PROJECT REPORT ON TF-IDF Query Retrieval on Shakespeare's Corpus

A Project Work Submitted in Partial Fulfilment of the requirements for

The Course

ALGORITHMS FOR INTELLIGENCE WEB AND INFORMATION RETRIEVAL

By

Manoj Mahesh Patil	**PES2201800656**
Hrithik Diwakar	**PES2201800666**
Sumukh R.R	**PES2201800078**

Under the supervision of

Prof. Nagegowda K S

Associate Professor

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

PES UNIVERSITY

EC CAMPUS

TABLE OF CONTENTS:

1. Introduction

Term Frequency is defined as the ratio of the number of times the "Term" appeared in a document to the total number of Terms in the document.

Inverse Document Frequency is the ratio of Total Number of Documents to the Number of Documents Containing the Term.

The product of the Term Frequency and Inverse Document Frequency is TF-IDF. This method is widely used for Information Retrieval and Text Mining. Computer can understand data in the form of numerical values , so for this reason we vectorize all of the text so the computer can understand better. By vectorizing the documents we can perform various numerical operations like ranking , clustering and so on. This is the same thing. This is the same thing that happens when you perform a Google search. The web pages are called Documents and the search text is called Query. When you search with a query, google will find the relevance of the query with all of the documents, ranks them in the order of relevance and shows you the top k documents.

2. Overview of the Project

The data that we are using is Shakespeare's Poems Corpus in a folder called **Stories** . The folder contains 23 Shakespeare's Poems in .txt format . Each document has different names. There is also an **index.html** file which contains the name and the location where all these poems are stored which can be used for efficient query retrieval.

The project is not only limited to Shakespeare's Corpus , it can also be applied to any corpus that is in text (i.e. .txt) format.

3. Algorithm and Pseudocode

As mentioned in the title we will be using TF-IDF as well as Cosine similarities to rank documents. In order to rank the documents according to their TF-IDF / Cosine similarities we have to first convert the input string to a set of Tokens viz called as Tokenization.

For this , we follow the below steps :

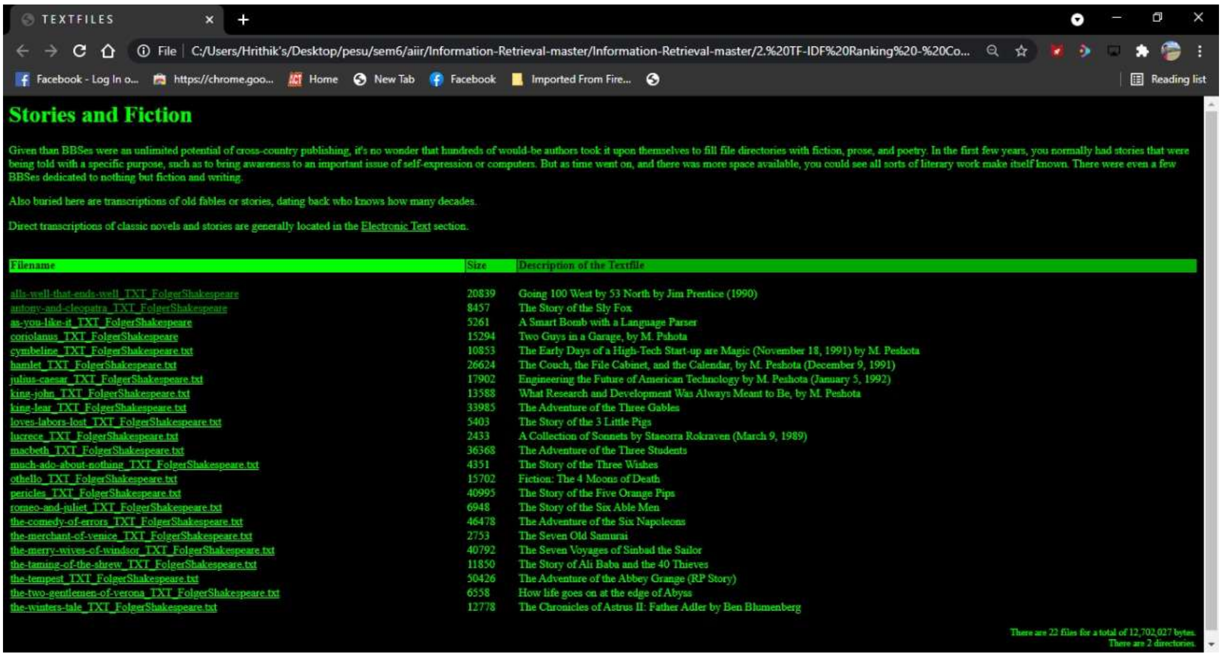
- We convert all words into lower case.
- Stop word removal using **ntlk** library.
- Punctuation removal.
- Elimination of Single Characters.
- Conversion of Numbers to Alphabets (For ex. $100 is converted to hundred dollars).
- Lemmatisation.
- And the most important part of pre-processing , Stemming.

