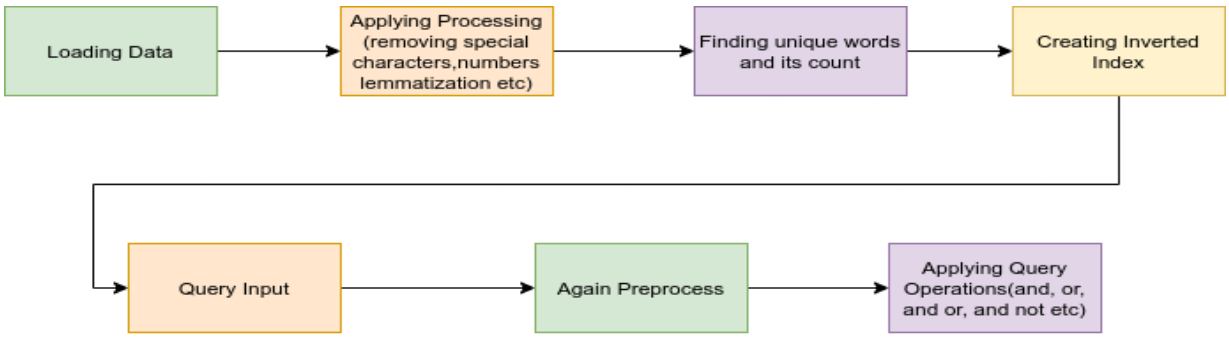
a. Carry out the following preprocessing steps on the given dataset
   i. Convert the text to lowercase
   ii. Perform word tokenization
   iii. Remove stopwords from tokens
   iv. Remove punctuation marks from tokens
   v. Remove blank space tokens

b. Implement the positional index data structure

c. Provide support for the searching of phrase queries. You may assume query length to be less than or equal to 5.

d. During the demo, your system would be evaluated against some phrase queries. Marks would be awarded based on the correctness of the output.

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
```

➢ 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]
```

➢ Removing Stop words:

After converting into lower case iterating through list word_lower and removing all the stop words and storing final result in list tokens_final.

```
tokens_final = [word for word in tokens_lower 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
```

4. **Implementing positional index data structure :**
   - Positional index is a dictionary with the tokens after preprocessing as the keys and the values are the corresponding posting lists.
   - Each posting list is a linked list where each node of the linked list contains the following : **doc_id** which denotes the id of the document which contains the word, **freq** which contains the frequency of the word in the document, **pos** which is a list of positions at which the word appears in the particular document.
   - We iterate through the tokens_final list that we get from each document after preprocessing and fill the positions of the words in the pos field of the linked list corresponding to each word. It is done as follows:

```
 for pos,word in enumerate(tokens_final):
   if word in inverted_index:  #if word already found in index
     found = 0
     temp = inverted_index[word].head
     while(temp):
       if temp.doc_id == doc_id:#current document already exists in posting list
         temp.pos.append(pos)
         temp.freq = temp.freq + 1
         found = 1
         break
     temp = temp.next
     if found == 0:#first occurrence of word in current doc
        pos_list = []
       pos_list.append(pos)
       inverted_index[word].append(doc_id,1,pos_list)#append node in linked list
   else:#first occurrence of word in any doc
      inverted_index[word] = linked_list()
      pos_list = []
```

```
        pos_list.append(pos)
        inverted_index[word].append(doc_id,1,pos_list)
```

5. **Processing phrase queries**
    1. We apply the same preprocessing techniques on the query.
    2. Iterate through the pre-processed query and take two words at a time.
    3. Iterate through the posting lists of both the words in consideration and if the doc_ids match, iterate through the position lists and find positions such that pos(word1) = pos(word2) - 1.
    4. Repeat step3 recursively by updating the first posting list as the result of the merging carried out in the previous iteration.
    5. Display the resultant linked list at the end of the iteration.

