IRWS PROJECT PART A

PRIYANK SONKIYA 17DCS009

RACHIT JAIN 17UCS118

March 2021

Contents

1 Introduction

This is the project report for the IRWS Project PART A as a component of the IRWS course even semester 2020- 2021 at The LNM Institute Of Information Technology, Jaipur. The information retrieval process starts with a user based query. Several documents may match that query, and then a Ranked Retrieval is returned in descending order of relevance.
So we have chosen five queries as instructed, performed the information retrieval task, and finally calculated our IR system's performance.

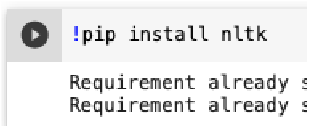
2 Dataset

The link of the data set is BBC NEWS(click-me).
This a BBC news dataset, although it contained pre-processed data, also we chose the raw-data as per the tasks of our project. The data consists of 2225 documents from the BBC news website corresponding to stories in five topical areas from 2004-2005 under five categories -
1.Business
2.Politics
3.Sports
4.Entertainment
5.Tech
We have further scaled-down the data to 50 documents per category as the Project question instructed to a total of 250 documents.

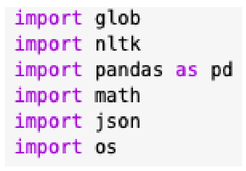
3 Objectives

3.1 Creating a Term Document matrix and Inverted Index

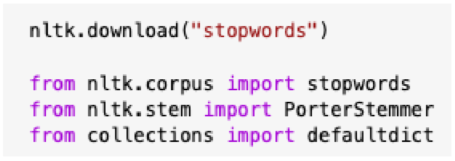
We start by importing necessary libraries and packages -



```
!pip install nltk

Requirement already s
Requirement already s
```

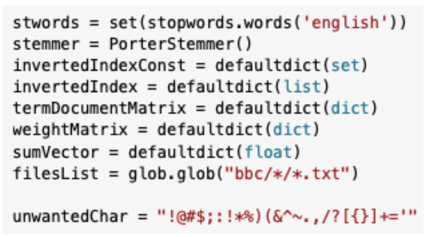


```
import glob
import nltk
import pandas as pd
import math
import json
import os
```

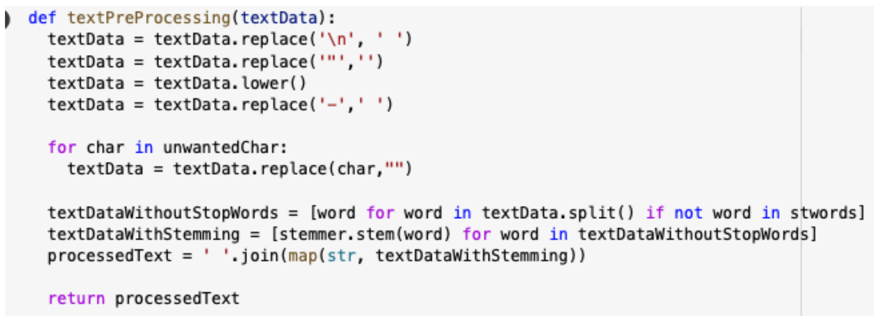


```
nltk.download("stopwords")

from nltk.corpus import stopwords
from nltk.stem import PorterStemmer
from collections import defaultdict
```

The bbc.zip package containing the data,queries and relevance judgement is unzipped and the 250 files are read and stored in a list for pre-processing.



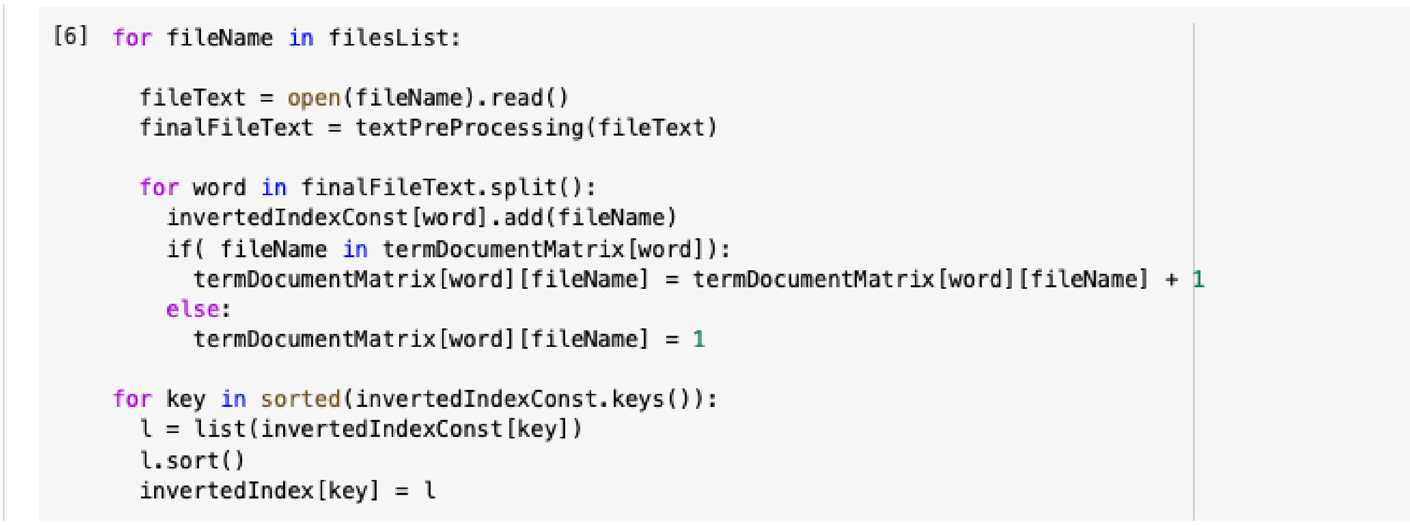
```
stwords = set(stopwords.words('english'))
stemmer = PorterStemmer()
invertedIndexConst = defaultdict(set)
invertedIndex = defaultdict(list)
termDocumentMatrix = defaultdict(dict)
weightMatrix = defaultdict(dict)
sumVector = defaultdict(float)
filesList = glob.glob("bbc/*/*.txt")

unwantedChar = "!@#$;:!*%)(&^~.,/?[{}]+='"
```



```
def textPreProcessing(textData):
  textData = textData.replace('\n', ' ')
  textData = textData.replace('"','')
  textData = textData.lower()
  textData = textData.replace('-',' ')

  for char in unwantedChar:
    textData = textData.replace(char,"")

  textDataWithoutStopWords = [word for word in textData.split() if not word in stwords]
  textDataWithStemming = [stemmer.stem(word) for word in textDataWithoutStopWords]
  processedText = ' '.join(map(str, textDataWithStemming))

  return processedText
```