Finally , after doing all this we calculate :

- TF-IDF = Term-Frequency *Inverse Document Frequency
- Cosine Similarity = Dot Product of vector a and vector b.

Since TF-IDF is a way to measure the importance of tokens in text and Cosine Similarity is used to compare the similarity of the query words with those present in other documents , we use Cosine Similarity as our final ranking parameter.

The **index.html** file:


TEXTFILES
File | C:/Users/Hrithik's/Desktop/pesu/sem6/aiir/Information-Retrieval-master/Information-Retrieval-master/2.%20TF-IDF%20Ranking%20-%20Co...
Facebook - Log In o...
https://chrome.goo...
Home
New Tab
Facebook
Imported From Fire...
Reading list

Stories and Fiction

Given than BBSes were an unlimited potential of cross-country publishing, it's no wonder that hundreds of would-be authors took it upon themselves to fill file directories with fiction, prose, and poetry. In the first few years, you normally had stories that were being told with a specific purpose, such as to bring awareness to an important issue of self-expression or computers. But as time went on, and there was more space available, you could see all sorts of literary work make itself known. There were even a few BBSes dedicated to nothing but fiction and writing.

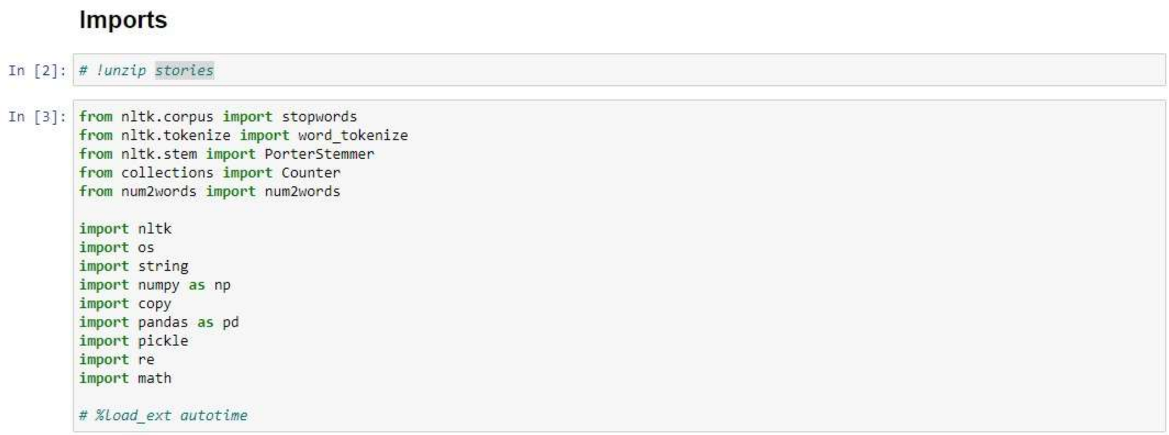
Also buried here are transcriptions of old fables or stories, dating back who knows how many decades.

Direct transcriptions of classic novels and stories are generally located in the Electronic Text section.

Filename	Size	Description of the Textfile
alls-well-that-ends-well_TXT_FolgerShakespeare	20839	Going 100 West by 53 North by Jim Prentice (1990)
antony-and-cleopatra_TXT_FolgerShakespeare	8457	The Story of the Sly Fox
as-you-like-it_TXT_FolgerShakespeare	5261	A Smart Bomb with a Language Parser
coriolanus_TXT_FolgerShakespeare	15294	Two Guys in a Garage, by M. Pshota
cymbeline_TXT_FolgerShakespeare.txt	10853	The Early Days of a High-Tech Start-up are Magic (November 18, 1991) by M. Peshota
hamlet_TXT_FolgerShakespeare.txt	26624	The Couch, the File Cabinet, and the Calendar, by M. Peshota (December 9, 1991)
julius-caesar_TXT_FolgerShakespeare.txt	17902	Engineering the Future of American Technology by M. Peshota (January 5, 1992)
king-john_TXT_FolgerShakespeare.txt	13588	What Research and Development Was Always Meant to Be, by M. Peshota
king-lear_TXT_FolgerShakespeare.txt	33985	The Adventure of the Three Gables
loves-labors-lost_TXT_FolgerShakespeare.txt	5403	The Story of the 3 Little Pigs
lucrece_TXT_FolgerShakespeare.txt	2433	A Collection of Sonnets by Staeorra Rokraven (March 9, 1989)
macbeth_TXT_FolgerShakespeare.txt	36368	The Adventure of the Three Students
much-ado-about-nothing_TXT_FolgerShakespeare.txt	4351	The Story of the Three Wishes
othello_TXT_FolgerShakespeare.txt	15702	Fiction: The 4 Moons of Death
pericles_TXT_FolgerShakespeare.txt	40995	The Story of the Five Orange Pips
romeo-and-juliet_TXT_FolgerShakespeare.txt	6948	The Story of the Six Able Men
the-comedy-of-errors_TXT_FolgerShakespeare.txt	46478	The Adventure of the Six Napoleons
the-merchant-of-venice_TXT_FolgerShakespeare.txt	2753	The Seven Old Samurai
the-merry-wives-of-windsor_TXT_FolgerShakespeare.txt	40792	The Seven Voyages of Sinbad the Sailor
the-taming-of-the-shrew_TXT_FolgerShakespeare.txt	11850	The Story of Ali Baba and the 40 Thieves
the-tempest_TXT_FolgerShakespeare.txt	50426	The Adventure of the Abbey Grange (RP Story)
the-two-gentlemen-of-verona_TXT_FolgerShakespeare.txt	6558	How life goes on at the edge of Abyss
the-winters-tale_TXT_FolgerShakespeare.txt	12778	The Chronicles of Astrus II: Father Adler by Ben Blumenberg

