Information Retrieval Report

Data Preprocessing—----------

i) Relevant Text Extraction:

Opened the original file and the file is parsed using BeautifulSoup, then the text with tags <Title> and <Text> is retrieved from the file and all the text is discarded from the file and then the empty file is appended with the text which we have retrieved early on.

Code snippet of the above operation—:

```
import os
from bs4 import BeautifulSoup
import nltk
dire = 'C:/Users/91775/Downloads/CSE508_Winter2023_Dataset3/CSE508_Winter2023_Dataset/'
i=1;
for filename in os.listdir(dire):
    f = os.path.join(dire, filename) # joining the file name with file..
    file1 = open(f)
    soup = file1.read()# reading the file 1..
    bs = BeautifulSoup(soup,"html.parser") # parsing the text in file1..
    text = bs.find('text').text # finding the text with <Text> in the given file
    title = bs.find('title').text # finding the text with <Title> in the given file
    texti = title+text # cocatenating the <Title> and <Text> string..
    file1.close();
    if i<6:
        print("-------------------------"+"file with tags:" + filename+"-----------------------")
        print(soup)
        print("-------------------------"+"file after extracting text from tags <Title> and <Text>:" + filename+"---------
        print(texti)
    file2 = open(f, 'r+')
    file2.truncate(0) # Discarding the text from the file...
    file2.close();
    file3 = open(f,'w') # openig file file in writing mode..
    file3.write(texti) # write the cocatenated string into the file..
    i=i+1;
    file3.close();
```

Output:

```
<DOC>
<DOCNO>
1
</DOCNO>
<TITLE>
experimental investigation of the aerodynamics of a
wing in a slipstream .
</TITLE>
<AUTHOR>
brenckman,m.
</AUTHOR>
<BIBLIO>
j. ae. scs. 25, 1958, 324.
</BIBLIO>
<TEXT>
  an experimental study of a wing in a propeller slipstream was
made in order to determine the spanwise distribution of the lift
increase due to slipstream at different angles of attack of the wing
and at different free stream to slipstream velocity ratios .  the
results were intended in part as an evaluation basis for different
theoretical treatments of this problem .
  the comparative span loading curves, together with supporting
evidence, showed that a substantial part of the lift increment
produced by the slipstream was due to a /destalling/ or boundary-layer-control
effect .  the integrated remaining lift increment,
after subtracting this destalling lift, was found to agree
well with a potential flow theory .
  an empirical evaluation of the destalling effects was made for
the specific configuration of the experiment .
</TEXT>
</DOC>

-------------------------file after extracting text from tags <Title> and <Text>:cranfield0001----------------------

experimental investigation of the aerodynamics of a
wing in a slipstream .

  an experimental study of a wing in a propeller slipstream was
made in order to determine the spanwise distribution of the lift
```

ii) Preprocessing:

Lowercase the text:

Opened the previously modified file and retrieved the text and covert every letter into lowercase and store it into the string. Then all the text is discarded from the opened file and then the empty file is appended with the above string.

Code snippet of the above operation—:

```
dire = 'C:/Users/91775/Downloads/CSE508_Winter2023_Dataset3/CSE508_Winter2023_Dataset/'
i=1
for filename in os.listdir(dire):
    f = os.path.join(dire, filename) # joining the file name with file..
    file1 = open(f,'r')
    txt = file1.read()
    #print(txt)
    txt_l = txt.lower()
    file1.close()
    if i<6:
        print("-------------------------""file before going into lowercase preprocessing:" + filename+"----------------------")
        print(txt)
        print("-------------------------""file after going into lowercase preprocessing:" + filename+"----------------------")
        print(txt_l)
    file2 = open(f, 'r+')
    file2.truncate(0) # Discarding the text from the file...
    file2.close()
    file3 = open(f,'w')
    file3.write(txt_l)
    i = i+1
    file3.close()
```

Output:

```
-------------------------file before going into lowercase preprocessing:cranfield0001----------------------

experimental investigation of the aerodynamics of a
wing in a slipstream .

  an experimental study of a wing in a propeller slipstream was
made in order to determine the spanwise distribution of the lift
increase due to slipstream at different angles of attack of the wing
and at different free stream to slipstream velocity ratios .  the
results were intended in part as an evaluation basis for different
theoretical treatments of this problem .
  the comparative span loading curves, together with supporting
evidence, showed that a substantial part of the lift increment
produced by the slipstream was due to a /destalling/ or boundary-layer-control
effect .  the integrated remaining lift increment,
after subtracting this destalling lift, was found to agree
well with a potential flow theory .
  an empirical evaluation of the destalling effects was made for
the specific configuration of the experiment .

-------------------------file after going into lowercase preprocessing:cranfield0001----------------------

experimental investigation of the aerodynamics of a
wing in a slipstream .

  an experimental study of a wing in a propeller slipstream was
made in order to determine the spanwise distribution of the lift
increase due to slipstream at different angles of attack of the wing
and at different free stream to slipstream velocity ratios .  the
results were intended in part as an evaluation basis for different
theoretical treatments of this problem .
```

Perform Tokenization:

Opened the previously modified file and retrieved the text and tokenize it using nltk and then join the token into string. Then all the text is discarded from the opened file and then the empty file is appended with the above string.

Code snippet of the above operation—:

```
dire = 'C:/Users/91775/Downloads/CSE508_Winter2023_Dataset3/CSE508_Winter2023_Dataset/'
i=1
for filename in os.listdir(dire):
    f = os.path.join(dire, filename) # joining the file name with file..
    file1 = open(f,'r')
    txt = file1.read()
    txt_tk = nltk.word_tokenize(txt)
    file1.close()
    if i<6:
        str_tk = ""
        print("\n")
        print("-------------------------"+"file before tokenization preprocessing:" + filename+"----------------------")
        print(txt)
        print("\n")
        print("**************************"+"file after tokenization preprocessing:" + filename+"**********************\n")
        print(txt_tk)
        print(type(txt_tk))


    #print(txt_tk)
    str_tk  = ' '.join(str(token) for token in txt_tk)
        #print(str_tk)
    file2 = open(f, 'r+')
    file2.truncate(0) # Discarding the text from the file...
    file2.close();
    file3 = open(f, 'w')
```