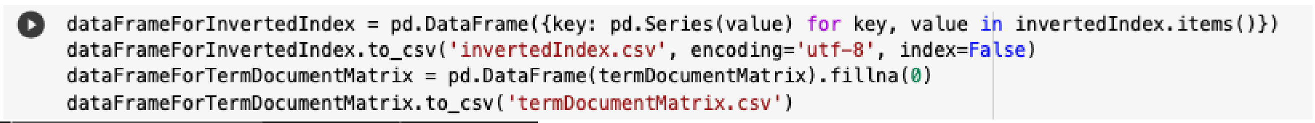
In the above snippets, we can see all the variables are declared, the unwanted characters are declared, further preprocessing is done that includes:
1. Removing unwanted characters from the data like !,@,+,= and also the next line character(" n ") which is converted to simple space and other necessary replacements are made.
2. Tokenization: The entire corpus is scanned from the GLOB package, and we have a list of all the words, which are basically tokens.
3. Stopwords downloaded from the NLTK library are removed from the corpus so that words like - "I", "my", "a", etc. are removed. These will not be further processed since they do not contain any semantic or pragmatic meaning and have a very high frequency hence reducing the data size. Hence the tokens are updated.

4. Stemming: Using the PorterStemmer package, which we studied in class, also we performed the stemming operation. This does operations like -
making -becomes- make
makes -becomes- make
make -remains- make
Hence our corpus is tokenized and ready for turning to a dictionary or a term document Matrix .

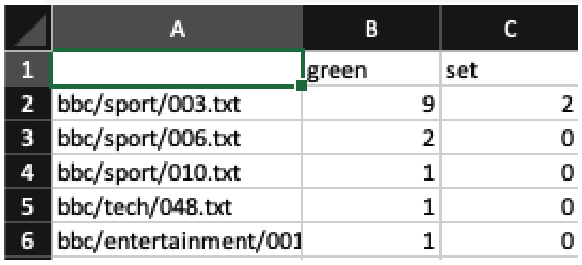


```
[6] for fileName in filesList:

      fileText = open(fileName).read()
      finalFileText = textPreProcessing(fileText)

      for word in finalFileText.split():
        invertedIndexConst[word].add(fileName)
        if( fileName in termDocumentMatrix[word]):
          termDocumentMatrix[word][fileName] = termDocumentMatrix[word][fileName] + 1
        else:
          termDocumentMatrix[word][fileName] = 1

    for key in sorted(invertedIndexConst.keys()):
      l = list(invertedIndexConst[key])
      l.sort()
      invertedIndex[key] = l
```

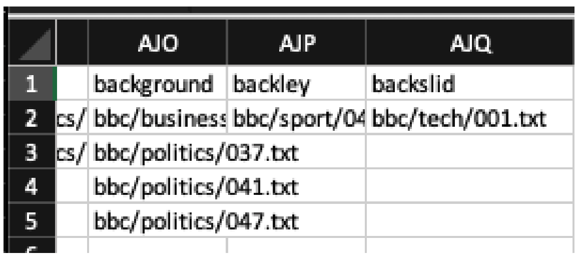
The above is a snippet of creating the term-document matrix and inverted index simultaneously. For all individual 250 files, the code checks with the processed data, create a column for a particular word and update the count in each cell, the same approach for the inverted index as well where only the file where the word is present is mentioned.



```
dataFrameForInvertedIndex = pd.DataFrame({key: pd.Series(value) for key, value in invertedIndex.items()})
dataFrameForInvertedIndex.to_csv('invertedIndex.csv', encoding='utf-8', index=False)
dataFrameForTermDocumentMatrix = pd.DataFrame(termDocumentMatrix).fillna(0)
dataFrameForTermDocumentMatrix.to_csv('termDocumentMatrix.csv')
```



	A	B	C
1		green	set
2	bbc/sport/003.txt	9	2
3	bbc/sport/006.txt	2	0
4	bbc/sport/010.txt	1	0
5	bbc/tech/048.txt	1	0
6	bbc/entertainment/001	1	0