There are 22 files for a total of 12,702,027 bytes.
There are 2 directories.

The libraries that we need :



Imports

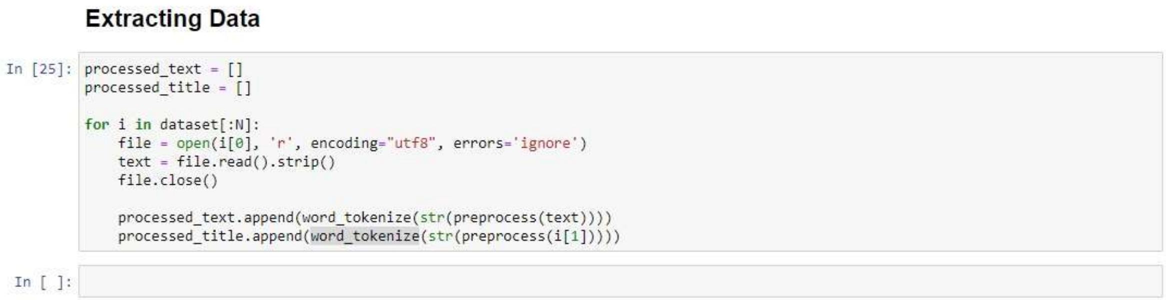
```
In [2]: # !unzip stories
```

```
In [3]: from nltk.corpus import stopwords
        from nltk.tokenize import word_tokenize
        from nltk.stem import PorterStemmer
        from collections import Counter
        from num2words import num2words

        import nltk
        import os
        import string
        import numpy as np
        import copy
        import pandas as pd
        import pickle
        import re
        import math

        # %load_ext autotime
```

Pre-processing :

Step 1 :Extracting Data:

All the corpus is stored in the **stories** zip folder in .txt format . First we need to extract them

