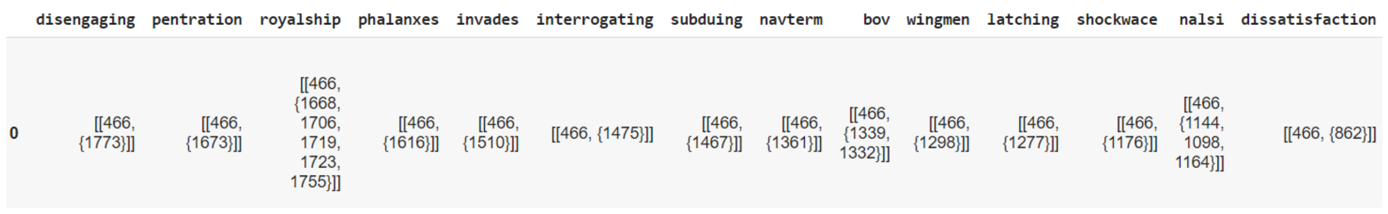
CSE508 IR Assignment 2
Group 41

Q1. Positional Index

Preprocessing steps performed:

1.Convert the text to lower case: Converting files text into lowercase
2.Perform Word Tokenization: Split files into tokens
3.Remove Stopwords from tokens: Stopwords are the common words like a,at,the, an which are not useful in analysis.So stop words were removed using nltk library.
4.punctuation marks were then removed from tokens
5.At last blank spaces were removed.

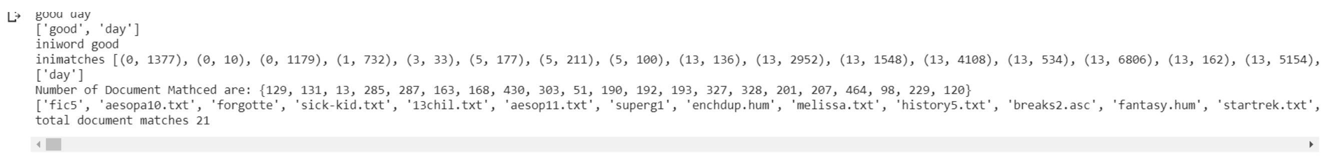
Positional index: In the positional index for each term in the vocab, we store the postings in the form like (documentID: position1, position2,) where each position is the index of the token in the document.
Here is the positional indexes we are getting:



	disengaging	pentration	royalship	phalanxes	invades	interrogating	subduing	navterm	bov	wingmen	latching	shockwace	nalsi	dissatisfaction
0	[[466, {1773}]]	[[466, {1673}]]	[[466, {1668, 1706, 1719, 1723, 1755}]]	[[466, {1616}]]	[[466, {1510}]]	[[466, {1475}]]	[[466, {1467}]]	[[466, {1361}]]	[[466, {1339, 1332}]]	[[466, {1298}]]	[[466, {1277}]]	[[466, {1176}]]	[[466, {1144, 1098, 1164}]]	[[466, {862}]]

Query:first input query is taken from user .Then preprocessing is performed on the query.Then postings are fetched according to the query and matched documents name and number of documents matched are returned.

Output for query good day:



```
good day
['good', 'day']
iniword good
inimatches [(0, 1377), (0, 10), (0, 1179), (1, 732), (3, 33), (5, 177), (5, 211), (5, 100), (13, 136), (13, 2952), (13, 1548), (13, 4108), (13, 534), (13, 6806), (13, 162), (13, 5154),
['day']
Number of Document Mathced are: {129, 131, 13, 285, 287, 163, 168, 430, 303, 51, 190, 192, 193, 327, 328, 201, 207, 464, 98, 229, 120}
['fic5', 'aesopa10.txt', 'forgotte', 'sick-kid.txt', '13chil.txt', 'aesop11.txt', 'superg1', 'enchdup.hum', 'melissa.txt', 'history5.txt', 'breaks2.asc', 'fantasy.hum', 'startrek.txt',
total document matches 21
```