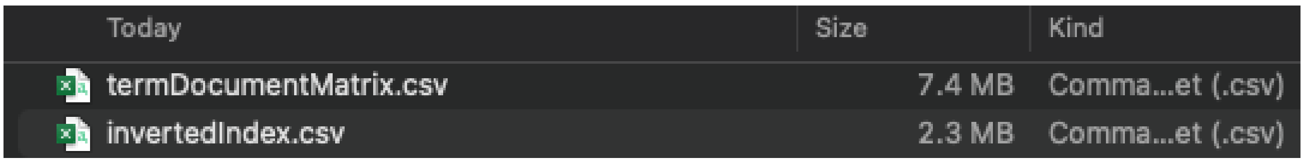


		AJO	AJP	AJQ
1		background	backley	backslid
2	cs/	bbc/business	bbc/sport/04	bbc/tech/001.txt
3	cs/	bbc/politics/037.txt		
4		bbc/politics/041.txt		
5		bbc/politics/047.txt		

The above code finally makes two files InvertedIndex.csv and TermDocumentMatrix.csv. Both files will be automatically generated and can be downloaded from the google colab notebook when it is executed. The snippets of both files are also attached. The term-document matrix contains all the words as columns and the rows as all the documents, and the count of presence in those documents of a particular word(column). The inverted index contains all the document where a particular word is present.

3.2 Comparing the storage requirements

The term-document matrix is a 250rows x 7360 columns matrix corresponding to 250 documents and 7360 words in the dictionary, and the inverted index is a matrix of a similar shape in the code, but their memory difference as on the local system is -

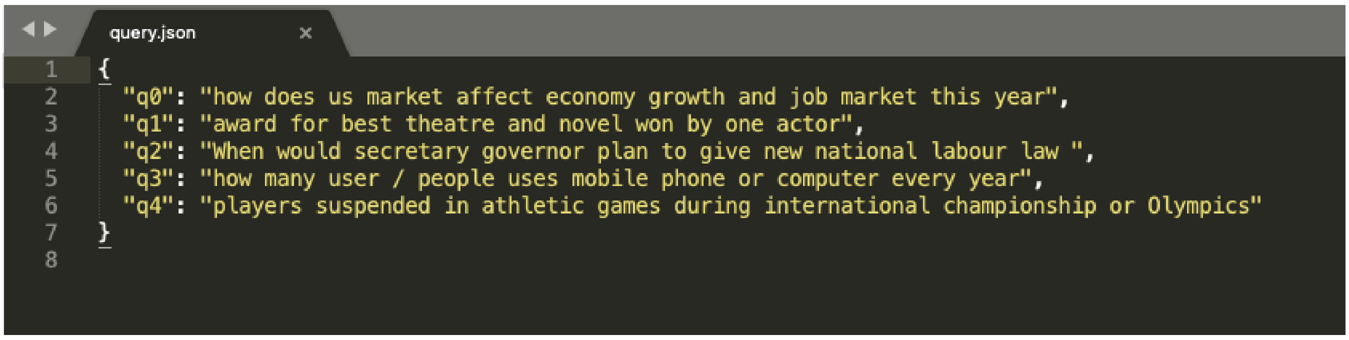


Today	Size	Kind
termDocumentMatrix.csv	7.4 MB	Comma...et (.csv)
invertedIndex.csv	2.3 MB	Comma...et (.csv)

The term-document matrix is of 7.4MB, and the inverted index is of 2.3MB.

3.3 Benchmark Queries

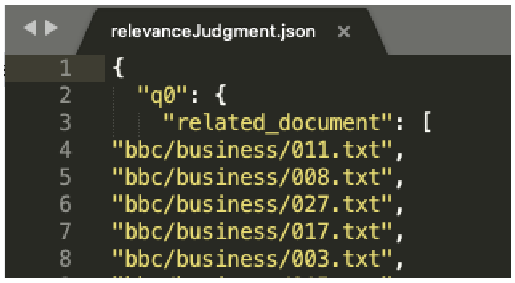
A set of 5 benchmark queries, each roughly belonging to each category, are fed to the IR system as a JSON file, the queries are designed in such a fashion that the entire data is explored, the queries are -



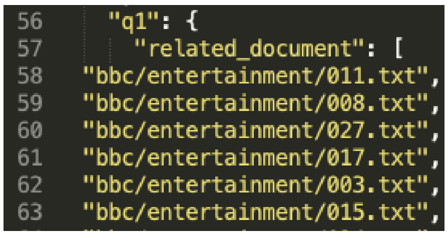
```
{
  "q0": "how does us market affect economy growth and job market this year",
  "q1": "award for best theatre and novel won by one actor",
  "q2": "When would secretary governor plan to give new national labour law ",
  "q3": "how many user / people uses mobile phone or computer every year",
  "q4": "players suspended in athletic games during international championship or Olympics"
}
```

3.4 Relevance Judgements

Relevance Judgement is evaluating what all documents are relevant and which all are not for a particular query. Here manually, we have specified all the relevant documents for a particular query for all the five queries and specified this information in a JSON file that the IR system will use for performance evaluation. All the documents belonging to a particular category have been marked relevant to the benchmark query belonging to that category and the rest as non-relevant. Hence, each query has 50 relevant and 200 non-relevant documents. The snippet of the JSON file:



```
{
  "q0": {
    "related_document": [
"bbc/business/011.txt",
"bbc/business/008.txt",
"bbc/business/027.txt",
"bbc/business/017.txt",
"bbc/business/003.txt",
```



```
  "q1": {
    "related_document": [
"bbc/entertainment/011.txt",
"bbc/entertainment/008.txt",
"bbc/entertainment/027.txt",
"bbc/entertainment/017.txt",
"bbc/entertainment/003.txt",
"bbc/entertainment/015.txt",
```