    Code is as follows

```
for i in range(len(process_word)-1):
      resultant = linked_list()
      word1 = process_word[i]
      word2 = process_word[i+1]
      if i == 0:
         temp1 = inverted_index[word1].head
      else:
         temp1 = result.head
      temp2 = inverted_index[word2].head
      flag = 0
      while temp1 and temp2:
         if temp1.doc_id == temp2.doc_id:
             pos_list = []
             for pos1 in temp1.pos:
                for pos2 in temp2.pos:
                    if pos1 == pos2 - 1:
                        pos_list.append(pos2)
                    elif pos2 > pos1:
                        break
                    if len(pos_list) > 0:
                        flag = 1
                       resultant.append(temp1.doc_id
                        ,len(pos_list),pos_list)
                        result = resultant
                   elif flag==0:
                        result = linked_list()
                        temp1 = temp1.next
```

```
                    temp2 = temp2.next
            elif temp1.doc_id < temp2.doc_id:
                temp1 = temp1.next
            else:
                temp2 = temp2.next
```

Question 2

Scoring and Term-Weighting

Similarity - The similarity between two data is a numerical measure of the degree to which the two data are alike, or similar.

1. **Jaccard Coefficient :** It is a measure of similarity for two sets of data, with a range from 0% to 100%. The higher the percentage, the more similar the two sets. Formula for Jaccard coefficient is as follows :

   Jaccard Coefficient = Intersection of (doc,query) / Union of (doc,query)

   jaccard(A,B) = |A ∩ B| / |A ∪ B|
   jaccard(A,A) = 1 : *represents set is completely similar*
   jaccard(A,B) = 0 if A ∩ B = 0 : *represents no similarity*

   Steps:

   1. Loading the Stories Dataset and performing required preprocessing steps.
   2. Created file_info dictionary to store its document id and name.
   3. Created file_data List of dictionaries to store each document's data i.e. there are a total 467 dictionaries are available for each document in the list.
   4. Taking input for query and preprocessing it.
   5. Created a calculate_jaccardcoef() to calculate the Jaccard coefficient for the query.

```
def calculate_jaccardCoef(query):
   #Function to Calculate Jaccard Coefficient
   doc_id=1
```

```
    #Dictionary to store jaccard coef with respective doc-id in
key:value format.
    dict_jaccard_val={}
    for li in file_data:
        # print(val)
        union_result=union(query,li)
        intersection_result=intersection(query,li)
        print(len(union_result))
        print(len(intersection_result))
        #Finding jaccard val for query with each document
        dict_jaccard_val[doc_id] =
len(intersection_result)/len(union_result)

        doc_id = doc_id + 1

    return dict_jaccard_val
```

```
def union(query,doc):
    #Calculating Union
    result = list(set(query) | set(doc))
    return result

def intersection(query,doc):
    #Calculating intersection
    result = list(set(query) & set(doc))
    return result
```

6. To calculate Jaccard coefficient, Traversing through file_data and calculating the Jaccard coefficient for the query using the above formula.
7. Finally calculating top 5 documents, with highest value of jaccard coeffecient

```
def calculate_top(jaccard_list):
    #Function to calculate top 5 documents according to the
value of jaccard coefficient.
    count=0
    jaccard_file=[]

    #Sorting the value of jaccard coef in decreasing order
    sorted_d = dict( sorted(jaccard_list.items(),
key=operator.itemgetter(1),reverse=True))
    print(sorted_d)
```

```
    for item in sorted_d.keys():
        #Finding top5 documents

        if (count != 5):
            jaccard_file.append(item)
        else:
            break
        count=count+1

    return jaccard_file
```

Results :
Query: 100 west 53 by north
Document Id: `[210, 205, 373, 391, 421]`
Documents names:
top 5 documents are:

peace.fun
snowmaid.txt
prince.art
campfire.txt
glimpse1.txt

Pros :

1. Easy to implement
2. Most basic method to compute similarity

Cons :

1. It doesn't consider term frequency i.e. how many times a term occurs in the document.
2. Complicate method for normalizing the length.
3. Usually rare terms are more informative than frequent terms which is not considered in this similarity measure.

   So we need a more sophisticated similarity measure.