Q2 : Scoring and Term-Weighting

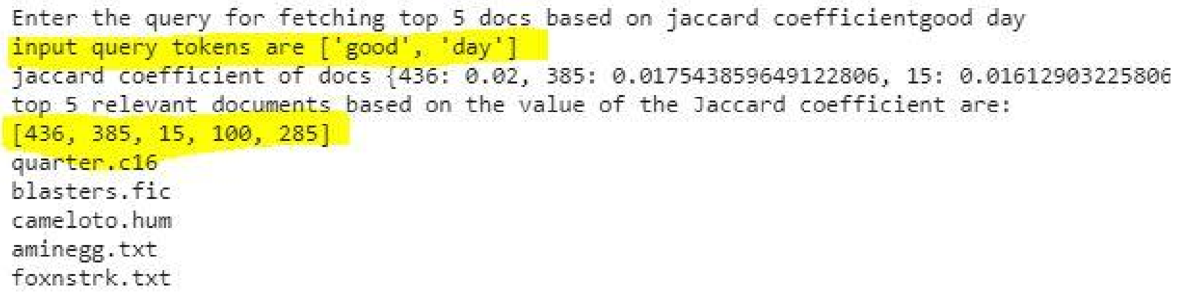
As we have same dataset use in above Question 1
We use the Positional index that we defined earlier in this question.

1. **Jaccard Coefficient:**
    a. First we insert a query. (Shown in below image.)
    b. Perform Preprocessing on query which steps are defined above.
    c. Create token for preprocess query query_tokens().
    d. For each document token list i.e. "`files_tokens`" we find the union and intersection with the query token list "`query_list`". And save the jaccard coefficient value for each document with the help of the jaccard formula i.e.

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|}$$

    e. we retrieve top 5 relevant documents based on the value of the jaccard coefficient.

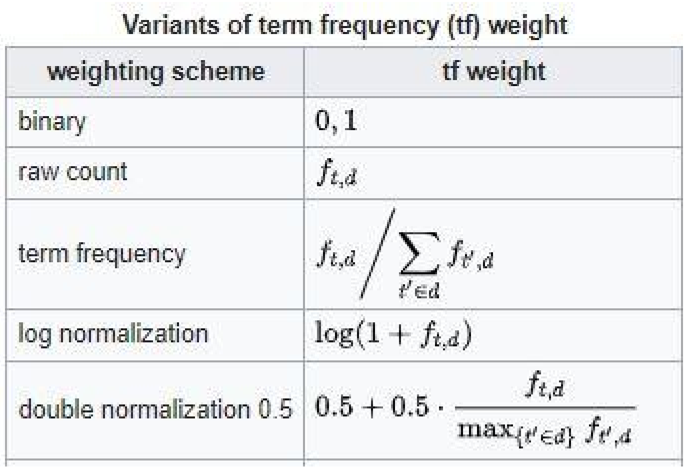
Here we have have query "**good day**" then we find jaccard coefficient the top 5 documents are shown in the highlighted part on below image.



```
Enter the query for fetching top 5 docs based on jaccard coefficientgood day
input query tokens are ['good', 'day']
jaccard coefficient of docs {436: 0.02, 385: 0.017543859649122806, 15: 0.01612903225806
top 5 relevant documents based on the value of the Jaccard coefficient are:
[436, 385, 15, 100, 285]
quarter.c16
blasters.fic
cameloto.hum
aminegg.txt
foxnstrk.txt
```

2. **TF-IDF Matrix:**
    a. First we calculate the Term frequency using 5 different weighting variant:
        i. **Raw Count Variant:** In Raw Count Variant we count the frequency of each token for each document and stored in the list.
        ii. **Term Frequency Variant:** Term Frequency Variant is calculated by counting no. of tokens in each document and then dividing each term frequency by all numbers of words in the document.
        iii. **Log Normalization Variant**: In Log Normalization Variant create dictionary for storing log information term frequency which is appended into list of list for storing Log normalization term frequency for each document .
        iv. **Double Normalization Variant:** In Double Normalization Variant creates a dictionary which stores double normalization term frequency for each document and then appends into list of lists for storing double normalization term frequency for each document.
        v. **Binary Variant**: In Binary Variant create a dictionary which stores binary term frequency for each document and then append into list of list for store Binary Variant term frequency for each document.

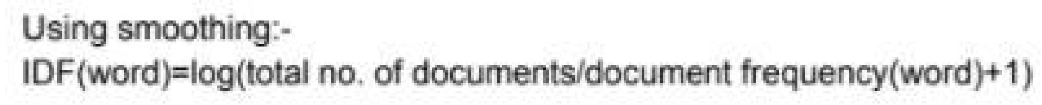
        Formula of all 5 variants of Term Frequency is shown below.