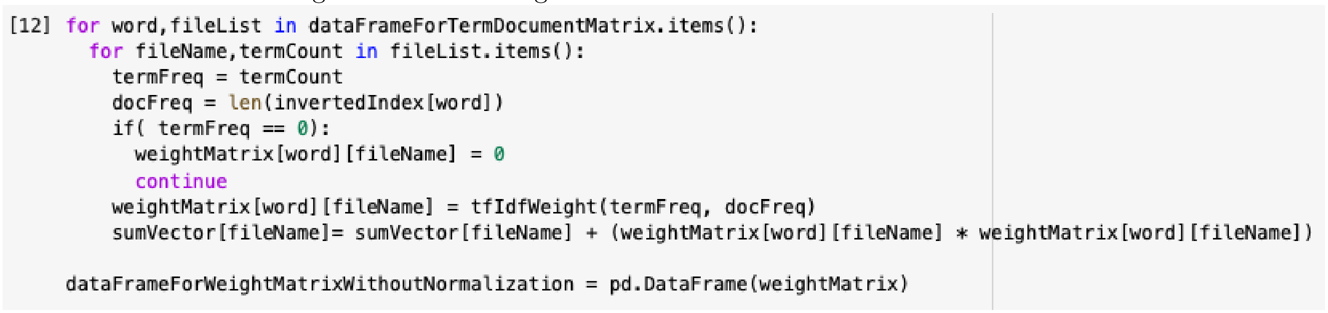
and similarly for the rest.

3.5 TF-IDF,Normalization for Document,Query

The queries, the term-document matrix, inverted index and the relevance judgement are now ready for tf-idf tagging. The ultimate aim is to find the weight matrix that includes finding TF and IDF values where TF refers to term frequency as to how many times a term occurs in a document. Furthermore, IDF refers to inverse document frequency, which means the count of documents that contain a particular word.
The formulae that is used to find the TF-IDF weight is -

```
TF weight = 1 + log(term frequency)
IDF weight = log(total number of documents/document frequency)
TF-IDF weight = TF weight * IDF weight
```

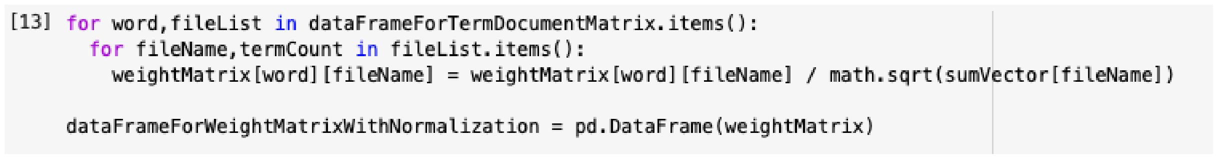
And with this TF-IDF weight we find the weight matrix -



```
[12] for word,fileList in dataFrameForTermDocumentMatrix.items():
       for fileName,termCount in fileList.items():
         termFreq = termCount
         docFreq = len(invertedIndex[word])
         if( termFreq == 0):
           weightMatrix[word][fileName] = 0
           continue
         weightMatrix[word][fileName] = tfIdfWeight(termFreq, docFreq)
         sumVector[fileName]= sumVector[fileName] + (weightMatrix[word][fileName] * weightMatrix[word][fileName])

     dataFrameForWeightMatrixWithoutNormalization = pd.DataFrame(weightMatrix)
```

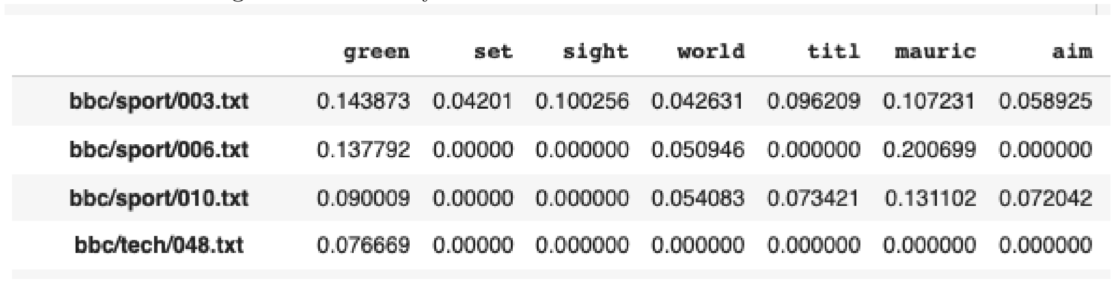
In the above code, we find the weight Matrix that contains the tf*idf value for each word v/s documents.