2. **TF-IDF Matrix :** TF-IDF is a statistical measure which evaluates how relevant a word is to a document in a collection of documents. We multiply two metrics :
    - Number of times a word appears in a document.
    - Inverse document frequency of the word across a set of documents.

Steps:

1. Loading the Stories Dataset and performing required preprocessing steps.
2. Created file_info dictionary to store its document id and name.
3. Created file_data List of dictionaries to store each document's data i.e. there are a total 467 dictionaries are available for each document in the list.
4. Taking input for query and preprocessing it.
5. To Calculate, Inverse Document Frequency Firstly Calculating Document Frequency which is defined as the number of documents that contains term t.
6. Secondly Finding inverse document frequency: IDF(word)=log(total no. of documents/document frequency(word)+1)

```
def idf():
 #Function to find inverse document frequency

   #Firstly finding Document Frequency: Number of Documents that
contain term t.

   #doc_freq : Dictionary which contains count of number of
documents for each term in key value format.
   doc_freq = {}

   for item in file_data:
       #traversing through file_data to get count of documents for
each term.

       terms  =  list(set(item))#finding all unique ters.

       for word in terms:
           #storing count of each term in doc_freq dictionary.
           if doc_freq.get(word) is not None:
               doc_freq[word] =doc_freq.get(word)+1
           else:
               doc_freq[word] = 1
```

```
    #Secondly Finding inverse document frequency: IDF(word)=log(total
no. of documents/document frequency(word)+1)

    #Finding total number of documents
    total_docs=len(file_data)

    #inverse_doc_freq: dictionary to store IDF value.
    inverse_doc_freq = {}

    for word in doc_freq.keys():
        #Calculating IDF value for each term using a given formula.

        inverse_val=np.log((total_docs)/doc_freq[word]+1)
        inverse_doc_freq[word] = inverse_val

    return doc_freq,inverse_doc_freq
```

7. Now Created term_frequency() to Calculate Term Frequency which is defined as Number of particular terms in Document d.

```
def term_frequency():
 #Function to Calculate Term Frequency: Number of particular terms
in Document d.

    #list of list to store term frequency where sub list is created
for each document.
    final_tf=[]

    for li in file_data:
        #Traversing through file data to get each document.

        #tf: dictionary which contain key value pair in the form of
-> "term": count
        tf={}
        for term in li:
            #Traversing through each document to get count of terms
in that document.
            if tf.get(term) is not None:
                tf[term] =tf.get(term)+1
```

```
            else:
                tf[term] = 1

        final_tf.append(tf)
    return final_tf
```

8. After that Calculating term frequency for a given 5 weighting scheme.
9. Then Calculated tf-idf value which is term_frequency * inverse document frequency for all 5 weighting schemes respectively.

```
def tf_idf(freq_val):
    #Function to Calculate TF-IDF VALUE= tf*idf

    #list to store calculated tf*idf value for each term in each
document.
    final_tf_idf=[]

    for term in freq_val:
        #Traversing through each term to calculate it's tf-idf value.
        tf_idf={}
        for word in term.keys():
            tf_idf[word]=term[word]*inverse_doc_freq[word]
        final_tf_idf.append(tf_idf)

    return final_tf_idf
```

10. Finally calculating tf-idf value for query for all given 5 schemes and finding top 5 documents according to highest value of tf-idf.

```
def process_query():

    query_input=input("Enter the Input text for Query ")
    query_preprocessed=delete_spec_chars(query_input)
    tokens = word_tokenize(query_preprocessed )
    query_final = [word.lower() for word in tokens if word not in
stop_words and len(word) > 1] #Removing stopwords
    print(query_final)

    binary=calculate_top(binary_tfidf,query_final)
    raw_count=calculate_top(raw_count_tfidf,query_final)
```

```
    termfreq=calculate_top(Term_Frequency_tfidf,query_final)
    log=calculate_top(log_tfidf,query_final)
    doublelog=calculate_top(double_log_tfidf,query_final)
    return binary,raw_count,termfreq,log,doublelog

binary,raw_count,termfreq,log,doublelog=process_query()
```