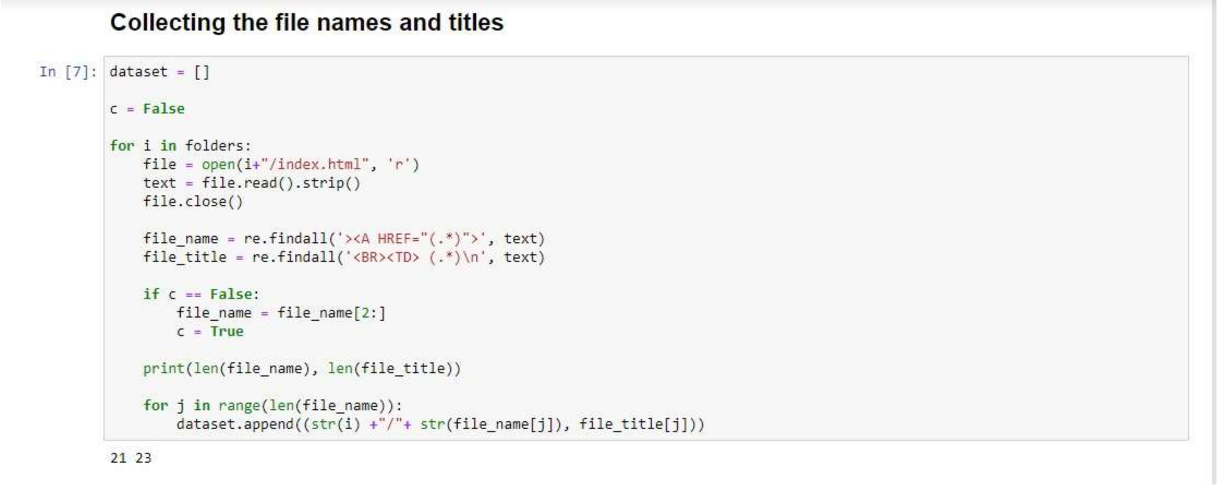


Extracting Data

```
In [25]: processed_text = []
         processed_title = []

         for i in dataset[:N]:
             file = open(i[0], 'r', encoding="utf8", errors='ignore')
             text = file.read().strip()
             file.close()

             processed_text.append(word_tokenize(str(preprocess(text))))
             processed_title.append(word_tokenize(str(preprocess(i[1]))))

In [ ]:
```

Step 2 : Collecting File Names and Titles :

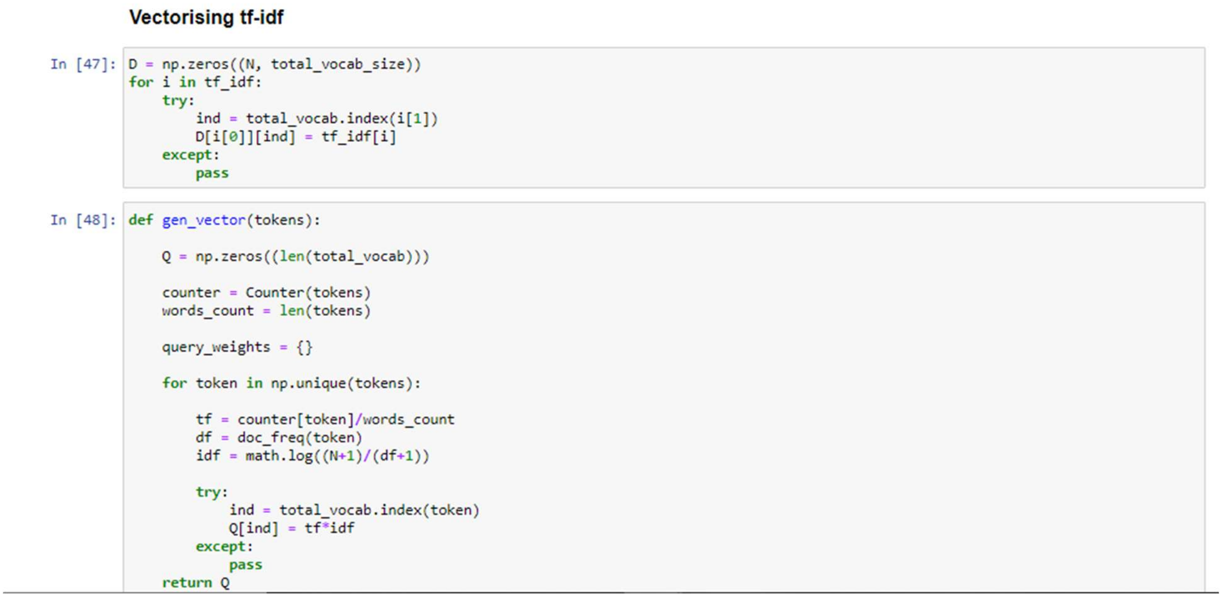
Next , we create an empty list called **dataset** which contains the titles of all the .txt files in the stories folder.



Collecting the file names and titles

```
In [7]: dataset = []

        c = False

        for i in folders:
            file = open(i+"/index.html", 'r')
            text = file.read().strip()
            file.close()

            file_name = re.findall('><A HREF="(.*)">', text)
            file_title = re.findall('<BR><TD> (.*)\n', text)

            if c == False:
                file_name = file_name[2:]
                c = True

            print(len(file_name), len(file_title))

            for j in range(len(file_name)):
                dataset.append((str(i) +"/"+ str(file_name[j]), file_title[j]))

        21 23
```

