Information Retrieval and Web Search

By

Neeraj B - 17ucs098
Sritej Reddy - 17ucs094

Course Coordinator
Dr. Preety Singh

Department of Computer Science Engineering
The LNM Institute of Information Technology, Jaipur

March 2021

Contents

Chapter *1*

Part- A

1.1 Overview

In this report we would be describing about the Information retrieval system we have created and will be evaluating it using benchmark queries. This report would include Data-set description, Coding Procedure, Benchmark Queries, Relevance Judgements and images of code and output where necessary.

1.2 Data-set Description

The Data-set contains 252 documents divided into five categories namely Freedom Fight, Cricket, Technology Development, Culture, Trending.
We have decided on 5 key words and using those we have generated **top rated links** from search engines like Google and Duck Duck Go. All these links were stored in **CSV format** and later using **jusText** tool from Python we extracted important text part and stored them in individual documents.
Topic Wise Documents Distribution:
1-43 *Freedom Fight*
44-98 *Cricket*
99-150 *Technology Development*
151-221 *Culture*
222-252 *Trending*

Number of Unique terms: **13,426**
Total Number of Terms: **2,59,337**

1.3 Methodology

1.3.1 Using Scapper for obtaining URL's from Search engines

```
from search_engines import Google
engine = Google()
Topic="Type Any topic here"
Num_of_Pages = 7
results = engine.search(Topic, Num_of_Pages)
engine.output("csv")
```

By executing the above code after choosing the Topic and number of pages we get output as a CSV file which contains URLs related to the chosen topic.

query	engine	domain	URL
freedom fight	Google	knowindia.gov.in	https://knowindia.gov.in/culture-and-heritage/freedom-struggle.php
freedom fight	Google	en.wikipedia.org	https://en.wikipedia.org/wiki/Indian_independence_movement
freedom fight	Google	en.wikipedia.org	https://en.wikipedia.org/wiki/Freedom_Fighters
freedom fight	Google	blog.mygov.in	https://blog.mygov.in/freedom-fighters-who-fought-valiantly-in-indias
freedom fight	Google	timesofindia.india	https://timesofindia.indiatimes.com/topic/freedom-fighter

1.3.2 Extracting Text data from URL's

We have used jusText module to extract text data from the URL's

To scrape the main text of web pages while preserving some structure, boilerplate removal is performed on Paragraphs.

Inorder to avoid the code from throwing error when ever a website is unable to be scrapped due to http connection errors or some other errors Try, except blocks are used.

To ensure that empty files are not created in Dataset from scraping the websites os.path.getsize() method is used.

```
import justext
df = pd.read_csv("freedom_fight.csv")
d=1
for i in range(len(df)):
    URL = df.iloc[i, 3]

    try:
        print("at {}" .format(i))
        r=requests.get(URL)
        paragraphs = justext.justext(r.content, justext.get_stoplist("English"))
        s=str(d)+".txt"
        d+=1
        f=open (s,"a",encoding="utf-8")
        for paragraph in paragraphs:
            if not paragraph.is_boilerplate:
                f.write(paragraph.text)
                f.write(" ")
        f.close()
        filesize = os.path.getsize(s)
        if filesize==0:
            d-=1
    except Exception:
        print("connection error at {}".format(i))
```

1.3.3 Preprocessing Text Documents

Firstly tokenize the text data using **word-tokenize from nltk module** followed by removing all the punctuations and non-alphabetic tokens by **isalpha**() method. **Snowball stemmer** is applied on tokens to further pre-process the tokens.

Sample input and output of preprocessing:

```
text="Hard working people does not fear to face struggles"
w=word_tokenize(text)
preProcessed=""
for i in w:
    if(i.isalpha()):
        preProcessed+=sno.stem(i)+" "
```

```
In [17]: preProcessed
Out[17]: 'hard work peopl doe not fear to face struggl '
```

All the Preprocessed documents are stored in the stemming folder.

With stem1.txt being the preprocessed document of text1.txt data and so on.

1.3.4 Count Term Document Matrix and Inverted Index

Both the matrices are created using python dictionaries.
For the Term Document Matrix each key represents the term, and value is a list of 253 size.
With nth index element in the list being the term-frequency(tf) of the key in nth Document.

