Natural Language Processing (CSE4022)

NAT GEO Traveler – Group Activity(Team 5)

Team Members:

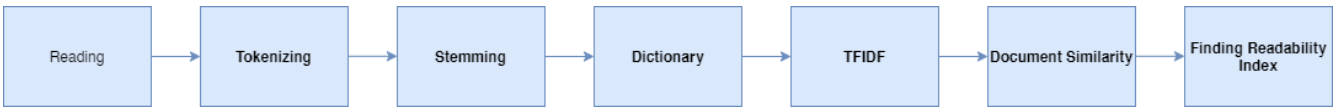
Rakshit Kumar: 18BCE0999
Harshith Chukka: 18BCE0941
Sarthak Banerjee: 18BCE2341
Samartha Pal: 18BCE2434
Abhishek Dhall: 18BCE2278
Keerthivasan: 18BCE2133
Guneshwar Singh Manhas: 18BCE0960
Jeswin Jacob J: 18BCE2121
Shubham Varade: 18BCE2388
Sanjoi Sethi: 18BCE2261

Problem Statement:

Assume you are a part of NLP Tech team that work for a Publshing House. There is a shortlisted applicant (with her writing samples) for Editor-in-chief position. How can you help the publishing house with the decision on hiring this applicant?

Design a pipeline for this problem statement. Showcase the outcomes of step in the pipeline

Pipeline for the Problem Statement:



1. Reading

- **By Web Scraping**

For carrying out web scraping, we have used the url that contains the links to all the articles of Ms. Lakshmi Sankaran. We have requested for that particular web-page and targeted the tags that contain the links to all her articles using a package called 'Beautiful Soup'. After making a list of all those links, we have again targeted the tags that contain the article content in the same way and stored it in a text file. That text file can act as our resource for all of Ms. Sankaran's articles**.**

Code Snippet:

In [ ]:

```
import requests
from bs4 import BeautifulSoup

def getArticle(url):
    req = requests.get(url)
    soup = BeautifulSoup(req.content, 'html5lib')
    title = soup.find('h1')
    print(title.text)
    title = title.text

    filename = "ScrapedArticles.txt"
    f = open(filename, "a")
    f.write(title)
    f.write('\n')

    table = soup.find('div', attrs = {'class':'storyWrap'})
    for row in table.findAll('p'):
        #print(row.text)
        f.write(row.text)
        f.write('\n')
    f.close()
```

```
In [1]: def article():
            url='http://www.natgeotraveller.in/author/lakshmi-sankaran/'

            resp=requests.get(url)
            Links=[]

            if resp.status_code==200:

                soup=BeautifulSoup(resp.text,'html.parser')
                l=soup.find("ul",{"class":"categoryList"})

                for i in l.findAll('div', attrs = {'class': 'cDescription'}):
                    Links.append(str(i.h1.a['href']))

                for i in Links:
                    getArticle(i)
            else:
                print("Error")

        article()

        A Culture Ride Through Chiang Mai and Chiang Rai
```

- **By Building a Custom Corpus**

As a set of text documents, a corpus can be described. You may think of it as just a bunch of text files in a directory, mostly alongside a lot of other text file folders. NLTK already describes nltk.data.path as a list of data paths or directories.
In each of these given routes, our custom corpora must be present so that it can be identified by NLTK. In our home directory, we can also create a custom nltk data directory and verify it is in the list of known paths defined by nltk.data.path.

Code Snippet:

```
In [1]: import os, os.path

        path = os.path.expanduser('~/nltk_data')

        if not os.path.exists(path):
            os.mkdir(path)

        print ("Does path exists : ", os.path.exists(path))

        import nltk.data
        print ("\nDoes path exists in nltk : ",
               path in nltk.data.path)

        Does path exists :  True

        Does path exists in nltk :  True

In [2]: text = nltk.data.load('corpora/nat_geo/ScrapedArticles1.txt', format='raw')

In [ ]: text = text.decode("utf-8")

In [22]: text[0:50]

Out[22]: '\ufeffA Culture Ride Through Chiang Mai and Chiang Rai\r'
```

2. Tokenizing

Here we have performed sentence tokenizing in-order to count average words per sentence. The following bit of code will add tokenized sentences into the array for stemming.

Code Snippet:

```
In [6]: from nltk.tokenize import word_tokenize, sent_tokenize

        file_docs = []

        tokens = sent_tokenize(text)
        for line in tokens:
            file_docs.append(line)

In [33]: print(file_docs[0:1])

         ['\ufeffA Culture Ride Through Chiang Mai and Chiang Rai\r\nLeaning out from a bridge leading into Wat Rong Khun, I squint at a
         stucco moat of outstretched hands and grisly skeletons.']
```

3. Stemming

Here we have used Lancaster Stemmer in-order to stem the data .