Variants of term frequency (tf) weight

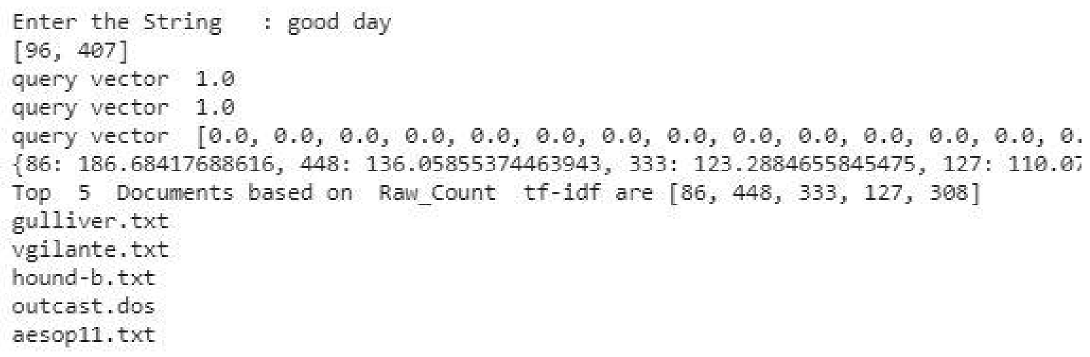
weighting scheme	tf weight
binary	$0, 1$
raw count	$f_{t,d}$
term frequency	$f_{t,d} \Big/ \sum_{t' \in d} f_{t',d}$
log normalization	$\log(1 + f_{t,d})$
double normalization 0.5	$0.5 + 0.5 \cdot \frac{f_{t,d}}{\max_{\{t' \in d\}} f_{t',d}}$

b. Create the Document Frequency dictionary for each vocabulary that stores the count the no of documents in which that term is present.
c. Create Inverse Document Frequency Dictionary using above document frequency dictionary i.e for each key term have value equal to logarithm of ratio of Total no of documents by document frequency of that term with smoothing technique.
Formula of IDF is:



Using smoothing:-
IDF(word)=log(total no. of documents/document frequency(word)+1)

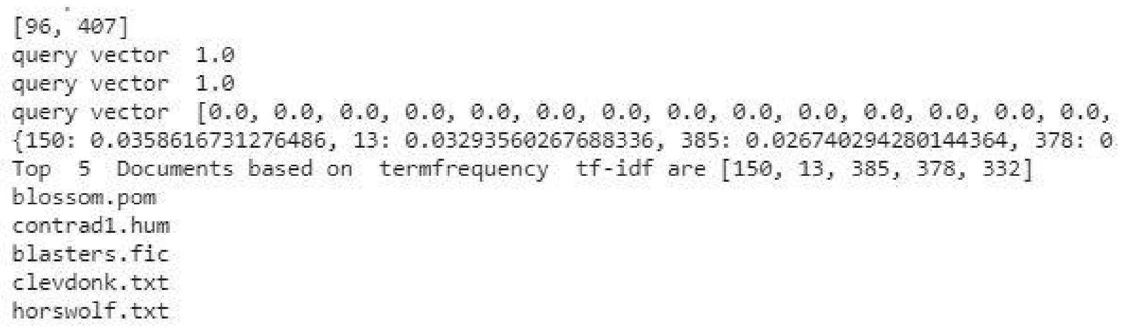
d. Find the Tf-Idf for all 5 variants
   i. Raw Count Variant:
      In this we use input query is -> “good day”



```
Enter the String   : good day
[96, 407]
query vector  1.0
query vector  1.0
query vector  [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.
{86: 186.68417688616, 448: 136.05855374463943, 333: 123.2884655845475, 127: 110.07
Top  5  Documents based on  Raw_Count  tf-idf are [86, 448, 333, 127, 308]
gulliver.txt
vgilante.txt
hound-b.txt
outcast.dos
aesop11.txt
```

   ii. Term Frequency Variant:

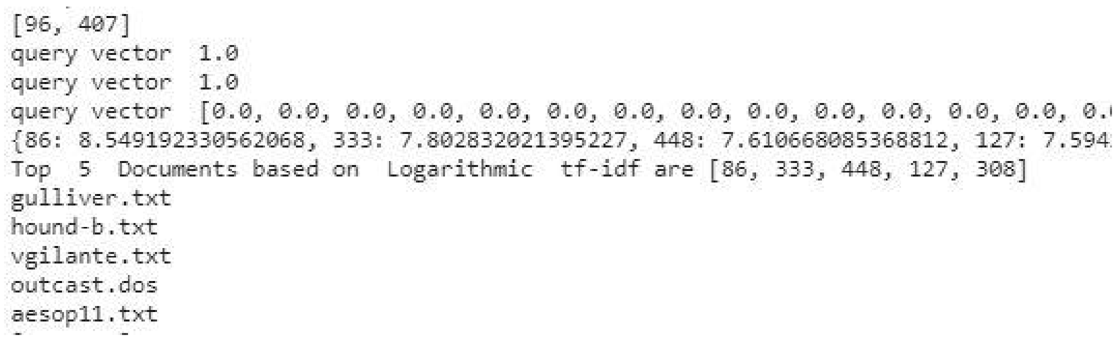
In this we use input query is -> “good day”



```
[96, 407]
query vector  1.0
query vector  1.0
query vector  [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
{150: 0.0358616731276486, 13: 0.03293560267688336, 385: 0.026740294280144364, 378: 0
Top  5  Documents based on  termfrequency  tf-idf are [150, 13, 385, 378, 332]
blossom.pom
contrad1.hum
blasters.fic
clevdonk.txt
horswolf.txt
```

iii. Log Normalization Variant:

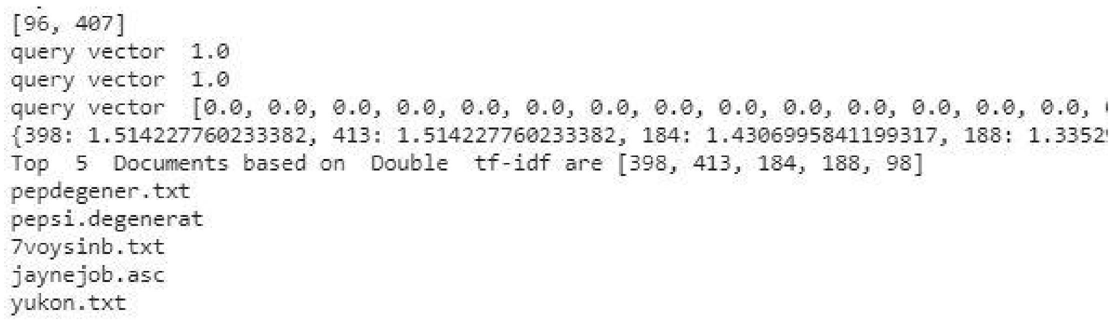
In this we use input query is -> “good day”



```
[96, 407]
query vector  1.0
query vector  1.0
query vector  [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.
{86: 8.549192330562068, 333: 7.802832021395227, 448: 7.610668085368812, 127: 7.594
Top  5  Documents based on  Logarithmic  tf-idf are [86, 333, 448, 127, 308]
gulliver.txt
hound-b.txt
vgilante.txt
outcast.dos
aesop11.txt
```

iv. Double Normalization Variant:

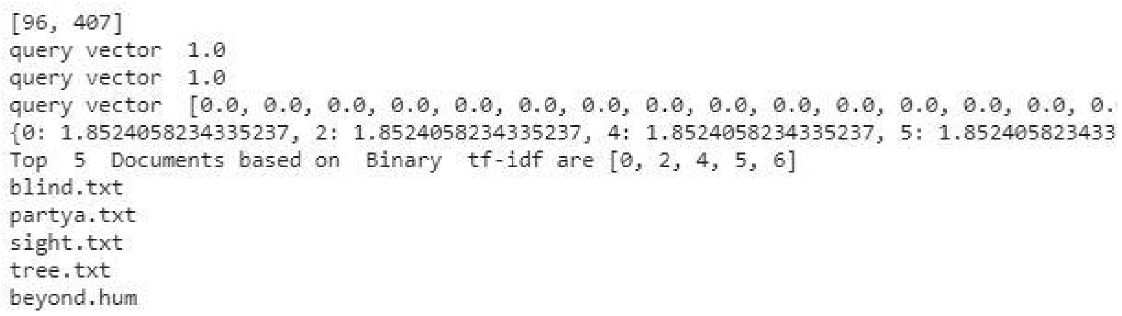
In this we use input query is -> “good day”



```
[96, 407]
query vector  1.0
query vector  1.0
query vector  [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
{398: 1.514227760233382, 413: 1.514227760233382, 184: 1.4306995841199317, 188: 1.3352
Top  5  Documents based on  Double  tf-idf are [398, 413, 184, 188, 98]
pepdegener.txt
pepsi.degenerat
7voysinb.txt
jaynejob.asc
yukon.txt
```

v. Binary Variant:

In this we use input query is -> “good day”



```
[96, 407]
query vector  1.0
query vector  1.0
query vector  [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.
{0: 1.8524058234335237, 2: 1.8524058234335237, 4: 1.8524058234335237, 5: 1.852405823433
Top  5  Documents based on  Binary  tf-idf are [0, 2, 4, 5, 6]
blind.txt
partya.txt
sight.txt
tree.txt
beyond.hum
```

e. Print the top 5 documents id and name for each Tf-Idf variant.

3. **Cosine Similarity:**
   The cosine similarity is used to measure the similarity between the query vector and the document vector of length of the vocabulary.

   a. **Query Vector:**
      We have taken a query vector on the basis of two approaches:
      - one is a query vector of [0, 1] and
      - another is a weighted one in which tfidf is calculated for the query vector using all the 5 TF weighting schemes.

   b. **TF-IDF Matrix:**
      Matrix is picked from the tf-idf part for all the 5 weighting schemes.

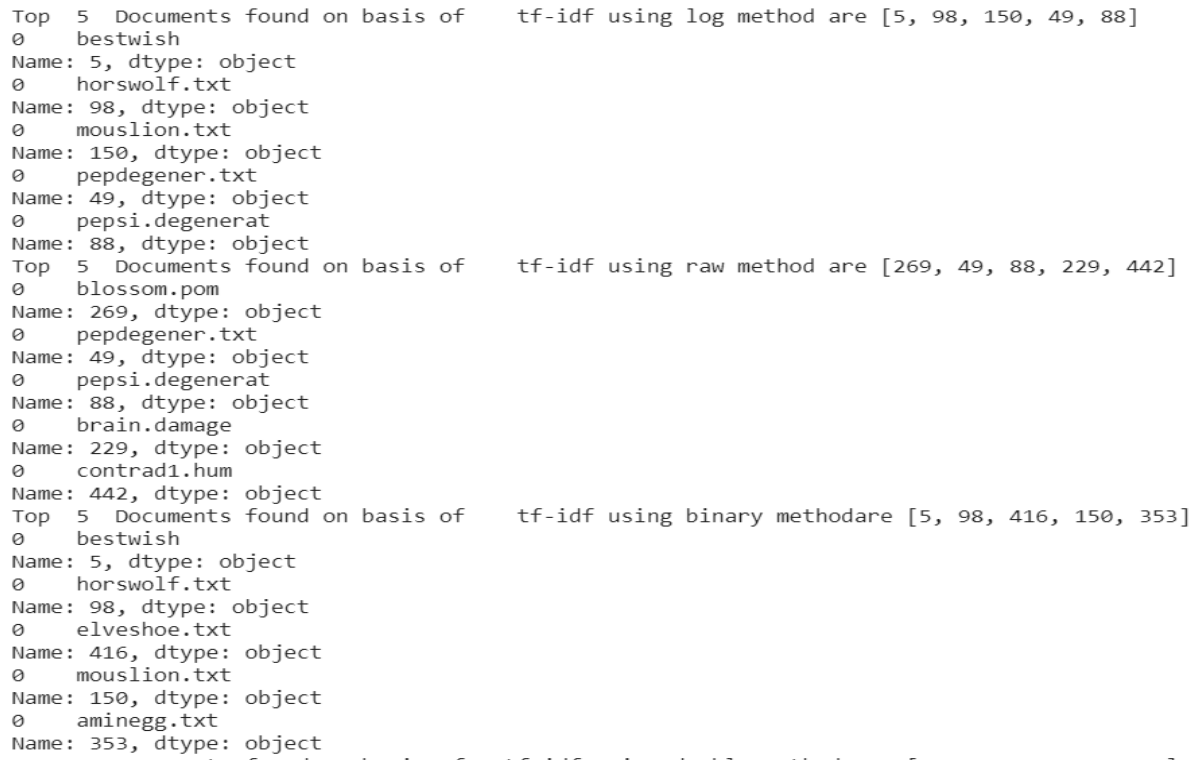
   c. **Calculated Cosine Similarity Score:**
      Cosine Similarity score is calculated using both types of query_vector and tf-idf matrix.
      Normalisation of query and document vector is done by dividing each of its components by its length. It is simply the dot product of query vector and document vector.