	Term/ Doc	1	2	3	4	5	6	7	8	9
2	freedom	1	4	10	2	7	5	0	30	0
3	struggl	1	2	2	3	3	2	0	4	0
4	in	48	358	46	16	21	3	2	20	1
5	ancient	3	0	0	0	0	0	0	0	0
6	time	2	13	0	0	0	0	0	0	0
7	peopl	4	19	3	3	0	0	0	2	0
8	from	9	58	5	3	7	2	0	2	1
9	all	5	31	0	1	1	0	0	0	0
10	over	3	17	0	0	2	0	0	2	0

Figure 1.1 Count term-document matrix output showing 9 terms in rows and 9 documents in columns

Similarly in *Inverted Index* each **key** represents the term but **value** is the list of documents in which that term is present.
Document frequency (df) of a term is calculated by taking the length of the term's value list in Inverted index.

Key	Type	Size	Value
abrupt	list	3	[114, 231, 239]
abruzzo	list	1	[187]
absenc	list	3	[44, 109, 249]
absent	list	2	[2, 191]
absolut	list	8	[109, 153, 154, 156, 159, 162, 204, 226]
absolv	list	1	[24]
absorb	list	2	[2, 109]

Figure 1.2 Screenshot of Inverted-Index Dictionary
First term in the above s.s is *'abrupt'* with its document-frequency being 3. It is present in 114.txt, 231.txt, 239.txt

1.3.5 Size Comparision

Both *Count Term Document Matrix and Inverted Index* are stored in json format while Count Term-Document matrix is also stored in CSV format.

Size Comparison of Outputs: Comparing json format output sizes of both matrices

Matrix	*Size*
Term Document count	10,119 KB
Inverted Index	602 KB

Output size of Term Document matrix is approximately 17 times larger than Inverted Index

1.3.6 Calculating Tf-Idf(Term Frequency/Inverted Index)

Term_frequency(TF) is simply calculated from Term_Document_Matrix(TDM) by applying **1+log10(tf)**
Similarly IDF can be obtained from Document-frequency(df) by applying **log10(N/df)**
Number of Documents **N**= 252

```
tfidf={}
for t,values in tdm.items():
    tfidf[t]=[0]*253
    idf=math.log10(252/dfreq[t])
    for i in range(1,253):
        if(tdm[t][i]>0):
            tf=1+math.log10(tdm[t][i])
            tfidf[t][i]=tf*idf
```

Figure 1.3 Screenshot of Tf-IDF stored in csv file

	Term/Doc	1	2	3	4
2	freedom	0.720159	1.153738	1.440319	0.936949
3	struggl	0.896251	1.166049	1.166049	1.323871
4	in	0.12166	0.161256	0.120822	0.100011
5	ancient	1.365273	0	0	0
6	time	0.369661	0.600633	0	0
7	peopl	0.558033	0.79374	0.514514	0.514514
8	from	0.230795	0.326359	0.200647	0.174447

Figure 1.4 Tfidf ['freedom'][1]=0.7205 represent that tf-idf of 'freedom' in 1.txt is 0.7205

1.4 Benchmark Queries

- ”What is trending on twitter”
- ”Recent Technological changes or advancements”
- ”Movements led by Mahatma Gandhi”
- ”Indian festivals different from holi, diwali celebrated across different countries of world”
- ”Live or latest cricket score and news”

Queries are also preprocessed by removing stop-words and non-alphabetic tokens along with applying stemming.

```
for query in Queries:
    query_tokens= word_tokenize(query)
    query_noStopWord = [w for w in query_tokens if not w in stops]
    query_stemmed=[sno.stem(w) for w in query_noStopWord if w.isalpha()]
```

After stemming, Tf-idf is calculated for each query. We are following the **ltc.ltc** weighting scheme.