Output:

```
-------------------------file before tokenization preprocessing:cranfield0001----------------------

experimental investigation of the aerodynamics of a
wing in a slipstream .

  an experimental study of a wing in a propeller slipstream was
made in order to determine the spanwise distribution of the lift
increase due to slipstream at different angles of attack of the wing
and at different free stream to slipstream velocity ratios .  the
results were intended in part as an evaluation basis for different
theoretical treatments of this problem .
  the comparative span loading curves, together with supporting
evidence, showed that a substantial part of the lift increment
produced by the slipstream was due to a /destalling/ or boundary-layer-control
effect .  the integrated remaining lift increment,
after subtracting this destalling lift, was found to agree
well with a potential flow theory .
  an empirical evaluation of the destalling effects was made for
the specific configuration of the experiment .



**************************file after tokenization preprocessing:cranfield0001**********************

['experimental', 'investigation', 'of', 'the', 'aerodynamics', 'of', 'a', 'wing', 'in', 'a', 'slipstream', '.', 'an', 'experime
ntal', 'study', 'of', 'a', 'wing', 'in', 'a', 'propeller', 'slipstream', 'was', 'made', 'in', 'order', 'to', 'determine', 'th
e', 'spanwise', 'distribution', 'of', 'the', 'lift', 'increase', 'due', 'to', 'slipstream', 'at', 'different', 'angles', 'of',
'attack', 'of', 'the', 'wing', 'and', 'at', 'different', 'free', 'stream', 'to', 'slipstream', 'velocity', 'ratios', '.', 'th
e', 'results', 'were', 'intended', 'in', 'part', 'as', 'an', 'evaluation', 'basis', 'for', 'different', 'theoretical', 'treatme
nts', 'of', 'this', 'problem', '.', 'the', 'comparative', 'span', 'loading', 'curves', ',', 'together', 'with', 'supporting',
'evidence', '.', 'showed', 'that', 'a', 'substantial', 'part', 'of', 'the', 'lift', 'increment', 'produced', 'by', 'the', 'slip
```

Remove stopwords:

Opened the previously modified file and retrieved the text and remove the sopwords using nltk library an stored it into the string Then all the text is discarded from the opened file and then the empty file is appended with the above string.

Code snippet of the above operation—:

```
import nltk
from nltk.corpus import stopwords
stopword = stopwords.words('english')
dire = 'C:/Users/91775/Downloads/CSE508_Winter2023_Dataset3/CSE508_Winter2023_Dataset/'
i=1
for filename in os.listdir(dire):
    f = os.path.join(dire, filename) # joining the file name with file..
    file1 = open(f,'r')
    txt = file1.read()
    txt_tk = nltk.word_tokenize(txt)
    rem_stop = [s_word for s_word in txt_tk if s_word not in stopword]
    str_stop  = ' '.join(str(token) for token in rem_stop)
    file1.close()
    #print (removing_stopwords)
    if i<6:
        print("\n")
        print("-------------------------"+"file before removing stopwords:" + filename+"----------------------")
        print(txt)
        print("\n")
        print("**************************"+"file after removing stopwords:" + filename+"**********************\n")
        print(str_stop)
    file2 = open(f, 'r+')
    file2.truncate(0) # Discarding the text from the file...
    file2.close()
    file3 = open(f,'w')
    file3.write(str_stop)
    i = i+1
    file3.close()
```

Output:

```
-------------------------file before removing stopwords:cranfield0001----------------------
experimental investigation of the aerodynamics of a wing in a slipstream . an experimental study of a wing in a propeller slips
tream was made in order to determine the spanwise distribution of the lift increase due to slipstream at different angles of at
tack of the wing and at different free stream to slipstream velocity ratios . the results were intended in part as an evaluatio
n basis for different theoretical treatments of this problem . the comparative span loading curves , together with supporting e
vidence , showed that a substantial part of the lift increment produced by the slipstream was due to a /destalling/ or boundary
-layer-control effect . the integrated remaining lift increment , after subtracting this destalling lift , was found to agree w
ell with a potential flow theory . an empirical evaluation of the destalling effects was made for the specific configuration of
the experiment .


**************************file after removing stopwords:cranfield0001**********************

experimental investigation aerodynamics wing slipstream . experimental study wing propeller slipstream made order determine spa
nwise distribution lift increase due slipstream different angles attack wing different free stream slipstream velocity ratios .
results intended part evaluation basis different theoretical treatments problem . comparative span loading curves , together su
pporting evidence , showed substantial part lift increment produced slipstream due /destalling/ boundary-layer-control effect .
integrated remaining lift increment , subtracting destalling lift , found agree well potential flow theory . empirical evaluati
on destalling effects made specific configuration experiment .
```

Remove punctuation:

Opened the previously modified file and retrieved the text and remove the puntuation using regex library an stored it into the string, then all the text is discarded from the opened file and then the empty file is appended with the above string.

Code snippet of the above operation—:

```
import re
import nltk
dire = 'C:/Users/91775/Downloads/CSE508_Winter2023_Dataset3/CSE508_Winter2023_Dataset/'
i=1
for filename in os.listdir(dire):
    f = os.path.join(dire, filename) # joining the file name with file..
    file1 = open(f,'r')
    txt = file1.read()
    txt_punc = re.sub(r'[^\w\s]','',txt)
    file1.close()
    if i<6:
        print("\n")
        print("-------------------------"+"file before removing punctuations:" + filename+"----------------------")
        print(txt)
        print("\n")
        print("**************************"+"file after removing punctuations:" + filename+"**********************\n")
        print(txt_punc)
    file2 = open(f, 'r+')
    file2.truncate(0) # Discarding the text from the file...
    file2.close()
    file3 = open(f,'w')
    file3.write(txt_punc)
    i = i+1
    file3.close()
```

Output:

```
-------------------------file before removing punctuations:cranfield0001----------------------
experimental investigation aerodynamics wing slipstream . experimental study wing propeller slipstream made order determine spa
nwise distribution lift increase due slipstream different angles attack wing different free stream slipstream velocity ratios .
results intended part evaluation basis different theoretical treatments problem . comparative span loading curves , together su
pporting evidence , showed substantial part lift increment produced slipstream due /destalling/ boundary-layer-control effect .
integrated remaining lift increment , subtracting destalling lift , found agree well potential flow theory . empirical evaluati
on destalling effects made specific configuration experiment .


**************************file after removing punctuations:cranfield0001**********************

experimental investigation aerodynamics wing slipstream  experimental study wing propeller slipstream made order determine span
wise distribution lift increase due slipstream different angles attack wing different free stream slipstream velocity ratios  r
esults intended part evaluation basis different theoretical treatments problem  comparative span loading curves  together suppo
rting evidence  showed substantial part lift increment produced slipstream due destalling boundarylayercontrol effect  integrat
ed remaining lift increment  subtracting destalling lift  found agree well potential flow theory  empirical evaluation destalli
ng effects made specific configuration experiment
```

Remove blank space tokens:

Opened the previously modified file and retrieved the text and remove the blank space tokens using replace function and stored it into the string, then all the text is discarded from the opened file and then the empty file is appended with the above string.