$$\cos(\vec{q},\vec{d}) = \vec{q} \bullet \vec{d} = \sum_{i=1}^{|V|} q_i d_i$$

for q, d length-normalized.

cosine_score+=doc*query

d. Top 5 documents are fetched on the basis of cosine score using all 5 weighting schemes.



```
Top  5  Documents found on basis of    tf-idf using log method are [5, 98, 150, 49, 88]
0    bestwish
Name: 5, dtype: object
0    horswolf.txt
Name: 98, dtype: object
0    mouslion.txt
Name: 150, dtype: object
0    pepdegener.txt
Name: 49, dtype: object
0    pepsi.degenerat
Name: 88, dtype: object
Top  5  Documents found on basis of    tf-idf using raw method are [269, 49, 88, 229, 442]
0    blossom.pom
Name: 269, dtype: object
0    pepdegener.txt
Name: 49, dtype: object
0    pepsi.degenerat
Name: 88, dtype: object
0    brain.damage
Name: 229, dtype: object
0    contrad1.hum
Name: 442, dtype: object
Top  5  Documents found on basis of    tf-idf using binary methodare [5, 98, 416, 150, 353]
0    bestwish
Name: 5, dtype: object
0    horswolf.txt
Name: 98, dtype: object
0    elveshoe.txt
Name: 416, dtype: object
0    mouslion.txt
Name: 150, dtype: object
0    aminegg.txt
Name: 353, dtype: object
```

```
Name: [illegible], dtype: object
Top  5  Documents found on basis of   tf-idf using double method are [5, 98, 150, 82, 416]
0    bestwish
Name: 5, dtype: object
0    horswolf.txt
Name: 98, dtype: object
0    mouslion.txt
Name: 150, dtype: object
0    blasters.fic
Name: 82, dtype: object
0    elveshoe.txt
Name: 416, dtype: object
Top  5  Documents found on basis of    tf-idf using term are [269, 49, 88, 229, 442]
0    blossom.pom
Name: 269, dtype: object
0    pepdegener.txt
Name: 49, dtype: object
0    pepsi.degenerat
Name: 88, dtype: object
0    brain.damage
Name: 229, dtype: object
0    contrad1.hum
Name: 442, dtype: object
```

Pro and Cons of each Scoring Schemes:

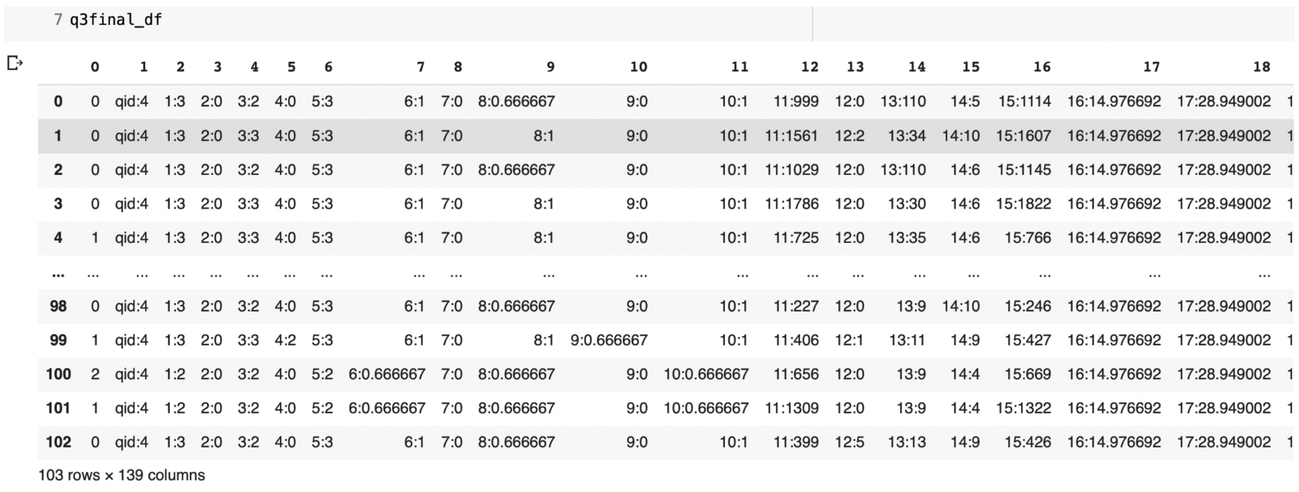
	Pros	**Cons**
Jaccard coefficient	Better Result where duplication or repetition of words does not matter.	Term frequency is not considered so it doesn’t consider rare terms in a collection.
TF-IDF Matrix	Easy to compute the similarity b/w 2 different documents. Rare terms are more informative than frequent terms.	It is based on the bag-of-words (BoW) model, therefore it does not capture position in text, semantics, co-occurrences in different-different documents, etc.
Cosine Similarity	Smaller angles between documents have higher similarity that helps in clustering and classification between the documents. Basically used to Classify the documents.	The cosine similarity looks at “directional similarity” rather than magnitudinal differences. It is only concerned with the orientation of two points and not with their exact placement. This means that **cosine distance** is much less affected by **magnitude** only.