Step 3 : Converting Strings to Numbers



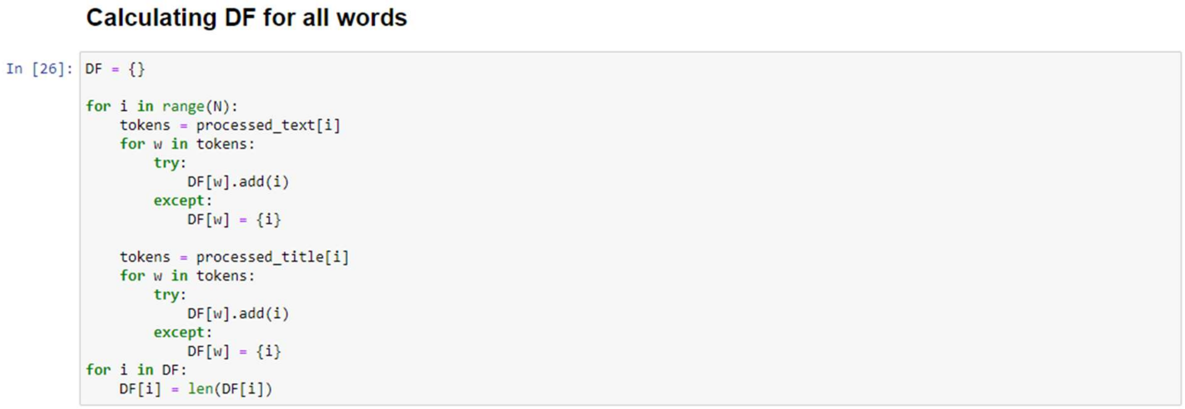
Vectorising tf-idf

```
In [47]: D = np.zeros((N, total_vocab_size))
         for i in tf_idf:
             try:
                 ind = total_vocab.index(i[1])
                 D[i[0]][ind] = tf_idf[i]
             except:
                 pass
```

```
In [48]: def gen_vector(tokens):

             Q = np.zeros((len(total_vocab)))

             counter = Counter(tokens)
             words_count = len(tokens)

             query_weights = {}

             for token in np.unique(tokens):

                 tf = counter[token]/words_count
                 df = doc_freq(token)
                 idf = math.log((N+1)/(df+1))

                 try:
                     ind = total_vocab.index(token)
                     Q[ind] = tf*idf
                 except:
                     pass
             return Q
```

Step 4 : Calculating Document Frequency

Document frequency is defined as the ratio of the Number of Documents containing the Term to the Total number of Documents.

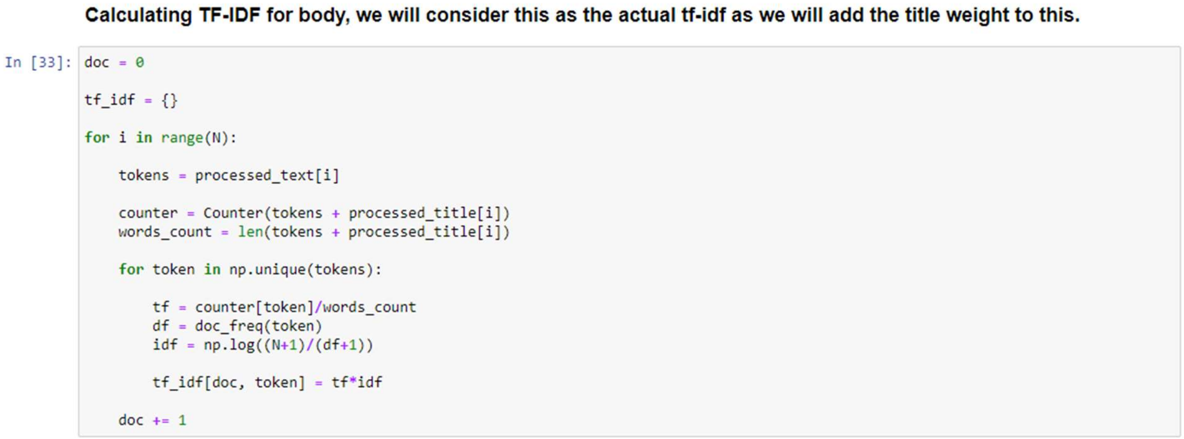


Calculating DF for all words

```
In [26]: DF = {}

         for i in range(N):
             tokens = processed_text[i]
             for w in tokens:
                 try:
                     DF[w].add(i)
                 except:
                     DF[w] = {i}

             tokens = processed_title[i]
             for w in tokens:
                 try:
                     DF[w].add(i)
                 except:
                     DF[w] = {i}
         for i in DF:
             DF[i] = len(DF[i])
```

Step 5 : Calculate TF-IDF of the body.

We calculate the TF-IDF value to the body of the document , ignoring the Terms in the Title.