Code snippet of the above operation—:

```
dire = 'C:/Users/91775/Downloads/CSE508_Winter2023_Dataset3/CSE508_Winter2023_Dataset/'
i=1
for filename in os.listdir(dire):
    f = os.path.join(dire, filename) # joining the file name with file..
    file1 = open(f,'r')
    txt = file1.read()
    #txt_tk = nltk.word_tokenize(txt)
    txt_b = txt.replace("  "," ")
    file1.close()
    if i<6:
        print("\n")
        print("-------------------------"+"file before removing blank spaces:" + filename+"----------------------")
        print(txt)
        print("\n")
        print("**************************"+"file after removing blank spaces:" + filename+"**********************\n")
        print(txt_b)
    file2 = open(f, 'r+')
    file2.truncate(0) # Discarding the text from the file...
    file2.close()
    file3 = open(f,'w')
    file3.write(txt_b)
    i = i+1
    file3.close()
```

Output:

```
-------------------------file before removing blank spaces:cranfield0001----------------------
experimental investigation aerodynamics wing slipstream  experimental study wing propeller slipstream made order determine span
wise distribution lift increase due slipstream different angles attack wing different free stream slipstream velocity ratios  r
esults intended part evaluation basis different theoretical treatments problem  comparative span loading curves  together suppo
rting evidence  showed substantial part lift increment produced slipstream due destalling boundarylayercontrol effect  integrat
ed remaining lift increment  subtracting destalling lift  found agree well potential flow theory  empirical evaluation destalli
ng effects made specific configuration experiment


**************************file after removing blank spaces:cranfield0001**********************

experimental investigation aerodynamics wing slipstream experimental study wing propeller slipstream made order determine spanw
ise distribution lift increase due slipstream different angles attack wing different free stream slipstream velocity ratios res
ults intended part evaluation basis different theoretical treatments problem comparative span loading curves together supportin
g evidence showed substantial part lift increment produced slipstream due destalling boundarylayercontrol effect integrated rem
aining lift increment subtracting destalling lift found agree well potential flow theory empirical evaluation destalling effect
s made specific configuration experiment
```

1.

Import required libraries

```
import numpy as np
import pandas as pd
```

```
#preprocessing
import nltk
import re
from nltk.corpus import stopwords
nltk.download('stopwords')
stopword = stopwords.words('english')
nltk.download('punkt')
```

We have created a preprocess function to perform preprocessing over the input data provided by the user for query purposes. All the preprocessing done in question 1 is also done on input data.

```
def preprocess(sequence):
  sequence = sequence.lower()           #convert to lowercase
  sequence = re.sub(r'[^\w\s]','',sequence)     #remove punctuation
  sequence = nltk.word_tokenize(sequence)          #perform tokenization
  rem_stop = [s_word for s_word in sequence if s_word not in stopword]      #remove stopwords
  sequence  = ' '.join(str(token) for token in rem_stop)
  sequence = sequence.replace("  "," ")         #remove extra space

  return sequence
```

Read the files and store the data structure created along with the list of filenames

```
data_structure = []           #using 2D array as a datastructure
file_list=[]                  #stores all file names

arr=[]

for i in range(1, 1401):      #There are a total of 1400 files to access

  fname = "cranfield"         #file name starts with "cranfield"

  #Filename consists of file number. Since the number if of 4 digit we add corresponding zeros in the begining
  if(i>=1 and i<10):
    fname = fname + "000" + str(i)
  elif(i>=10 and i<100):
    fname = fname + "00" + str(i)
  elif(i>=100 and i<1000):
    fname = fname + "0" + str(i)
  else:
    fname = fname + str(i)

  file_list.append(fname)
```

For every word present in the file add it to list_of_words. It is a dictionary with the key being the word and the value being its count.
For the Binary weighing scheme, the value is 1 if the word is present in the file. If the word is not present in the file then the value for that word is zero.

```
list_of_words ={}

with open(fname) as f:       #open the corresponding file
  while True:
      l = f.readline()        #read line by line
      if not l:               #if line dosent exists means its EOF, hence get out of loop
          break
      else:
        for w in l.split():   #for every word in that line
          if w not in list_of_words:
            list_of_words[w] = 1
```

If the word appears in file number k for the first time (which means it is not present in the universal list of words), then the values for previous k-1 files for that should be zero. This is done via zeroArray. zeroArray contains the zeros to be appended for new words. And for the remaining elements(words) in the data structure we append zero for them(for that particular document)

```
for k,v in list_of_words.items():
  if k not in arr:
    arr.append(k)
    zeroArray = []
    if i!=1:
      zeroArray = [0] * (i-1)
    zeroArray.append(v)
    data_structure.append(zeroArray)
  else:
    indexofElement = arr.index(k)
    data_structure[indexofElement].append(v)
    for ele in data_structure:
      if len(ele)<i-1:
        ele.append(0)
```

===> Raw Count
Everything remains the same, instead of 1 and 0 we store the count of the word.

```
with open(fname) as f:       #open the corresponding file
  while True:
      l = f.readline()        #read line by line
      if not l:               #if line dosent exists means its EOF, hence get out of loop
          break
      else:
        for w in l.split():   #for every word in that line
          if w not in list_of_words:
            list_of_words[w] = 1

          list_of_words[w] = list_of_words[w] + 1
```

===> Term Frequency

After obtaining the count for the word we divide the respective word by the total number of words in the document.

```
with open(fname) as f:       #open the corresponding file
  while True:
      l = f.readline()        #read line by line
      if not l:               #if line dosent exists means its EOF, hence get out of loop
          break
      else:
        for w in l.split():   #for every word in that line
          wordCount = wordCount + 1

          if w not in list_of_words:
            list_of_words[w] = 1

          list_of_words[w] = list_of_words[w] + 1


for k,v in list_of_words.items():
  list_of_words[k] = list_of_words[k]/wordCount
```

===> Log Normalization

After obtaining the count for respective words apply log normalization on it.

```
with open(fname) as f:       #open the corresponding file
  while True:
      l = f.readline()        #read line by line
      if not l:               #if line dosent exists means its EOF, hence get out of loop
          break
      else:
        for w in l.split():   #for every word in that line
          if w not in list_of_words:
            list_of_words[w] = 1

          list_of_words[w] = list_of_words[w] + 1

for k,v in list_of_words.items():
  list_of_words[k] = np.log(1 + v)
```

===> Double Normalization

===> IDF calculation

IDF calculation will be the same on all the weighting schemes. It is the log of the total number of documents divided by the number of documents in which the word occurred. This we obtain for all the unique words and add it to IDF list

```
IDF = []
for i in range(0, len(data_structure)):
  res = len(np.nonzero(data_structure[i])[0])
  IDF.append(np.log(1400/res))

return IDF
```