Q3: Ranked-Information Retrieval and Evaluation

Discounted cumulative gain:

DCG measures the gain in a document based on its position in the result list and gain is accumulated from top to the bottom of the result list where for the lower ranks, the result is discounted.
So DGC measures the ranking quality and the competentivity of the search algorithms.

1)
The given data we have read into a pandas dataframe:



```
7 q3final_df
```

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	
0	0	qid:4	1:3	2:0	3:2	4:0	5:3	6:1	7:0	8:0.666667	9:0	10:1	11:999	12:0	13:110	14:5	15:1114	16:14.976692	17:28.949002	1
1	0	qid:4	1:3	2:0	3:3	4:0	5:3	6:1	7:0	8:1	9:0	10:1	11:1561	12:2	13:34	14:10	15:1607	16:14.976692	17:28.949002	1
2	0	qid:4	1:3	2:0	3:2	4:0	5:3	6:1	7:0	8:0.666667	9:0	10:1	11:1029	12:0	13:110	14:6	15:1145	16:14.976692	17:28.949002	1
3	0	qid:4	1:3	2:0	3:3	4:0	5:3	6:1	7:0	8:1	9:0	10:1	11:1786	12:0	13:30	14:6	15:1822	16:14.976692	17:28.949002	1
4	1	qid:4	1:3	2:0	3:3	4:0	5:3	6:1	7:0	8:1	9:0	10:1	11:725	12:0	13:35	14:6	15:766	16:14.976692	17:28.949002	1
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	
98	0	qid:4	1:3	2:0	3:2	4:0	5:3	6:1	7:0	8:0.666667	9:0	10:1	11:227	12:0	13:9	14:10	15:246	16:14.976692	17:28.949002	1
99	1	qid:4	1:3	2:0	3:3	4:2	5:3	6:1	7:0	8:1	9:0.666667	10:1	11:406	12:1	13:11	14:9	15:427	16:14.976692	17:28.949002	1
100	2	qid:4	1:2	2:0	3:2	4:0	5:2	6:0.666667	7:0	8:0.666667	9:0	10:0.666667	11:656	12:0	13:9	14:4	15:669	16:14.976692	17:28.949002	1
101	1	qid:4	1:2	2:0	3:2	4:0	5:2	6:0.666667	7:0	8:0.666667	9:0	10:0.666667	11:1309	12:0	13:9	14:4	15:1322	16:14.976692	17:28.949002	1
102	0	qid:4	1:3	2:0	3:2	4:0	5:3	6:1	7:0	8:0.666667	9:0	10:1	11:399	12:5	13:13	14:9	15:426	16:14.976692	17:28.949002	1

103 rows × 139 columns

As we need to find only the documents with qid:4 ,

q3final_df = rire_df[rire_df[1] == "qid:4"]

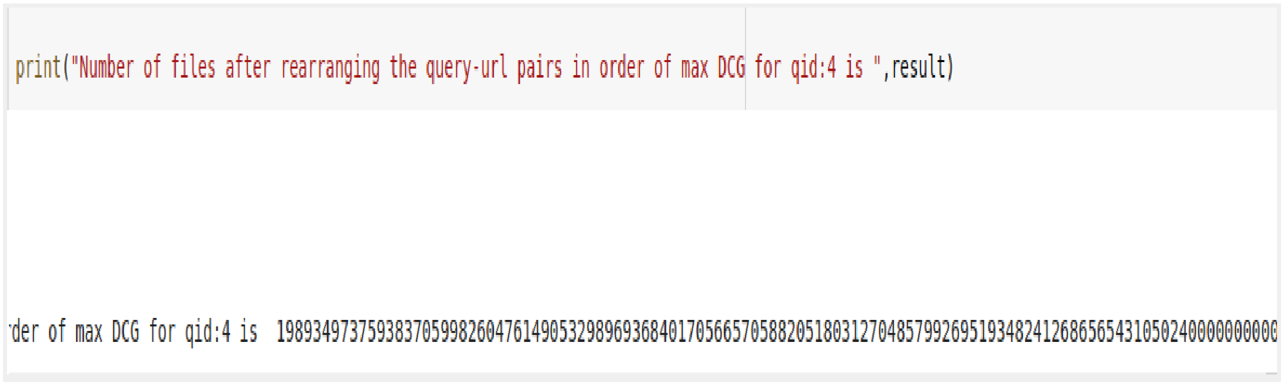
Also, we have considered the relevance judgement labels as relevance scores.