Calculating vector-length for Documents and Queries:

```
doc_length={}
for i in range(1,253):
    length=0
    for t,v in tdm.items():
        length+=v[i]*v[i]
    doc_length[i]=math.sqrt(length)
```

Key	Type	Size	
10	float	1	51.73
11	float	1	11.0
12	float	1	80.26

Figure 1.5 Vector Doc-lengths for stem10.txt, stem11.txt and stem12.txt respectively are 51.73,11,80.26

Cosine-Similarity between query and Document :

It is calculated by dividing the dot-product (scalar multiplication) of Tf-idf vectors of Document and Query with lengths of query and document.

```
cosine_similarity={}
for i in range(1,253):
    score=0
    for key,value in query_tfidf.items():
        score+= value*tfidf[key][i]
    denominator=doc_length[i]*query_length
    cosine_similarity[i]=score/denominator
```

Key	Type	Size	
66	float	1	0.190
67	float	1	0.022
68	float	1	0.042
69	float	1	0.018

Figure 1.6 Cosine_similarity between query-5 and documents 66,67,68 and 69 are shown in the screen-shot

1.5 Relevance Judgements

1. Our queries are related to 5 different topics in the dataset.
2. Let's take 5th query for example, it is related to cricket topic which belongs to 44-98 docs in the dataset. So, it can be simply marked Not relevant for docs in the dataset.
3. For Documents from 44.txt to 98.txt we have marked the Relevancy and also noted down the key topics in them manually.
4. Like if the topic in doc-44 is about cricket equipment deals then we marked it as store/buy. Similarly if the topic is about a channel or website that gives updates of live news or scores we have marked it as live.
5. Top-10 documents are retrieved from cosine-similarity weights by nlargest() method from heapq module.
6. We have stored relevant document numbers for each query in a text file in the Relevance_Judgements folder.

```
Top10 = nlargest(10, cosine_similarity, key = cosine_similarity.get)
```

```
In [25]: Top10
Out[25]: [66, 70, 41, 84, 243, 63, 72, 62, 80, 149]
```

Figure 1.7 Cosine Similarity between query 5 and documents 66,67,68 and 69 are shown in the screenshot

1.6 Average and Mean Average Precision

Average precision of each query is stored in query_Avg_prec
Mean-Average-precision is directly calculated by taking average of values in query_Avg_prec

```
p=0
precision=0
Avg_pre=0

for i in range(10):
    if str(Top10[i]) in set(relevant_docs):
        p+=1
        precision=p/(i+1)
        Avg_pre+=precision

Avg_pre/=len(relevant_docs)
query_Avg_prec[count]=Avg_pre
```

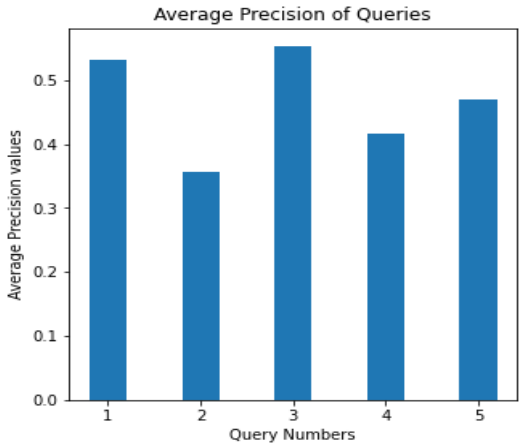

Figure 1.8 Average precisions of all the 5 queries are shown respectively in the above screenshot

```
MAP=0
for key,value in query_Avg_prec.items():
    MAP+=value
MAP=MAP/5
```

MAP for our queries has come out to be 0.466

1.7 Conclusion:

The first step we took to start the project was figure out a way to scrap the search-engines to create the dataset.
Then we found a github repository (**`https://github.com/tasos-py/Search-Engines-Scraper`**) that could be useful for us in obtaining URLs for a given Keyword from search engines. After the first step is done, we spent some days trying out different website scrapers (such as dragnet, newspaper, justext etc.) to see which one is giving meaningful text data to move forward with the project. We faced multiple issues with the scraping module in extracting text data like

1. Code running forever without giving any output on some websites.
   Manually removed those URLs to resolve this.

2. Empty text files returned by module from some URLs
   To resolve this we checked the size of .txt files before storing them in Data-set.

3. Http connection errors over some websites.
   Used try and except blocks for the code to continue running with a printed connection error message at that URL number.

The rest of the tasks, from creating term-document-count matrix to calculating MAP were computed without facing any issues.