===> TFIDF Calculation

To obtain TFIDF data structure just multiply the the TF structure with IDF list

```
for i in range(0, len(data_structure)):
  data_structure[i] = list(np.array(data_structure[i]) * IDF[i])

return data_structure
```

===> Matrix

Here we convert the TFIDF data structure to a data frame for mathematical use

```
def convertToDataframe(data_structure, file_list, arr):
  df = pd.DataFrame(data_structure, columns =file_list)
  df['cranfield1400'] = df['cranfield1400'].fillna(0)
  return df
```

===> Matrix with words

Here words are added so that we can keep easy track for every word.

```
def dataframeWithWords(data_structure, file_list, arr):
  df = pd.DataFrame(data_structure, columns =file_list)
  df.insert(loc = 0, column = 'words', value = arr)
  df['cranfield1400'] = df['cranfield1400'].fillna(0)
  return df
```

===> function calls

A series of function calls are made for all the weighting schemes to obtain the data structures. First, we calculate the TF data structure, followed by the list of IDF values for every word, then we obtain the product of both to get the TFIDF data structure. TDIDF structure obtained is then converted into a data frame for further calculations. Below is an example of binary weighting scheme.

```
# --------------------------------- Binary ---------------------------------
data_structure_Binary, file_list, arr = BinaryTFCalculation()
IDF_Binary = BinaryIDFCalculation(data_structure_Binary)
TFIDF_Binary = BinaryTFIDFCalculation(data_structure_Binary, IDF_Binary)
TFIDF_Matrix_Binary = convertToDataframe(data_structure_Binary, file_list, arr)
TFIDF_WordMatrix_Binary = dataframeWithWords(data_structure_Binary, file_list, arr)
```

===> Jaccard

For this we are using TFIDF matrix for the binary weighting scheme. We convert the matrix to a numpy array for mathematical use. similarityArray contains the jaccard score for all the query with respect to all the documents in a key value pair. Where the keys are the filenames and the value is the score.

```
def JaccardCoefficient(query):

  similarityArray = {}

  data_structure_for_Jaccard = list(TFIDF_Matrix_Binary.to_numpy())

  for i in range(0, 1400):

    intersection = 0
    DocSet = 0

    wordList = query.split()
    wordSet = len(wordList)
```

```
    for j in wordList:
      if j in arr:
        indexofElement = arr.index(j)
        val = data_structure_for_Jaccard[indexofElement][i]
        if val!=0:
          intersection = intersection + 1

    for  j in range(0, len(data_structure_for_Jaccard)):
      val = data_structure_for_Jaccard[j][i]
      if val!=0:
        DocSet = DocSet + 1
```

We then calculate Jaccard score by taking intersection/union for all the files and then pick the top five with highest score and return them.

```
    union = wordSet + DocSet - intersection

    JaccardValue = intersection / union

    similarityArray[file_list[i]] = JaccardValue


  similarityArray = dict(sorted(similarityArray.items(), key=lambda item: item[1], reverse = True))
  topFive = list(similarityArray)[0:5]
  finalList ={}
  for ele in topFive:
    finalList[ele] = similarityArray[ele]
  return finalList
```

===> Top Five

This function returns the top five files which are most related to the query for TFIDF and also displays them.

```
def topfive(similarityArray, scheme):
  similarityArray = dict(sorted(similarityArray.items(), key=lambda item: item[1], reverse = True))
  topFive = list(similarityArray)[0:5]
  finalList ={}
  for ele in topFive:
    finalList[ele] = similarityArray[ele]

  print()
  print('---------------------------------------- Weighting Scheme:  ' + scheme + ' ----------------------------------------')
  print(finalList)
  print()
  print()
```

===> Driver code: Jaccard

```
query = input("Enter your query   ===>  ")
if query =="":
  print("cannot perform any operation on empty query. Please provide some input")
else:
  query = preprocess(query)
  JaccardList = JaccardCoefficient(query)
  print(JaccardList)
```

===> Driver Code: TFIDF

Once we obtain the query we preprocess it. Convert the TFIDF matrices obtained for all the weighting schemes into a numpy array for easy calculation and store them into different arrays respectively.

```
query = input("Enter your query   ===>  ")
if query =="":
  print("cannot perform any operation on empty query. Please provide some input")
else:
  query = preprocess(query)
  binary = list(TFIDF_Matrix_Binary.to_numpy())
  raw = list(TFIDF_Matrix_RawCount.to_numpy())
  term_frequency = list(TFIDF_Matrix_TermFrequency.to_numpy())
  log_norm = list(TFIDF_Matrix_LogNormalization.to_numpy())
  double_norm = list(TFIDF_Matrix_DoubleNormalization.to_numpy())
```

Score Dictionaries (eg. score_Binary, score_raw, …etc) are used to keep track of the TFIDF scores of the query for every document. The val list (eg. val_Binary, val_raw ….. etc) holds the TFIDF values for the word from the TFIDF matrices.

```
score_Binary ={}
score_raw ={}
score_term_frequency ={}
score_log_norm ={}
score_double_norm ={}

wordList = query.split()
```

```
for i in range(0, 1400):

  val_Binary =[]
  val_raw =[]
  val_term_frequency =[]
  val_log_norm =[]
  val_double_norm =[]

  for j in wordList:

    if j in arr:
      indexofElement = arr.index(j)
      val_Binary.append(binary[indexofElement][i])
      val_raw.append(raw[indexofElement][i])
      val_term_frequency.append(term_frequency[indexofElement][i])
      val_log_norm.append(log_norm[indexofElement][i])
      val_double_norm.append(double_norm[indexofElement][i])
```

These values are then used to obtain the overall score for the query (for this we have used some temporary variables called temp_Binary, … etc.). Post obtaining the scores for the document add them to the score dictionaries with the key being the document name and the value being the score of the query for that document.

```
temp_binary = 0
temp_raw = 0
temp_term_frequency = 0
term_log_norm = 0
term_double_norm = 0
for count in range(0, len(val_Binary)):
  temp_binary = temp_binary + val_Binary[count]
  temp_raw = temp_raw + val_raw[count]
  temp_term_frequency = temp_term_frequency + val_term_frequency[count]
  term_log_norm = term_log_norm + val_log_norm[count]
  term_double_norm = term_double_norm + val_double_norm[count]
```

Then call the topfive function and obtain the top five documents with highest similarity

```
    score_Binary[file_list[i]] = temp_binary
    score_raw[file_list[i]] = temp_raw
    score_term_frequency[file_list[i]] = temp_term_frequency
    score_log_norm[file_list[i]] = term_log_norm
    score_double_norm[file_list[i]] = term_double_norm

topfive(score_Binary, 'Binary')
topfive(score_raw, 'Raw Count')
topfive(score_term_frequency, 'Term Frequency')
topfive(score_log_norm, 'Log Normalization')
topfive(score_double_norm, 'Double Normalization')
```