2)
 We have created a csv file attached along which has rearranged query-url pairs in order of max DCG with name **q3DCG.csv**.
The unique_vals stores the unique values in column 0 of the dataframe and we have looped over the unique values for the column 0 and appended the no of instances for each unique value 0,1 and 2 in a list lenOflist and the result is calculated by

result = result * math.factorial(lenOflist)

With this we have got the output as below:

Number of files after rearranging the query-url pairs in order of max DCG for qid:4 is 198934973759383705998260476149053298969368401705665705882051803127048579926951934824126865654310502400000000000000000000000



```
print("Number of files after rearranging the query-url pairs in order of max DCG for qid:4 is ",result)
```

```
rder of max DCG for qid:4 is  19893497375938370599826047614905329896936840170566570588205180312704857992695193482412686565431050240000000000
```

3)

We have created a function named computenDCG which takes the dataframe and the value of n as parameters, nDGC is the normalized DCG.

```
#function to compute nDCG
def computenDCG(rire_df, n):
  result = 0;
  for i in range(1, n+1):
     result = result + (pow(2, rire_df[0][i-1]) - 1)/(np.log2(i+1))
  return result
```

In normalized DCG, the cumulative gain is normalized over the queries with no value as the search algorithms performance across queries cannot be compared using DCG only.

- **nDGC for 50**

  It is computed by calling the function as follows:

  computenDCG(q3final_df,50)/computenDCG(dfr,50)

- **nDGC for the whole document**

  It is computed by calling the function as follows:

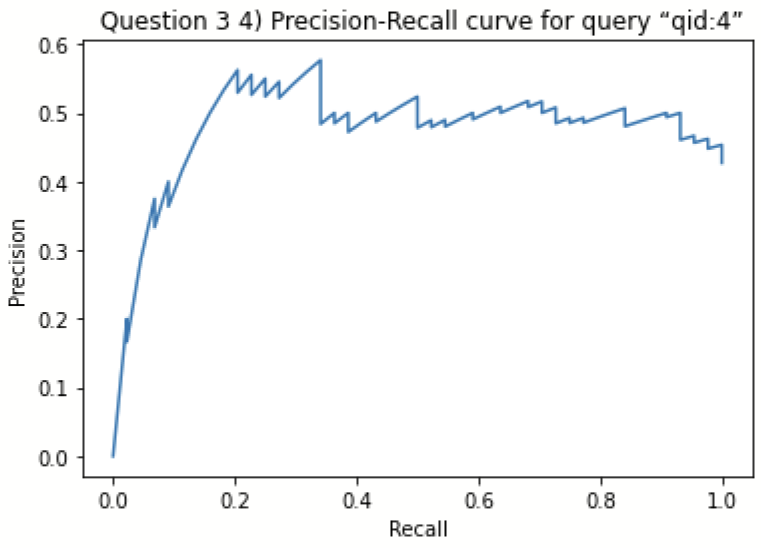
  computenDCG(q3final_df,len(q3final_df))/computenDCG(dfr,len(q3final_df))

Output:

```
Ques 3 part 3 i) nDCG at 50: 0.35612494416255847
Ques 3 part 3 ii) nDCG for whole dataset: 0.5784691984582591
```

4)

Here we need to plot a Precision-Recall curve for query “qid:4”.
Assumption: Assume any non zero relevance judgment value to be relevant.
The plot grows straight exponentially between 0 and 0.2 Recall values and reaches value above 0.5 recall. After that the precision remains between 0.4 and 0.55 as the recall values span over 0.2 and 1.0


Question 3 4) Precision-Recall curve for query “qid:4”
Precision
Recall