After having the weight matrix, it is necessary to NORMALIZE the weight matrix so that the angle measure or the COSINE similarity can be used as a similarity measure instead of length as a measure. We perform the Normalization by the following piece of code -



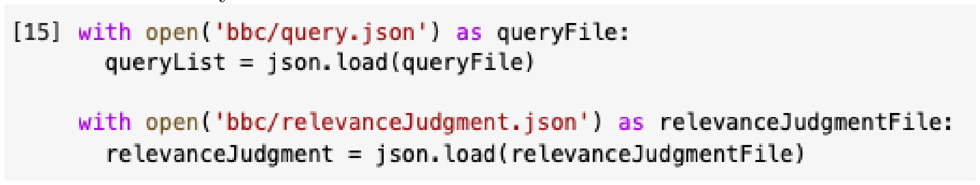
```
[13] for word,fileList in dataFrameForTermDocumentMatrix.items():
       for fileName,termCount in fileList.items():
         weightMatrix[word][fileName] = weightMatrix[word][fileName] / math.sqrt(sumVector[fileName])

     dataFrameForWeightMatrixWithNormalization = pd.DataFrame(weightMatrix)
```

The normalised weight matrix finally looks like -



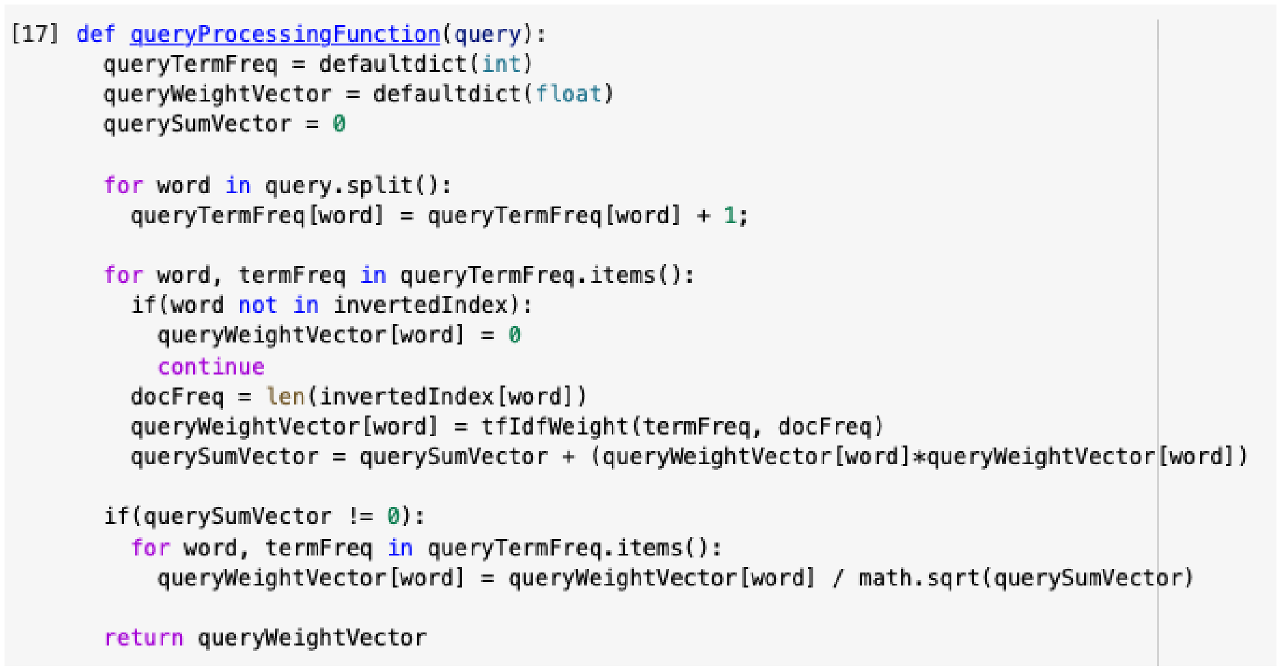
	green	set	sight	world	titl	mauric	aim
bbc/sport/003.txt	0.143873	0.04201	0.100256	0.042631	0.096209	0.107231	0.058925
bbc/sport/006.txt	0.137792	0.00000	0.000000	0.050946	0.000000	0.200699	0.000000
bbc/sport/010.txt	0.090009	0.00000	0.000000	0.054083	0.073421	0.131102	0.072042
bbc/tech/048.txt	0.076669	0.00000	0.000000	0.000000	0.000000	0.000000	0.000000

Now we have made our weight matrix; now, our IR system is ready and ready to accept queries and deliver results.
Hence we load the query.json file and the relevance judgement file also since our final aim is to find the precision of the system .
We load the files by -



```
[15] with open('bbc/query.json') as queryFile:
       queryList = json.load(queryFile)

     with open('bbc/relevanceJudgment.json') as relevanceJudgmentFile:
       relevanceJudgment = json.load(relevanceJudgmentFile)
```

Like we processed our corpus, the same logic of processing needs to be done on the queries so that tokens are similar and can be matched with each other. To do the same processing on the queries, we run the following piece of code -