===>output:

```
Enter your query   ===>  takeoff conformal circulation tapered spaced table incidences

------------------------------------------- Weighting Scheme:  Binary -------------------------------------------
{'cranfield0246': 10.298800289180274, 'cranfield0452': 10.298800289180274, 'cranfield0249': 9.924106839738865, 'cranfield1162': 9.89333518107211, 'cranfield0315': 9.692664485609958}


------------------------------------------- Weighting Scheme:  Raw Count -------------------------------------------
{'cranfield0246': 26.050068624735843, 'cranfield1162': 24.83367330041135, 'cranfield0793': 24.231661214024896, 'cranfield0452': 20.59760057836055, 'cranfield1095': 20.188011753068523


------------------------------------------- Weighting Scheme:  Term Frequency -------------------------------------------
{'cranfield0778': 0.4199400763954721, 'cranfield1358': 0.41318287791388114, 'cranfield1357': 0.35619213613265616, 'cranfield0246': 0.3339752387786647, 'cranfield1162': 0.331115644005


------------------------------------------- Weighting Scheme:  Log Normalization -------------------------------------------
{'cranfield0246': 12.882965863790488, 'cranfield1162': 12.320871870789274, 'cranfield0793': 12.042683217228086, 'cranfield0452': 11.314388556232132, 'cranfield0249': 10.902745728192


------------------------------------------- Weighting Scheme:  Double Normalization -------------------------------------------
{'cranfield0793': 0.002679528517008189, 'cranfield0246': 0.0026795285170081885, 'cranfield1162': 0.0026497559779303202, 'cranfield0315': 0.0025604383606967138, 'cranfield0452': 0.002
```

Pros and Cons

1. Binary ⇒ It is used when we only care about the presence of the word in the document and frequency is not important. It is easy to calculate and daily simple. The drawback is that for some tasks frequency plays an important role which is ignored in this scheme.
2. Raw Count ⇒ used when we care about the impact of the frequency of the term in the query. It is easy to calculate. Here the drawback is it gives too much importance at times to certain words and ignores the inverse document frequency. It ignores the variation in the frequency of the word across all documents.
3. Term Frequency ⇒ It gives probabilistic weight to the words in document instead of the count. This also ignores the variation in the frequency of the word across all documents.

4. Log Normalization ⇒ It is used to nullify the effect of large documents. Here term may occur multiple times and it may not be of any use. It is not much impactful for documents of smaller size.
5. Double Normalization ⇒ There might be some cases where the document lengths vary to a large extent and some terms occur way many times as compared to the rest. To deal with this we use double normalization. It may need a normalization constant which depends on the average length of the documents.

Hence, as per the above analysis Log Normalization is best for our analysis.

2. Naive Bayes Classifier with TF-ICF

Intially the required libraries are imported , the train dataset is loaded, then removed the unnecessary columns, and checked for null values are present in any of the columns.

1. Preprocessing the dataset :

```
[65] # Preprocessing the given dataset by removing punctuations,converting all the characters into lower,
     # doing tokenization and removing the stopwords, performing the stemming and making a list. Finally return the list to the calling function.
     def preprocess(row):
       text = row[0]
       text = re.sub(r'[^\w\s]','',text)
       text = text.lower()
       text = nltk.word_tokenize(text)
       text = [s_word for s_word in text if s_word not in stopword]
       ps = PorterStemmer()
       text = [ps.stem(word) for word in text]
       return text
```

```
[66] trainset['updated_text'] = trainset.apply(preprocess, axis=1)     # adding the new column to the trainset by updating the 'Text' column data
```

In this code, the preprocessing is done on the ‘Text’ column in the train dataset and updated in the new column ‘updated_text’.

The preprocessing steps are

- Punctuations are removed
- Made all the words in the text to lowercase
- Perfomed tokenization
- Removed stopwords from the tokenized words
- Performed stemming on the words

After preprocessing, the updated_text column contains list of words which are preprocessed.

The below image is the text before preprocessing -

```
trainset.iloc[1]['Text']    # displaying the content from the train dataset 1st row 'Text' column
```

```
'german business confidence slides german business confidence fell in february knocking hopes of a speedy recovery in europe s largest economy.  munic
h-based research institute ifo said that its confidence index fell to 95.5 in february from 97.5 in january  its first decline in three months. the st
udy found that the outlook in both the manufacturing and retail sectors had worsened. observers had been hoping that a more confident business sector
would signal that economic activity was picking up.  we re surprised that the ifo index has taken such a knock  said dz bank economist bernd weidens
teiner.  the main reason is probably that the domestic economy is still weak  particularly in the retail trade.  economy and labour minister wolfgang
clement called the dip in february s ifo confidence figure  a very mild decline . he said that despite the retreat  the index remained at a relatively
high level and that he expected  a modest economic upswing  to continue.  germany s economy grew 1.6% last year after shrinking in 2003. however  the
economy contracted by 0.2% during the last three months of 2004  mainly due to the reluctance of consumers to spend. latest indications are that growt
h is still proving elusive and ifo president hans-werner sinn said any improvement in german domestic demand was sluggish. exports had kept things goi
ng during the first half of 2004  but demand for exports was then hit as the value of the euro hit record levels making german products less competiti
ve overseas. on top of that  the unemployment rate has been stuck at close to 10% and manufacturing firms  including daimlerchrysler  siemens and volk
swagen  have been negotiating with unions over cost cutting measures. analysts said that the ifo figures and germany s continuing problems may delay a
n interest rate rise by the european central bank. eurozone interest rates are at 2%  but comments from senior officials have recently focused on the
threat of inflation  prompting fears that interest rates may rise.'
```

The below image is the text after preprocessing -

```
[67] trainset.iloc[1]['updated_text']       # displaying the content from the train dataset 1st row 'Text' column after preprocessing is done
```

```
['german',
 'busi',
 'confid',
 'slide',
 'german',
 'busi',
 'confid',
 'fell',
 'februari',
 'knock',
 'hope',
 'speedi',
 'recoveri',
 'europ',
 'largest',
 'economi',
 'munichbas',
 'research',
 'institut',
 'ifo',
 'said',
 'confid',
 'index',
 'fell'
```

TF implementation -

```
[69] # term_frequency is a dictionary. The keys are different categories and the values are another dictionary wh
     term_frequency = {}
     for category in categories:
       term_frequency[category] = {}
     print(term_frequency)
```

```
{'sport': {}, 'entertainment': {}, 'politics': {}, 'business': {}, 'tech': {}}
```

```
[70] unique_words = []    # this list contain the list of unique word in the entire dataset
     total_words_without_repetition = 0   # this variable will have the sum of number of unique words row by row.

     # updating the term_frequency dictionary
     for i in range(len(trainset)):
       category = trainset['Category'][i]
       total_words_without_repetition += len(set(trainset['updated_text'][i]))
       for term in trainset['updated_text'][i]:
         if term in term_frequency[category]:
           term_frequency[category][term] += 1
         else:
           term_frequency[category][term] = 1
         unique_words.append(term)
```

term_frequency is a dictionary. The keys are different categories and the values are another dictionary where the keys in that dictionary are words belonging to that category and values are count of that word in that category .