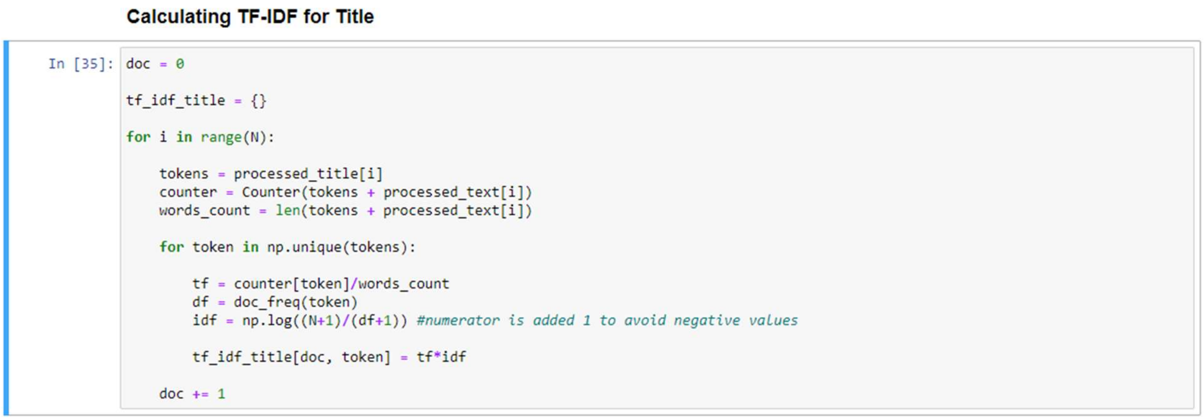


Calculating TF-IDF for body, we will consider this as the actual tf-idf as we will add the title weight to this.

```
In [33]: doc = 0

         tf_idf = {}

         for i in range(N):

             tokens = processed_text[i]

             counter = Counter(tokens + processed_title[i])
             words_count = len(tokens + processed_title[i])

             for token in np.unique(tokens):

                 tf = counter[token]/words_count
                 df = doc_freq(token)
                 idf = np.log((N+1)/(df+1))

                 tf_idf[doc, token] = tf*idf

             doc += 1
```

Step 6 : Calculate TF-IDF of the Title.

We calculate the TF-IDF value of the Title , ignoring the Terms present in the Body of the document.

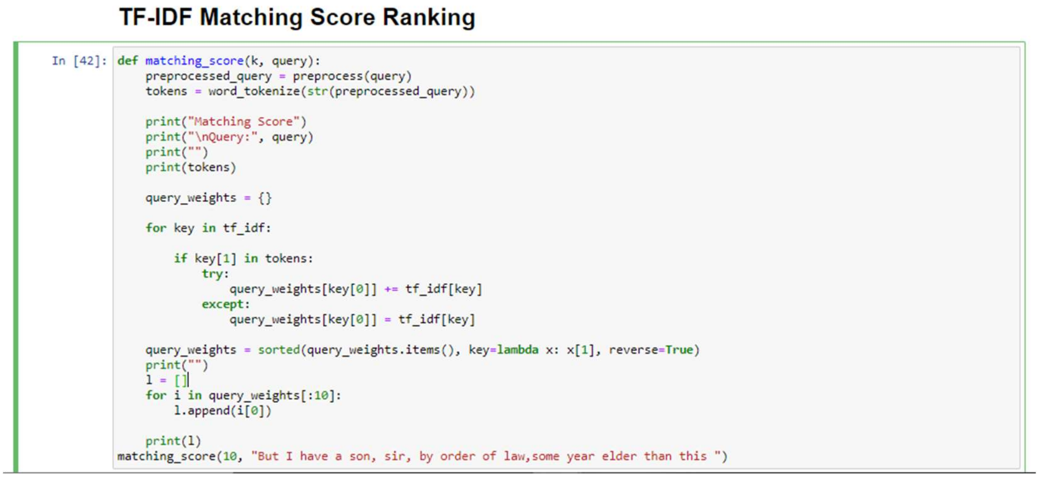


Calculating TF-IDF for Title

```
In [35]: doc = 0

         tf_idf_title = {}

         for i in range(N):

             tokens = processed_title[i]
             counter = Counter(tokens + processed_text[i])
             words_count = len(tokens + processed_text[i])

             for token in np.unique(tokens):

                 tf = counter[token]/words_count
                 df = doc_freq(token)
                 idf = np.log((N+1)/(df+1)) #numerator is added 1 to avoid negative values

                 tf_idf_title[doc, token] = tf*idf

             doc += 1
```

Step 7 : Calculating TF-IDF Matching Score

We multiply the weight alpha to the body and 1-alpha to the title and find the Matching Score.