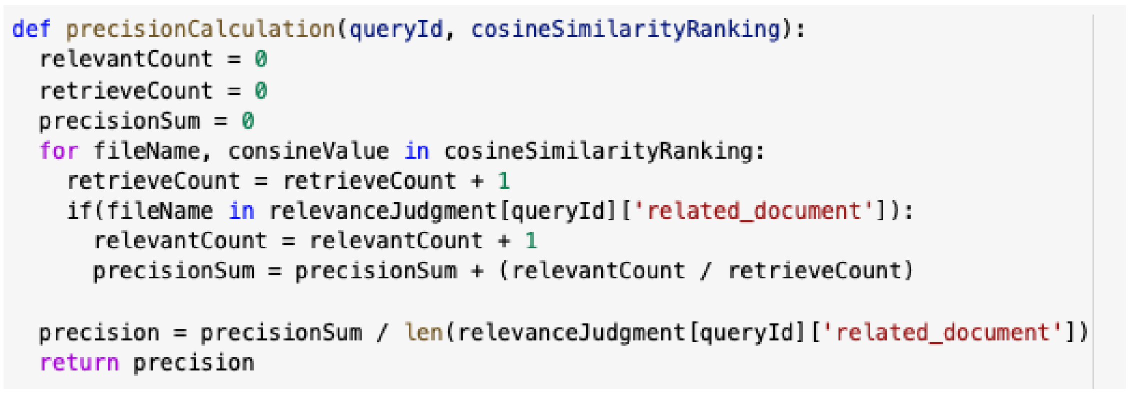


```
[17] def queryProcessingFunction(query):
       queryTermFreq = defaultdict(int)
       queryWeightVector = defaultdict(float)
       querySumVector = 0

       for word in query.split():
         queryTermFreq[word] = queryTermFreq[word] + 1;

       for word, termFreq in queryTermFreq.items():
         if(word not in invertedIndex):
           queryWeightVector[word] = 0
           continue
         docFreq = len(invertedIndex[word])
         queryWeightVector[word] = tfIdfWeight(termFreq, docFreq)
         querySumVector = querySumVector + (queryWeightVector[word]*queryWeightVector[word])

       if(querySumVector != 0):
         for word, termFreq in queryTermFreq.items():
           queryWeightVector[word] = queryWeightVector[word] / math.sqrt(querySumVector)

       return queryWeightVector
```

Here the query is processed, and the weight matrix, just like for the corpus, is made for the query list and tf and idf multiplication and then finally normalizes each query by query sum vector.

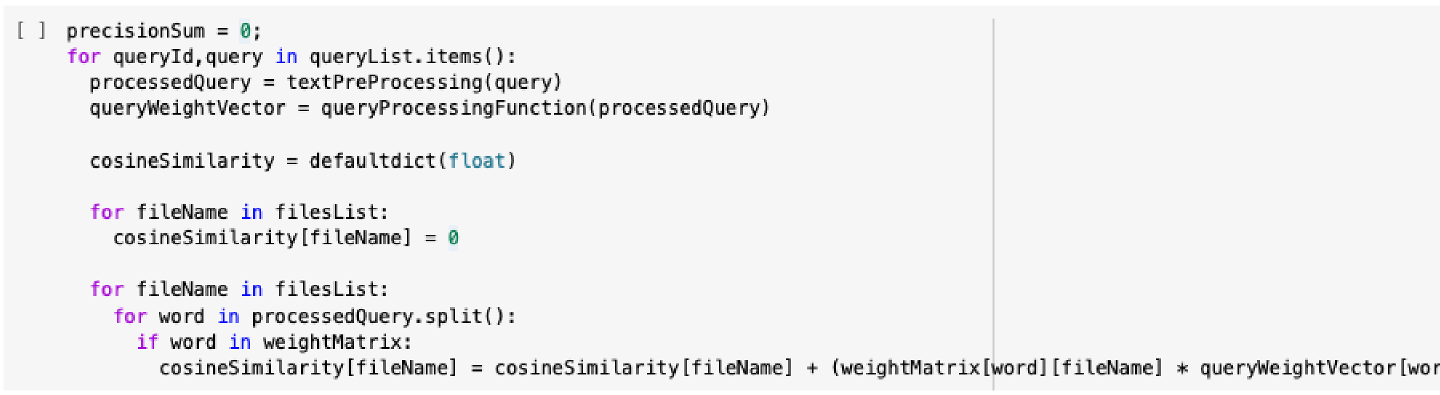
3.6 Precision Calculation

Firstly we implement the function that calculates the precision, and then we will be sending it the values for a particular query (id) and the ranking based on cosine similarity.



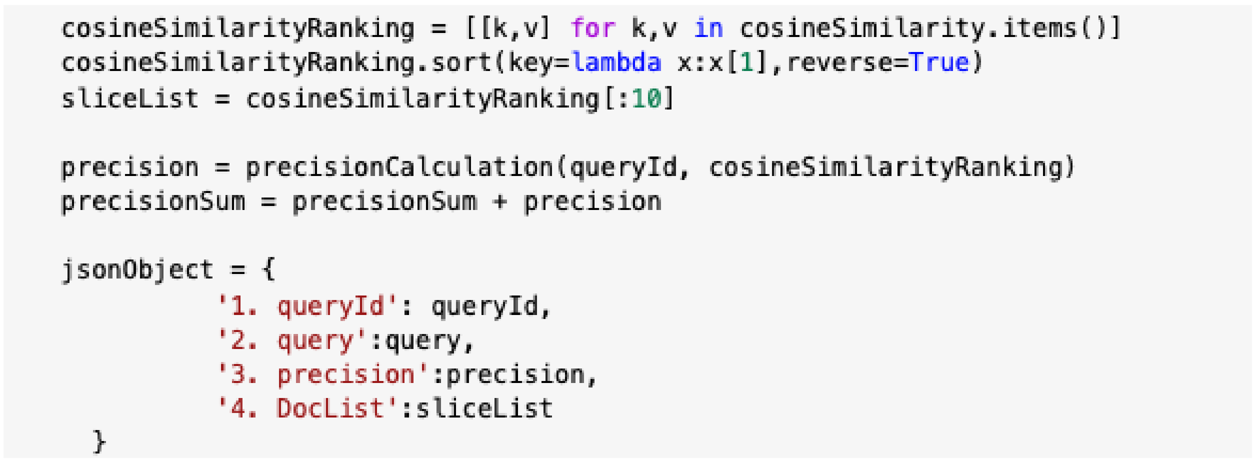
```
def precisionCalculation(queryId, cosineSimilarityRanking):
  relevantCount = 0
  retrieveCount = 0
  precisionSum = 0
  for fileName, consineValue in cosineSimilarityRanking:
    retrieveCount = retrieveCount + 1
    if(fileName in relevanceJudgment[queryId]['related_document']):
      relevantCount = relevantCount + 1
      precisionSum = precisionSum + (relevantCount / retrieveCount)

  precision = precisionSum / len(relevanceJudgment[queryId]['related_document'])
  return precision
```