CF implementation -

```
# CF_list is a dictionary where keys are unique words and values are count of each word in how many categories it is present
CF_list = {}
for word in unique_words:
  count=0
  for category in categories:
    if word in term_frequency[category]:
      count+=1
  CF_list[word] = count
```

```
[77] print(CF_list)    # displaying the CF_list
```

```
{'plaid': 1, '161strong': 1, '2008': 4, 'liu': 2, 'aw': 1, 'leeway': 1, '22yearold': 3, 'guilti': 5, 'burgeon': 2, 'ambuja': 1,
```

CF_list is a dictionary where keys are unique words in the train dataset ‘updated_text’ column and values are count of each word in how many categories it is present.

ICF implementation -

```
import math

# ICF_list is a dictionary where the keys are unique words and the values are log(N/cf[word]) where N is the no. of categories
N = len(categories)
ICF_list = {}
for word in CF_list:
  ICF_list[word] = math.log10(N / CF_list[word])
```

```
[80] print(ICF_list)

{'plaid': 0.6989700043360189, '161strong': 0.6989700043360189, '2008': 0.09691001300805642, 'liu': 0.3979400086720376, 'aw': 0.6989700043360189,
```

ICF_list is a dictionary where the keys are unique words and the values are log(N/CF_list[word]) where N is the no. of categories.

TF-ICF implementation -

```
[82] # tf_icf is just multiply term frequency and icf. But it should be done category wise. That why I used dictionary of dictionary
     tf_icf = {}
     for category in categories:
       tf_icf[category] = {}
     for category in categories:
       for word in term_frequency[category].keys():
         tf_icf[category][word] = term_frequency[category][word] * ICF_list[word]
```

Tf_icf is a dictionary where the keys are different categories and the values are another dictionary where the keys in that dictionary are unique words and the values are term_frequency of the word belong to that category * ICF_list[word].

Construction of new dataframe from tf-icf weights -

```
# creating a new dataframe with size len(trainset) and len(unique words) and the columns are unique words. Initially the dataset contains NaN values.
columns = list(unique_words)
rows = np.full((len(trainset),len(columns)),np.nan)
df = pd.DataFrame(rows,columns=columns)
df
```

	plaid	161strong	2008	liu	aw	leeway	22yearold	guilti	burgeon	ambuja	...	sonapt	searchengin	sevenmonth	antiblog	concern	maje	mil
0	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...	NaN	NaN	NaN	NaN	NaN	NaN	
1	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...	NaN	NaN	NaN	NaN	NaN	NaN	
2	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...	NaN	NaN	NaN	NaN	NaN	NaN	
3	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...	NaN	NaN	NaN	NaN	NaN	NaN	
4	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...	NaN	NaN	NaN	NaN	NaN	NaN	
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	
1485	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...	NaN	NaN	NaN	NaN	NaN	NaN	
1486	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...	NaN	NaN	NaN	NaN	NaN	NaN	
1487	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...	NaN	NaN	NaN	NaN	NaN	NaN	
1488	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...	NaN	NaN	NaN	NaN	NaN	NaN	

creating a new dataframe with size len(trainset) and len(unique words) and the columns are unique words. Initially the dataset contains NaN values.

```
[86] # Updating the new dataframe df with tf_icf values correspondily.
     for index,row in df.iterrows():
       category = row['category']
       text = trainset['updated_text'][index]
       for word in unique_words:
         if word in text and word in tf_icf[category]:
           df[word][index] = tf_icf[category][word]
```

The above code will update the values in the new dataframe df by taking from tf_icf dictionary. After this, all the NaN values are replaced with zeros and label encoding is applied on the columns which contains objects (in our dataset, category column contains object).

2. The dataset :

```
[92] # splitting the new dataframe into train and test sets in 70-30 ratio
     from sklearn.model_selection import train_test_split
     x_train, x_test, y_train, y_test = train_test_split(df.drop('category',axis=1),df['category'],test_size=0.3,random_state=65,shuffle=True)
```

The new dataframe df is splitted in 70 - 30 ratio.

3. Training the Naive Bayes classifier with TF-ICF -

```
# applying naive bayes classifier on the train set
from sklearn.naive_bayes import GaussianNB

gnb = GaussianNB()
gnb.fit(x_train,y_train)
```

```
GaussianNB
GaussianNB()
```

The training is done on the training dataset using naive bayes classifier. Here I used GaussianNB variant of naive bayes classifier.

```
# calculating the probability of each category based on the frequency of documents
category_prob = {}   # this dictionary contains the probabilites category wise
total_documents = len(y_train)
print("Total documents in training set are - ",total_documents)

for category in set(y_train):
    category_documents = y_train[y_train == category]
    category_prob[labelencoder_classes[category]] = (len(category_documents)/total_documents)
```

```
Total documents in training set are -  1043
```

The above code is calculating the probability of each category based on the frequency of documents in the training set that belong to that category. The loop will run for no. of unique categories. category_documnets will have the documents related to the category and then probability is calculated using the formula no. of documents in that category / total number of documents and stored in the dictionary category_prob where the key is the category name taken from the labelencoder_classes and the value is the corresponding category probability.

```
# calculating the conditional probability of each feature given category
feature_prob = {}
for category in set(y_train):
    # Subset the training data for the current category
    subset = x_train[y_train == category]

    # Calculate the sum of TF-ICF scores for each feature
    sum_scores = subset.sum(axis=0)
    sum_scores_df = pd.DataFrame([sum_scores],columns=df.columns)

    # Convert the scores to a list and get the feature names from the dataframe df constructed above
    scores_list = [(score, feature) for feature, score in zip(df.columns,sum_scores.tolist())]

    # Sort the list in descending order and keep the top 2500 features
    scores_list = sorted(scores_list, reverse=True)[:2500]

    # Calculate the conditional probability of each feature given the category
    feature_prob[labelencoder_classes[category]] = {}

    for score, feature in scores_list:
      feature_prob[labelencoder_classes[category]][feature] = (score+1)/(sum_scores_df[feature][0]+len(unique_words)+1)
```

The above piece of code will calculate the probability of each feature given each category based on the TF-ICF values of that feature in documents belonging to that category. This is nothing but calculating the conditional probability of each feature given category. As google colab is not showing every feature for every category (i.e., 5*19721) ,to show the probabilities, I have taken top 2500 features.