Results :

Query: I will endeavour, in my statement, to avoid such terms as would serve to limit the events to any particular place, or give a clue as to the people concerned

Weighting Scheme	TF Weight	TF-IDF Score	Resulting Documents
Binary	0,1	29.3879313210617 25.0205954017978 24.4466725841418 24.0327595524623 22.4336231322488	3student.txt darkness.txt history5.txt hound-b.txt radar_ra.txt
Raw count	f(t,d)	780.890531905074 549.442025462667 358.612397045353 317.378841195747 289.114632756608	gulliver.txt vgilante.txt hound-b.txt hitch3.txt hitch2.txt
Term Frequency	f(t,d)/∑f(t`,d)	0.09465521245131 0.05419773264508 0.04135114332628 0.03955580705502 0.03661454747972	jim.asc dwar quarter.c11 sre02.txt wanderer.fun
Log Normalization	log(1+f(t,d))	52.9313373813274 46.3161730104630 41.1804739267762 39.1829189318143 37.2654888098417	gulliver.txt hound-b.txt vgilante.txt radar_ra.txt hitch2.txt
Double Normalization	0.5+0.5*(f(t,d)/ max(f(t`,d))	15.3828869071525 13.3797511171881 13.2584404198979 12.5286832005816 11.5317405051663	3student.txt history5.txt darkness.txt hound-b.txt

			radar_ra.txt

Pros :

1. You can easily compute similarity between two documents.
2. Very basic method to extract most descriptive terms in a document.

Cons :

1. Doesn't capture position in text and co-occurrences in different documents as it is a Bag of Words model.
2. Couldn't capture semantics.
3. It can only be used as a lexical level feature.

3. **Cosine Similarity :** It is a metric used to measure how similar the documents are irrespective of their size. Formula for Jaccard coefficient is as follows :

$$\text{Cosine Similarity}(x,y) = x.y \,/\, ||x|| * ||y||$$

Steps:

11. Loading the Stories Dataset and performing required preprocessing steps.
12. Created file_info dictionary to store its document id and name.
13. Created file_data List of dictionaries to store each document's data i.e. there are a total 467 dictionaries are available for each document in the list.
14. Taking input for query and preprocessing it.
15. To Calculate, Inverse Document Frequency Firstly Calculating Document Frequency which is defined as the number of documents that contains term t.
16. Secondly Finding inverse document frequency: IDF(word)=log(total no. of documents/document frequency(word)+1)
17. Now Created term_frequency() to Calculate Term Frequency which is defined as Number of particular terms in Document d.
18. After that Calculating term frequency for a given 5 weighting scheme.
19. Then Calculated tf-idf value which is term_frequency * inverse document frequency for all 5 weighting schemes respectively.
20. Taken Query input, and calculating term frequency i.e. count of term for query and storing it in a dictionary.

```
def binary_query_tfidf(query):
 # Function to find tf idf of query using binary weighting scheme

   #------------------term frequency----------------------------#

   #query_tf:Dictionary to store term frequency i.e. count of term
for query
   query_tf = {}
   for term in query:
       #Traversing through query and calculating count of each term
and storing it in a dictionary in key:value format.
       if term not in query_tf:
           query_tf[term] = 1
       else:
           query_tf[term] = query_tf[term]+1


   #--------------binary query term
frequency------------------------#

   #query_binary_tf:Dictionary to store binary term frequency i.e.
if available then 1 otherwise 0.
   #But binary term frequency will always be 1 for all the values.
   query_binary_tf= {}
   for term in query_tf.keys():
       #Calculating binary term frequency in query.
       if query_tf[term] < 0:
           query_binary_tf[term] = 0
       else:
           query_binary_tf[term] = 1


   #-------------------tfidf
value-------------------------------------#

   #Calculating tf idf value for each term of query.
   #tfidf_query: dictionary to store tf-idf value for each term of
query.
   tfidf_query = {}
```

```
    for item in query_binary_tf.keys():
        #Calculating tf idf value: tf*idf
        if item not in inverse_doc_freq  :
            tfidf_query[item] = 0.0
        else:
            tfidf_query[item] =
query_binary_tf[item]*inverse_doc_freq[item]

    return tfidf_query
```