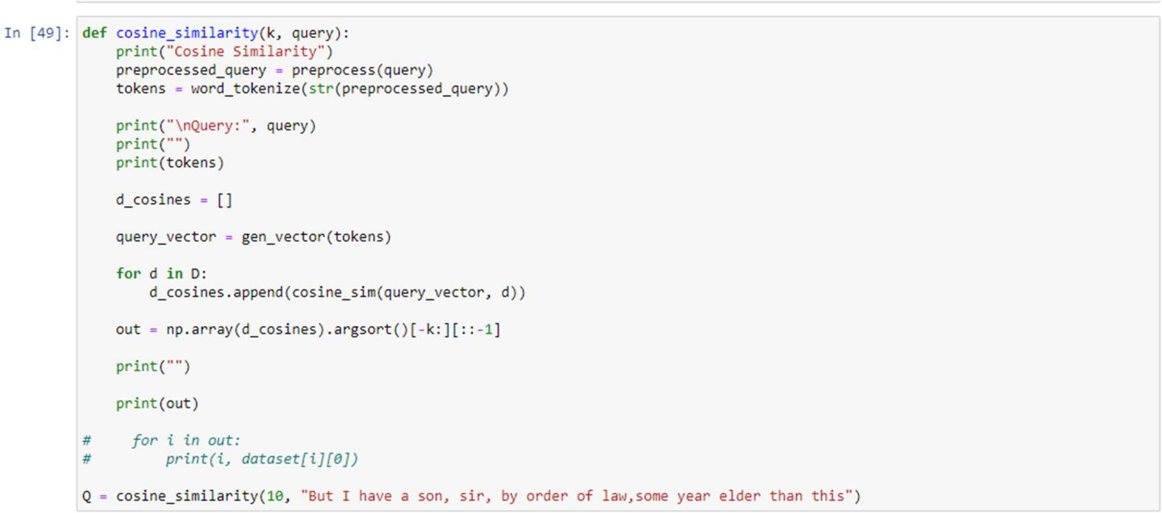


TF-IDF Matching Score Ranking

```
In [42]: def matching_score(k, query):
             preprocessed_query = preprocess(query)
             tokens = word_tokenize(str(preprocessed_query))

             print("Matching Score")
             print("\nQuery:", query)
             print("")
             print(tokens)

             query_weights = {}

             for key in tf_idf:

                 if key[1] in tokens:
                     try:
                         query_weights[key[0]] += tf_idf[key]
                     except:
                         query_weights[key[0]] = tf_idf[key]

             query_weights = sorted(query_weights.items(), key=lambda x: x[1], reverse=True)
             print("")
             l = []
             for i in query_weights[:10]:
                 l.append(i[0])

             print(l)
         matching_score(10, "But I have a son, sir, by order of law,some year elder than this ")
```

Step 8 : Calculating Cosine Similarity

We find cosine similarity using dot product



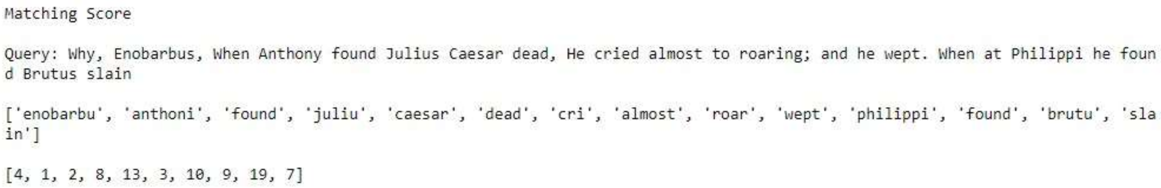
```
In [49]: def cosine_similarity(k, query):
             print("Cosine Similarity")
             preprocessed_query = preprocess(query)
             tokens = word_tokenize(str(preprocessed_query))

             print("\nQuery:", query)
             print("")
             print(tokens)

             d_cosines = []

             query_vector = gen_vector(tokens)

             for d in D:
                 d_cosines.append(cosine_sim(query_vector, d))

             out = np.array(d_cosines).argsort()[-k:][::-1]

             print("")

             print(out)

         #     for i in out:
         #         print(i, dataset[i][0])

         Q = cosine_similarity(10, "But I have a son, sir, by order of law,some year elder than this")
```

Using the above 2 similarity methods we rank the documents.

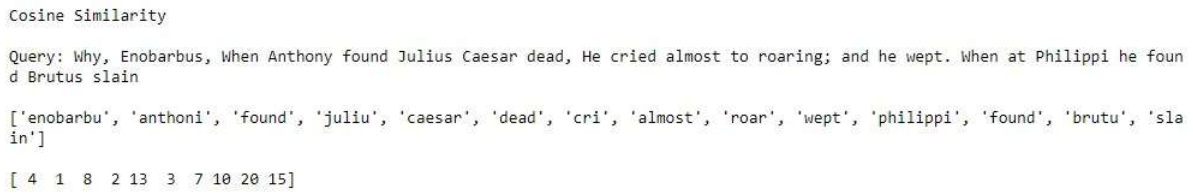
4. Results and Discussions

The output rankings for the following queries are :

Query 1 – "Why , Enobarbus , When Anthony found Julius Caesar dead , He cried almost to roaring ; and he wept . When at Philippi he found Brutus slain "