4. Testing the Naive Bayes classifier with TF-ICF -

```
[102] # predicting the values for the test test
      from sklearn.metrics import accuracy_score

      ypred_gaussian_70 = gnb.predict(x_test)
```

Predicting the values for the testing set where the model is trained on the training set.

Comparison of actual and predicted values -

```
# comparing the actual and predicted values
compare_df = pd.DataFrame()
compare_df['Actual'] = y_test
compare_df['Predicted'] = ypred_gaussian_70
compare_df['compare'] = y_test == ypred_gaussian_70
compare_df
```

	Actual	Predicted	compare
302	0	0	True
142	0	0	True
1105	4	4	True
915	1	1	True
1204	0	0	True
...	...	...	...
788	1	1	True
1187	2	2	True
857	3	3	True

Calculating the recall, precision, f1-score for each category

```
# calculating the precision, recall, f1-score category wise
report = pd.DataFrame(classification_report(y_test, ypred_gaussian_70,output_dict=True))
report
```

	0	1	2	3	4	accuracy	macro avg	weighted avg
precision	0.990291	1.0	1.0	1.000000	0.987179	0.995526	0.995494	0.995554
recall	0.990291	1.0	1.0	0.991597	1.000000	0.995526	0.996378	0.995526
f1-score	0.990291	1.0	1.0	0.995781	0.993548	0.995526	0.995924	0.995528
support	103.000000	74.0	74.0	119.000000	77.000000	0.995526	447.000000	447.000000

Accuracy also calculated.

```
# calculating the accuracy for the predicted values
acc_score_gaussian_70 = accuracy_score(y_test,ypred_gaussian_70)
acc_score_gaussian_70
```

5. Improving the classifier -

```
# splitting the new dataframe df in various splits such as 80-20, 60-40, 50-50 and applying the naive bayes classifier and predicting the accuracies
for i in [50,40,20] :

  print("\nDividing the trainset in",100-i,"-",i,"split\n")
  x_train, x_test, y_train, y_test = train_test_split(df.drop('category',axis=1),df['category'],test_size=i,random_state=65,shuffle=True)

  gnb.fit(x_train,y_train)
  ypred = gnb.predict(x_test)

  acc_score = accuracy_score(y_test, ypred)
  print("Accuracy score is",acc_score)
  print("\nConfusion matrix -\n",confusion_matrix(y_test,ypred))
  print("\nClassification report -\n",pd.DataFrame(classification_report(y_test, ypred,output_dict=True)))
```

I tried for various splits (such as 50-50, 60-40, 80-20) of the dataframe constructed using tf-icf weights and calculated the accuracy, precision, recall, f1-scores for every category.

```
# applying the naive bayes with tf-idf weights using tfidfvectorizer instead of tf-icf weights
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.naive_bayes import MultinomialNB

tfidf = TfidfVectorizer()
tfidf_matrix = tfidf.fit_transform(trainset['Text'])

X_train, X_test, y_train, y_test = train_test_split(tfidf_matrix, df['category'], test_size=0.3, random_state=42)

mnb = MultinomialNB()
mnb.fit(X_train,y_train)
ypred = mnb.predict(X_test)
print("accuracy: ", accuracy_score(y_test, ypred))
```

```
accuracy:  0.9552572706935123
```

Instead of dataframe with tf-icf weights, I have used tf-idf and applied naive bayes (MultinomialNB) on that tf-idf dataset. The reason for not applying GaussianNB on the tf-idf dataset is, the dataset is sparse and not possible to apply and also GaussianNB needs dense dataset. Here, I used TfidVectorizer to construct tf-idf matrix.

```
# applying the naive bayes with tf-idf weights using countvectorizer instead of tf-icf weights
from sklearn.feature_extraction.text import TfidfVectorizer, CountVectorizer
from sklearn.naive_bayes import MultinomialNB

X = trainset["Text"]
y = trainset["Category"]

X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=42, test_size=30, shuffle=True)

tf_vectorizer = CountVectorizer()
X_train_tf = tf_vectorizer.fit_transform(X_train)
X_test_tf = tf_vectorizer.transform(X_test)

mnb = MultinomialNB()
mnb.fit(X_train_tf, y_train)
y_pred = mnb.predict(X_test_tf)
print("accuracy: ", accuracy_score(y_test, y_pred))
```

```
accuracy:  0.9333333333333333
```

Here, instead of TfidVectorizer, I used CountVectorizer to construct tf-idf matrix. The CountVectorizer focuses only on the frequency of the words.

The major difference between the TF-IDF and Count Vectorizers is, TF-IDF is better than Count Vectorizers because it not only focuses on the frequency of words present in the corpus but also provides the importance of the words.

The GaussianNB and MultinomialNB are different variants of Naive bayes classifier. The input features in GaussianNB have gaussian distribution and can be used for both discrete and continuous values whereas the input features in the MultinomialNB follows multinomial distribution and can be used for discrete values only.

3.

Import required libraries

```
import numpy as np
from google.colab import drive
import pandas as pd
import math
import matplotlib.pyplot as plt
```

Prepared a list of columns that are used as column names in the data frame. Column ‘val’ represents the relevance score

```
#prepare a list of column names which can be used while dataframe creation
col = ['val', 'qid']
for i in range(1, 138):
  col.append(str(i))
```

```
df = pd.read_csv("dataset.txt", sep = " ", names = col)
df
```

Make a data frame of all rows having ‘qid’ as 4. The new data frame has 103 rows and 139 columns now.

```
#take out the rows having qid as 4 from the dataset

df1 = df.loc[df['qid'] == "qid:4"]
df1
```

	val	qid	1	2	3	4	5	6	7	8	...
0	0	qid:4	1:3	2:0	3:2	4:0	5:3	6:1	7:0	8:0.666667	...
1	0	qid:4	1:3	2:0	3:3	4:0	5:3	6:1	7:0	8:1	...
2	0	qid:4	1:3	2:0	3:2	4:0	5:3	6:1	7:0	8:0.666667	...
3	0	qid:4	1:3	2:0	3:3	4:0	5:3	6:1	7:0	8:1	...
4	1	qid:4	1:3	2:0	3:3	4:0	5:3	6:1	7:0	8:1	...
...	...	...	...	...	...	...	...	...	...	...	...
98	0	qid:4	1:3	2:0	3:2	4:0	5:3	6:1	7:0	8:0.666667	...
99	1	qid:4	1:3	2:0	3:3	4:2	5:3	6:1	7:0	8:1	...
100	2	qid:4	1:2	2:0	3:2	4:0	5:2	6:0.666667	7:0	8:0.666667	...
101	1	qid:4	1:2	2:0	3:2	4:0	5:2	6:0.666667	7:0	8:0.666667	...
102	0	qid:4	1:3	2:0	3:2	4:0	5:3	6:1	7:0	8:0.666667	... 1