21. Calculating term frequency according to 5 weighting schemes.
22. After that generating a query vector and normalizing it.

```
 # Function to find normalized query vector

    #-------------------query vector-------------------------#

    #query_tf_vector: list to store all tf idf value in form of
vector representation.
    query_tf_vector=[]
    for term in total_terms:
        if term not in tfidf_query  :
            query_tf_vector.append(0.0)
        else:
            query_tf_vector.append(tfidf_query [term])

    #-------------------Normalize Query
Vector----------------------#

    #normalizing query vector
    #l2 normalization
    query_vector_normalize = []

    norm = 0.0
    for term in query_tf_vector:
        norm  = norm+pow(term,2)

    # norm  = 1/np.sqrt(norm )
```

```
    #normalizing vector to calculate cosine similarity
    for term in query_tf_vector:
    #    query_vector_normalize.append(term* val )
     query_vector_normalize.append(term/np.sqrt(norm )  )

    return query_vector_normalize
```

23. Also generating document vectors for respective weighting schemes and normalizing that.

```
"""Document Normalization"""

def doc_normalize(doc_vector):
    #------------------binary doc normalized
vector----------------------#

    #list with sub list contain normalized vectors of document.
    doc__norm_vector = []

    for value in doc_vector:
        norm = 0.0
        for count in value:
            norm += pow(count,2)
        # norm = 1 / np.sqrt(norm)

        sub_list = []

        for count in value:
            # k = freq * norm
            sub_list.append(count/np.sqrt(norm))

        doc__norm_vector.append(sub_list)

    return doc__norm_vector
```

24. Creating a cosine_value dictionary to store all the cosine values with respect to document id for query using given formula.
25. At the end Finding top 5 documents according to highest cosine values.

```
def process_binary(query):
 #Function to find Cosine similarity between query and document.
   tfidf_query =binary_query_tfidf(query)
   query_vector_normalize= query_normalize(tfidf_query)
   doc_vector=binary_doc_vector()
   binary_doc__norm_vector=doc_normalize(doc_vector)

   #cosine_value: dictionary to store all the cosine values with
respect to document id for query
   cosine_value = {}

   doc_id = 0
   for item in binary_doc__norm_vector:
       #calculating cosine similarity
       doc_id += 1
       document = np.array(item)
       query= np.array(query_vector_normalize)
       cosine_value [doc_id]= sum(list(document * query))

   final_result=calculate_top( cosine_value )

   return final_result

cosine_binary=process_binary(query_final)
print("top 5 documents for binary cosine are:")
for i in cosine_binary:
   print(file_info[i])
```

26. Repeating all these steps for all the weighting schemes.

Results :

Query: I will endeavour, in my statement, to avoid such terms as would serve to limit the events to any particular place, or give a clue as to the people concerned

Weighting Scheme	TF Weight	Cosine Similarity Score	Resulting Documents

Binary	0,1	0.07880057984292 0.04432548906444 0.03580292278805 0.03562733971790 0.03407998569123	3student.txt goldenp.txt monkking.txt szechuan lament.txt
Raw count	f(t,d)	0.08680444411447 0.04287494734591 0.04192362088749 0.03453633542115 0.03432647043376	jim.asc 3student.txt gulliver.txt dwar sretrade.txt
Term Frequency	f(t,d)/∑f(t`,d)	0.08680444411447 0.04287494734591 0.04192362088749 0.03453633542115 0.03432647043376	jim.asc 3student.txt gulliver.txt dwar sretrade.txt
Log Normalization	log(1+f(t,d))	0.07765792771423 0.05570947072558 0.04089764888712 0.03923409699016 0.03818585874918	3student.txt jim.asc goldenp.txt sretrade.txt wisteria.txt
Double Normalization	0.5+0.5*(f(t,d)/ max(f(t`,d))	0.07882780358023 0.04449950240156 0.04335840492979 0.03595747520057 0.03554480138058	3student.txt goldenp.txt jim.asc monkking.txt szechuan