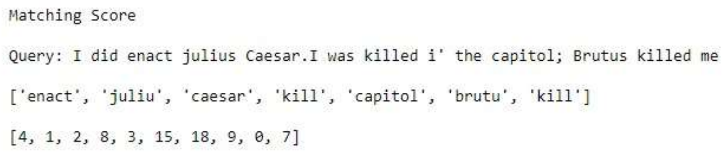
```
Matching Score

Query: Why, Enobarbus, When Anthony found Julius Caesar dead, He cried almost to roaring; and he wept. When at Philippi he foun
d Brutus slain

['enobarbu', 'anthoni', 'found', 'juliu', 'caesar', 'dead', 'cri', 'almost', 'roar', 'wept', 'philippi', 'found', 'brutu', 'sla
in']

[4, 1, 2, 8, 13, 3, 10, 9, 19, 7]
```



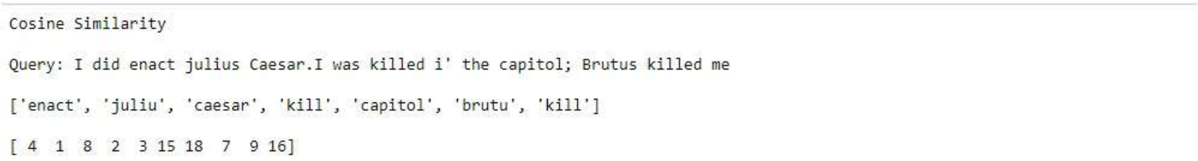
```
Cosine Similarity

Query: Why, Enobarbus, When Anthony found Julius Caesar dead, He cried almost to roaring; and he wept. When at Philippi he foun
d Brutus slain

['enobarbu', 'anthoni', 'found', 'juliu', 'caesar', 'dead', 'cri', 'almost', 'roar', 'wept', 'philippi', 'found', 'brutu', 'sla
in']

[ 4  1  8  2 13  3  7 10 20 15]
```

Hence , seeing the above rankings we can say that Document 4 is highly ranked.

Query 2 – “I did enact Julius Caesar . I was killed i’ the capitol ; Brutus killed me”



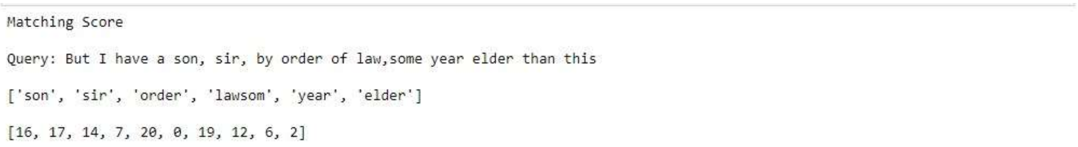
```
Matching Score

Query: I did enact julius Caesar.I was killed i' the capitol; Brutus killed me

['enact', 'juliu', 'caesar', 'kill', 'capitol', 'brutu', 'kill']

[4, 1, 2, 8, 3, 15, 18, 9, 0, 7]
```



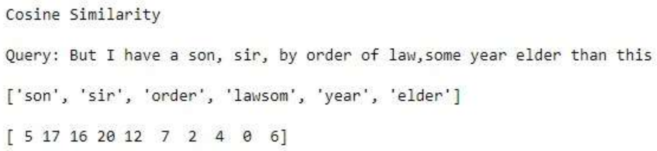
```
Cosine Similarity

Query: I did enact julius Caesar.I was killed i' the capitol; Brutus killed me

['enact', 'juliu', 'caesar', 'kill', 'capitol', 'brutu', 'kill']

[ 4  1  8  2  3 15 18  7  9 16]
```

Hence , seeing the above rankings we can say that Document 4 is highly ranked.

Query 3 – “But I have a son , sir , by order of law , some year older than this”



```
Matching Score

Query: But I have a son, sir, by order of law,some year elder than this

['son', 'sir', 'order', 'lawsom', 'year', 'elder']

[16, 17, 14, 7, 20, 0, 19, 12, 6, 2]
```



```
Cosine Similarity

Query: But I have a son, sir, by order of law,some year elder than this

['son', 'sir', 'order', 'lawsom', 'year', 'elder']

[ 5 17 16 20 12  7  2  4  0  6]
```

Hence , seeing the above rankings we can say that Document 17 is highly ranked.

5. Conclusion and Future Scope

As a part od our 6th semester AIW&IR project we have created a TF-IDF Query Retriever on Shakespeare’s’ Corpus.

There is still room for improvement in our Project , for example our Project is only limited to Corpus which is in .txt format , we can expand our project to include more document types and use more complex algorithms to retrieve queries.

6. References & Sources

We referred the following websites to learn more about how to improve our project:

- www.geeksforgeeks.com
- www.wikipedia.com
- www.stackoverflow.com
- https://cs.stanford.edu