103 rows × 139 columns

Replace NaN values with zero

```
#replacing NaN values with 137:0 for column "137"
df1['137'] = df1['137'].fillna('137:0')
df1
```

Since column ‘qid’ is not needed further we can drop it. Every column contains data of the format ‘column-name: value’, here only the value will be used so we have stripped out all data till ‘:’ and kept the number. Convert the number to float for mathematical ease.

```
df1.drop(['qid'], axis=1, inplace = True)       #since column "qid" is not needed now so we can drop it
col.remove('qid')

for i in col:
  df1[i] = df1[i].str[(len(i) +1 ):]        #anything before colon (:) is not needed so we remove it

df1 = df1.astype(float)   #converting everything to float for mathematical convenience
```

Made a copy of the data frame for DCG calculation.
df2 ===> used for DCG calculation
df1 ===> used for IDCG calculation

```
df2 = df1.copy(deep=True)       #making a copy of dataframe so that it can be used for DCG calculation
df2
```

Sort df1 on the basis of the relevance score in descending order. Here relevance score is ‘val’ column

```
df1 = df1.sort_values(by='val', ascending=False)         #sort dataframe by relevence score in descending order
```

Reset the indexes so that we can easily track all the values in the data frame

```
df1.reset_index(drop=True, inplace=True)        #reset index values to keep track of dataframe easily
```

===> we count the number of times each relevance score occurred. Then compute the product of the factorial of each count.

First Objective ===> number of files

```
[ ] count  =df1['val'].value_counts()   #count number of times each relevence score occured
    count
```

```
0.0    59
1.0    26
2.0    17
3.0     1
Name: val, dtype: int64
```

```
[ ] res=1
    for v in count:
      res = res * math.factorial(v)
    res
```

```
198934973759383705998260476149053298969368401705665705882051803127048579926951934824126865654310502400000000000000000000000
```

Compute the IDCG value on df1 and the DCG value on df2. While doing this, for every iteration store the respective IDCG and DCG values in the array.
IDCGArr ===> stores the list of IDCG values at every iteration
DCGArr ===> stores the list of DCG values at every iteration

IDCG value obtained for the entire dataset = `20.989750804831445`
DCG value obtained for the entire dataset = `12.550247459532576`

NDCG = DCG / IDCG
NDCG for the entire dataset = `0.5979226516897831`

```
#first Compute IDCG on sorted relevence score

IDCG =0
IDCGArr=[]        #keeps track of IDCG values for every row
for i in range(0, 103):
  if i==0:
    IDCG = IDCG + df1['val'][i]       #for first value take relevence score directly
  else:
    logval = math.log2(i+1)
    IDCG = IDCG + (df1['val'][i]/logval)

  IDCGArr.append(IDCG)

print("IDCG  ===>" + str(IDCG))
```

```
IDCG  ===>20.989750804831445
```

```
#first Compute DCG on unsorted relevence score

DCG =0
DCGArr=[]       #keeps track of DCG values for every row
for i in range(0, 103):
  if i==0:
    DCG = DCG + df2['val'][i]   #for first value take relevence score directly
  else:
    logval = math.log2(i+1)
    DCG = DCG + (df2['val'][i]/logval)     #for next relevence scores divide by log

  DCGArr.append(DCG)

print("DCG  ===>" + str(DCG))
```

```
DCG  ===>12.550247459532576
```

```
#calculate nDCG for the entire dataset
NDCG = DCG/IDCG
NDCG
```

```
0.5979226516897831
```

Similarly, compute NDCG at position 50
We get it as `0.3521042740324887`

```
#calculate nDCG at position 50 .
NDCG_pos50 = DCGArr[49]/IDCGArr[49]
NDCG_pos50
```

```
0.3521042740324887
```

Sort the df2 data frame on column name ‘75’ in descending order.
Compute the precision and recall at every iteration and store it in a list

Precision = (relevant relevance numbers at iteration i)/ (ith iteration)

```
df2 = df2.sort_values(by='75', ascending=False)   #sort column named 75 in reverse order
```

```
precision=[]      #Precision Array
recall =[]        #Recall Array
relevent=0        #keeps a count of total number of relevence scores

#Calculate Precision
for i in range(0, 103):
  if df2['val'][i]!=0:
    relevent = relevent +1
  precision.append(relevent/(i+1))

print(precision)
```

```
[0.0, 0.0, 0.0, 0.0, 0.2, 0.16666666666666666, 0.2857142857142857, 0.375, 0.3333333333333333, 0.3, 0.
```

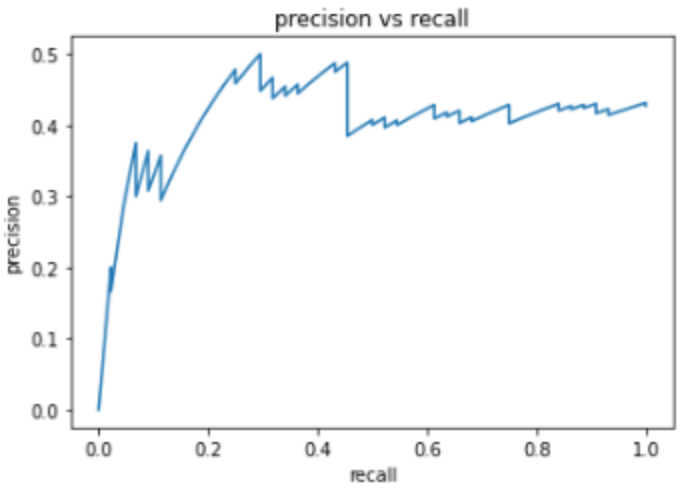
Recall = (relevant relevance numbers at iteration i)/ (total relevant relevance numbers)

```
releventpoint=0   #keeps track of relevent relevence score at a particular point

#Calculate recall
for i in range(0, 103):
  if df2['val'][i]!=0:
    releventpoint = releventpoint +1
  recall.append(releventpoint/relevent)

print(recall)
```

```
[0.0, 0.0, 0.0, 0.0, 0.022727272727272728, 0.022727272727272728, 0.045454545454545456, 0.
```

Plot precision vs recall

```
#plot precision vs recall

plt.plot(recall, precision)
plt.xlabel("recall")
plt.ylabel("precision")
plt.title("precision vs recall")
plt.show()
```


precision vs recall
precision
recall
0.0
0.1
0.2
0.3
0.4
0.5
0.0
0.2
0.4
0.6
0.8
1.0