Now in the cosine similarity matrix, everything is initialized to 0 and then iteratively, the weight value from the weight matrix of documents and the weight value from the query weight matrix are multiplied if a match of words is found in the query and the corpus. Moreover, this goes on for each word. This calculates our cosine similarity .
This entire thing is implemented by the following code:



```
[ ] precisionSum = 0;
    for queryId,query in queryList.items():
      processedQuery = textPreProcessing(query)
      queryWeightVector = queryProcessingFunction(processedQuery)

      cosineSimilarity = defaultdict(float)

      for fileName in filesList:
        cosineSimilarity[fileName] = 0

      for fileName in filesList:
        for word in processedQuery.split():
          if word in weightMatrix:
            cosineSimilarity[fileName] = cosineSimilarity[fileName] + (weightMatrix[word][fileName] * queryWeightVector[wor
```

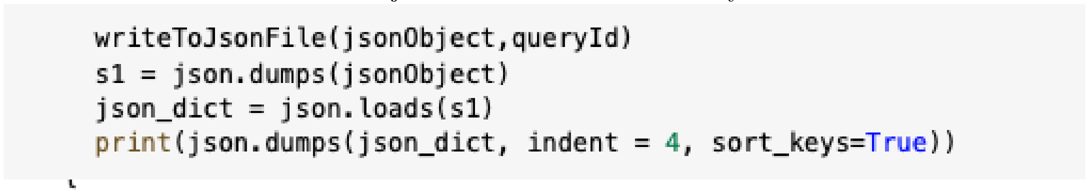
Now the precision calculation function present on the top of this page is called. For this ranking based on the cosine similarity, the top 10 documents, the system's average precision is stored into a folder named as - result, which will automatically be made when we run our google colab notebook. It contains five JSON files, each having the top 10 documents based on cosine similarity, the cosine values, and the system's average precision. This is achieved by the following code:



```
  cosineSimilarityRanking = [[k,v] for k,v in cosineSimilarity.items()]
  cosineSimilarityRanking.sort(key=lambda x:x[1],reverse=True)
  sliceList = cosineSimilarityRanking[:10]

  precision = precisionCalculation(queryId, cosineSimilarityRanking)
  precisionSum = precisionSum + precision

  jsonObject = {
              '1. queryId': queryId,
              '2. query':query,
              '3. precision':precision,
              '4. DocList':sliceList
    }
```

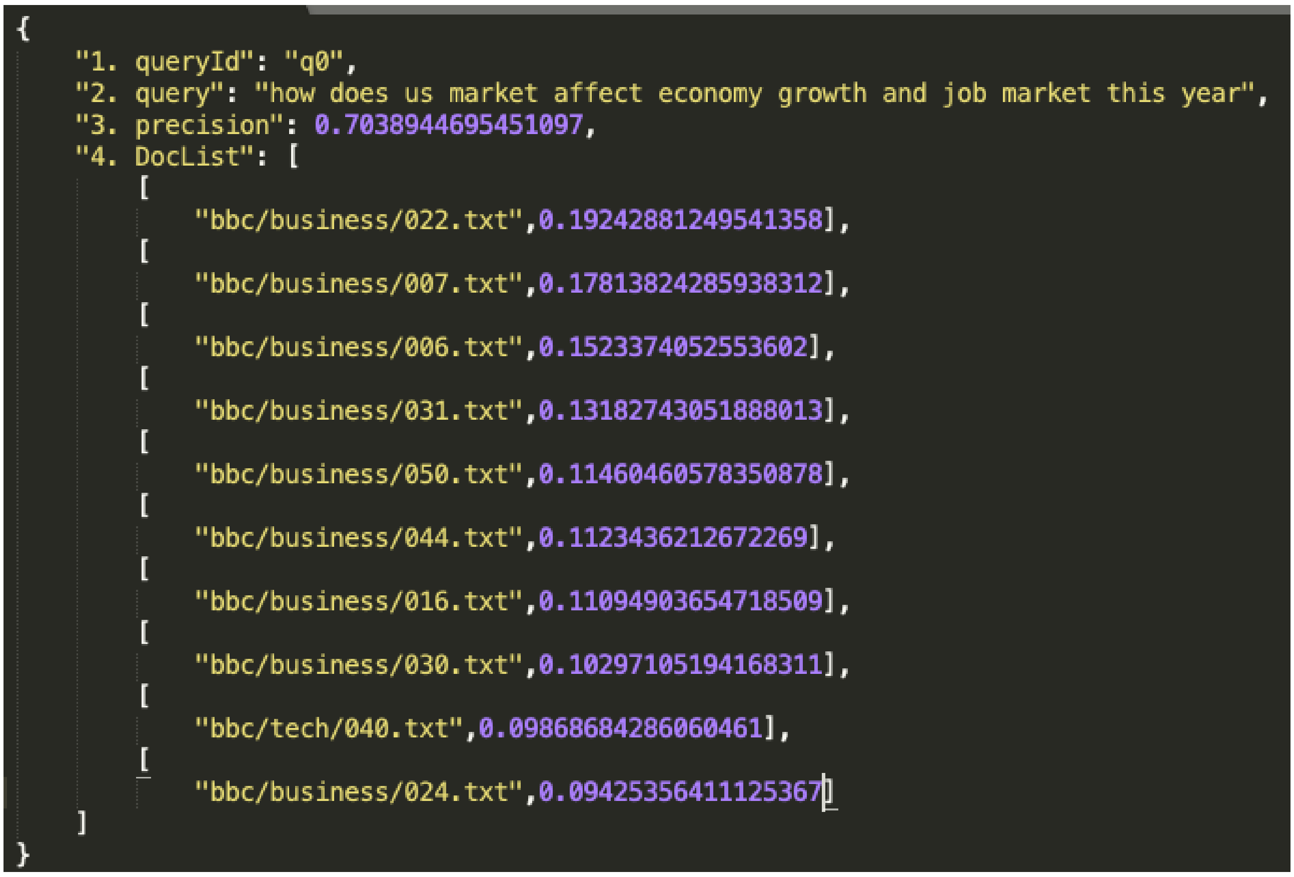
We were told to calculate this for only 1 query, but we have done it for all five queries.
This all information is stored under the results folder having five files each corresponding to a query.