Pros :

1. It is good in cases where duplication exists and matters while analyzing text similarity.
2. Algorithm can be adapted to scoring document-at-a-time (DAAT)

Cons :

1. Magnitude of vectors which is dealing with term frequency does not play any role in this similarity measurement.

Question 3 : Ranked-Information Retrieval and Evaluation

Use the data file provided here. This has been taken from Microsoft learning to rank dataset, which can be found here. Read about the dataset carefully, and what all it contains.
1. Consider only the queries with qid:4 and the relevance judgement labels as relevance score.

2. [10 points] Make a file rearranging the query-url pairs in order of max DCG. State how many such files could be made.
3. [5 points] Compute nDCG
a. At 50
b. For the whole dataset
4. [10 points] Assume a model that simply ranks URLs on the basis of the value of feature 75 (sum of TF-IDF on the whole document) i.e. the higher the value, the more relevant the URL. Assume any non zero relevance judgment value to be relevant. Plot a Precision-Recall curve for query “qid:4”.

METHODOLOGY:

a. The file is iterated and the queries containing the string values are splitted by using split() function with space as the breaking point. The queries containing the qid:4 are stored in a list data.
Total length of the queries contained in the qid:4 is also printed.

Code:

```
#Loading the Dataset
path = "IR-assignment-2-data.txt"
file = open(path, 'r',encoding='ISO-8859-1')
data_doc = file.readlines()
data=[]
#Getting data only of qid:4
for x in data_doc:
    if(x.split()[1]=='qid:4'):
        data.append(x)
print("Total Qid:4 Data ", len(data))
print("Data")
print(data)
```

b.

Given, Relevance judgement labels are used as a relevance score for each query. So, highly relevant documents are more useful.

Computation of Discounted Cumulative Gain:

1. The data is iterated till the position specified. The relevance score is extracted from the query, which is used as the gain
2. The DCG is calculated using the cumulative sum of relevance/log(position), while if the position is 1 then, relevance is equal to DCG at that position.
3. So, DCG is the total gain accumulated at a particular rank p:

   $$\text{DCG at p} = \text{rel}(1) + \sum_{i=2}^{p} \frac{rel(i)}{log(i)}$$

   Where rel(i) specified the relevance score at position i.

4. The DCG value is returned.

Code:

```
#Function to find DCG
def discounted_cumulative_gain(data, count):

  dcg = 0 #Initialising the DCG
  i = 1 #Rank

  #Iterating over data
  for x in data:
    if i <= count and i != 1: #Iterate over till the position specified
      #Calculating the DCG
      relevance = int(x.split()[0])
      numerator = relevance
      denominator = math.log2(i)
      dcg = dcg + (numerator/denominator)
    elif i == 1:
      relevance = int(x.split()[0])
      dcg = relevance
    else:
      break
    i += 1
  return dcg
```

File rearranging the query-url pairs in order of max DCG.

1. A file named Q3_a.txt is opened in write mode. And the labels which is the relevant score is stored in a set to get the unique values of the relevance score.
2. The data is sorted into descending order according to the relevance score to obtain the max DCG value.

3. The data is iterated and written into the file and count of the respective relevance score is also calculated.
4. The number of files that can be obtained will be equal to the multiplication of the factorial of the count of the respective relevance score queries.