The result folder which has 5 json files which are written by a function:



```
writeToJsonFile(jsonObject,queryId)
s1 = json.dumps(jsonObject)
json_dict = json.loads(s1)
print(json.dumps(json_dict, indent = 4, sort_keys=True))
```

Out of these five files, one file q0.json corresponding to 1st query looks like:

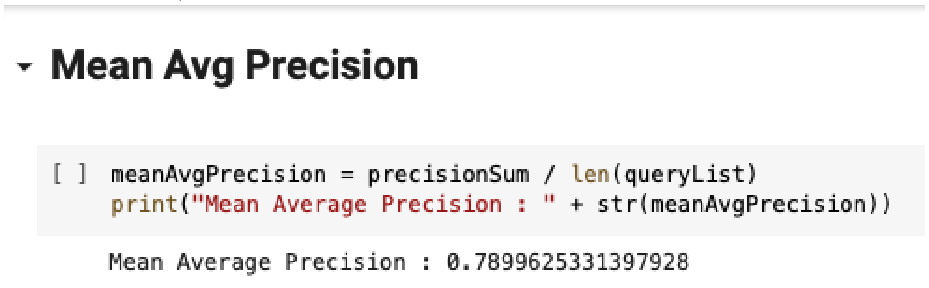


```
{
    "1. queryId": "q0",
    "2. query": "how does us market affect economy growth and job market this year",
    "3. precision": 0.7038944695451097,
    "4. DocList": [
        [
            "bbc/business/022.txt",0.19242881249541358],
        [
            "bbc/business/007.txt",0.17813824285938312],
        [
            "bbc/business/006.txt",0.1523374052553602],
        [
            "bbc/business/031.txt",0.13182743051888013],
        [
            "bbc/business/050.txt",0.11460460578350878],
        [
            "bbc/business/044.txt",0.1123436212672269],
        [
            "bbc/business/016.txt",0.11094903654718509],
        [
            "bbc/business/030.txt",0.10297105194168311],
        [
            "bbc/tech/040.txt",0.09868684286060461],
        [
            "bbc/business/024.txt",0.09425356411125367]
    ]
}
```

Interpretation of this file - This means that for query: "how does US market affect economy growth and job market this year" we have top documents belonging to business category - document number - 22,7,6,31,50,44,16,30 and 24. Also, one document from tech - document 40 is retrieved.
So for this our PRECISION @10 = 9/10 = 0.9 since 9 relevant (business category) and 10 total .
The average precision for this query as written in JSON file - 0.7038.
Similar files belonging to q1,q2,q3,q4 can be found in the result folder.

3.7 Mean Average Precision

Finally, towards the end of our assignment, we calculated the mean average precision for all the queries. That is the precision for all the queries averaged, representing the Mean Average Precision of the system for this particular query set. This looks like -



Mean Avg Precision

```
[ ]  meanAvgPrecision = precisionSum / len(queryList)
     print("Mean Average Precision : " + str(meanAvgPrecision))

     Mean Average Precision : 0.7899625331397928
```

4 Conclusion

We successfully developed an IR system with Mean Average Precision as - 0.789962 or 78.996 per cent for the query set q0,q1,q2,q3,q4 stored in query.json in bbc.zip file. Different queries can be used to find the precision of the system as well. The google colab notebook can be found here - click me
And the bbc.zip can be found here -click me How to run the notebook ?
Just upload the bbc.zip file in the google colab environment, and then it is good to go. Click on the run all under Runtime Tab, and everything will be done.

5 References

The following resources were used in making this project -
1.LATEX TUTORIAL
2.LATEX TUTORIAL -2
3.REFERENCE FROM STACK OVERFLOW

6 A Note of Thanks

A Big Note of Thanks to our instructors, MRS. Preety Singh mam and Suvidha mam who constantly guided us and maintained effective communication through emails in this pandemic.
We are truly grateful to have teachers like you. Thank you so much.

PRIYANK SONKIYA 17DCS009

RACHIT JAIN 17UCS118