Code:

```
#Q3_a
def Q3_a(data):
    file_dcg = open("Q3_a.text", "w")
    labels = set()

    for query in data:
        #Relevance score labels
        labels.add(query.split()[0])

    sorted_dcg = data.copy()
    sorted_dcg.sort(reverse = True)

    labels = sorted(labels)

    #Initialising the number of URL pairs for each relevance score
    count_relevance = {}
    for i in labels:
        count_relevance[i] = 0
    print(count_relevance)

    #Adding data in the file sorted according to max DCG
    print("Data Stored")
    for x in sorted_dcg:
        print(x)
        file_dcg.write(x)
        count_relevance[x[0]] += 1

    #Max DCG
    max_dcg = discounted_cumulative_gain(sorted_dcg, len(sorted_dcg))
    print("Obtained max DCG", max_dcg)

    #Counting the number of files that can be made
    number_of_files = 1
    for count in count_relevance.values():
        number_of_files *= math.factorial(count)

    file_dcg.close()

    print("Number of files that can be made ", number_of_files)
    print("Count of relevance score ", count_relevance)

    return count_relevance

count_relevance = Q3_a(data)

```

c.

Methodology:

1. The DCG of the data obtained from the original data of qid:4 is calculated by specifying the position as 50 and length of the data using the function discounted_cumulative_gain() explained above.
2. For Ideal DCG, the data is sorted upto the respective position, and the DCG is calculated for the sorted data till that respective position using the function discounted_cumulative_gain() .
3. The nDCG is calculated by dividing the DCG and IDCG.

Code:

```
#Q3_b
def Q3_b(data, position):

  #Finding the DCG at the position
  dcg = discounted_cumulative_gain(data, position)

  #Sorting the data  in decreasing order according to relevance score which will the ideal rankings
  sorted_data = data.copy()
  sorted_data = sorted_data[0:position]
  sorted_data.sort(reverse = True)

  #Finding the Ideal DCG at that position
  idcg = discounted_cumulative_gain(sorted_data, position)

  #Calculating the Normalised DCG
  ndcg = dcg/idcg

  if position == len(data):
    print("\nWhole Dataset ")
  else:
    print("\nPosition at ", position)

  print("DCG: ", dcg)
  print("IDCG: ", idcg)
  print("NDCG: ", ndcg)

Q3_b(data, 50)
Q3_b(data, len(data))
```

d.

Methodology:

1. A dictionary data_feature_75 stores the query as key and the 75th feature value, extracted from the data as the value for all the queries.
2. The data is sorted in descending order according to the value of the 75th feature. As given that higher the value, the more relevant is the URL.
3. The total number of relevant documents are the ones which do not contain the relevance score as 0. So, the subtraction of the total documents and the documents with relevant scores is 0.
4. Now, for each query, the precision and recall is calculated. As precision is equal to current_relevant_docs/total_retrieved_docs and

recall is equal to current_relevant_docs/total_relevant_docs.

5. For every query, the total retrieved docs will be the position of the query at which we are and the current relevant docs is equal to the number of documents whose relevance score is not equal to 0 till the respective position. The precision and recall is calculated and stored in the list.
6. The plot is created between the precision and recall.

Code:

```
#Q3_c
def Q3_c(data):

  #Extracting the feature 75 values
  data_feature_75 = {}
  for query in data:
    value = query.split()[76]
    value = value[3:]
    data_feature_75[query] = float(value)

  #Sorting the data according to the 75 value
  sorted_values = sorted(data_feature_75.items(), key=lambda item: item[1], reverse = True)
  sorted_dcg_dict = {k: v for k, v in sorted_values}

  #Total relevant docs exclude the one with relevance score as 0
  total_relevant_docs = len(data) - count_relevance['0']
  total_retrieved_docs = 0

  precision = []
  recall = []

  #Iterating over data
  current_retrieved_docs = 0
  for query in sorted_dcg_dict:
    #If relevant score is 0 then not a relevant document
    if query.split()[0] != '0':
      current_retrieved_docs += 1

    total_retrieved_docs += 1

    #Finding the precision and recall
    precision_value = current_retrieved_docs/total_retrieved_docs
    recall_value = current_retrieved_docs/total_relevant_docs
    precision.append(precision_value)
    recall.append(recall_value)

  print("Precision Recall Curve")
  #Plotting Precision vs Recall
  plt.plot(recall, precision)
  plt.xlabel('Recall')
  plt.ylabel('Precision')
  plt.title('A Precision-Recall Curve')
  plt.show()


Q3_c(data)
```