Code Snippet:

```
In [ ]: from nltk.stem import LancasterStemmer
        stemmerLan = LancasterStemmer()
        gen_docs = [[stemmerLan.stem(w.lower()) for w in word_tokenize(t)]
                    for t in file_docs]

In [37]: print(gen_docs[0][0])

         a
```

4. Dictionary

In order to work on text documents, Gensim requires the words (aka tokens) be converted to unique ids. So, Gensim lets you create a Dictionary object that maps each word to a unique id. Here we haveconverted our sentences to a [list of words] and then passed it to the corpora.Dictionary() object.

Code Snippet:

```
In [39]: import gensim

         dictionary = gensim.corpora.Dictionary(gen_docs)
         print(dictionary.token2id)
```

```
{',': 0, '.': 1, 'a': 2, 'and': 3, 'at': 4, 'bridge': 5, 'chiang': 6, 'culture': 7, 'from': 8, 'grisly': 9, 'hands': 10, 'i':
11, 'into': 12, 'khun': 13, 'leading': 14, 'leaning': 15, 'mai': 16, 'moat': 17, 'of': 18, 'out': 19, 'outstretched': 20, 'ra
i': 21, 'ride': 22, 'rong': 23, 'skeletons': 24, 'squint': 25, 'stucco': 26, 'through': 27, 'wat': 28, '\ufeffa': 29, 'ahea
d': 30, 'arch': 31, 'bouncers': 32, 'burly': 33, 'death': 34, 'entrance': 35, 'entry': 36, 'few': 37, 'giant': 38, 'guard': 3
9, 'heaven': 40, 'horns': 41, 'like': 42, 'massive': 43, 'my': 44, 'over': 45, 'poised': 46, 'rahu': 47, 'restrict': 48, 'sta
tues': 49, 'steps': 50, 'the': 51, 'to': 52, 'two': 53, 'walkway': 54, 'while': 55, 'couple': 56, 'foreign': 57, 'overhear':
58, 'rebirth': 59, 'talking': 60, '“': 61, '”': 62, 'arms–lie': 63, 'blocking': 64, 'desire–enslaved': 65, 'in': 66, 'morta
l': 67, 'nirvana': 68, 'road': 69, 's': 70, 'swampland': 71, 'wait': 72, '’': 73, 'an': 74, 'as': 75, 'blinking': 76, 'circl
e': 77, 'could': 78, 'disney': 79, 'dream': 80, 'elton': 81, 'every': 82, 'for': 83, 'fragmented': 84, 'gonzo': 85, 'inch': 8
6, 'infernal': 87, 'john': 88, 'lakeside': 89, 'life.': 90, 'mirrors': 91, 'monument': 92, 'not': 93, 'pg-13': 94, 'spectre':
```

5. Term Frequency – Inverse Document Frequency(TF-IDF)

Term Frequency – Inverse Document Frequency(TF-IDF) is also a bag-of-words model but unlike the regular corpus, TFIDF down weights tokens (words) that appears frequently across document.
Here we have used TfidfModel() function in gensim library.

Code Snippet:

```
In [10]: import numpy as np

         tf_idf = gensim.models.TfidfModel(corpus)
         for doc in tf_idf[corpus]:
             print([[dictionary[id], np.around(freq, decimals=2)] for id, freq in doc])
```

```
[[',', 0.01], ['.', 0.0], ['a', 0.06], ['and', 0.07], ['at', 0.08], ['bridge', 0.19], ['chiang', 0.32], ['culture', 0.15],
['from', 0.07], ['grisly', 0.24], ['hands', 0.24], ['i', 0.06], ['into', 0.11], ['khun', 0.2], ['leading', 0.22], ['leaning',
0.22], ['mai', 0.17], ['moat', 0.24], ['of', 0.03], ['out', 0.11], ['outstretched', 0.24], ['rai', 0.19], ['ride', 0.17], ['r
ong', 0.2], ['skeletons', 0.24], ['squint', 0.24], ['stucco', 0.24], ['through', 0.12], ['wat', 0.2], ['\ufeffa', 0.24]]
[[',', 0.04], ['.', 0.0], ['a', 0.03], ['and', 0.04], ['into', 0.11], ['of', 0.03], ['ahead', 0.21], ['arch', 0.21], ['bounce
rs', 0.24], ['burly', 0.24], ['death', 0.21], ['entrance', 0.19], ['entry', 0.19], ['few', 0.14], ['giant', 0.24], ['guard',
0.21], ['heaven', 0.24], ['horns', 0.24], ['like', 0.1], ['massive', 0.24], ['my', 0.07], ['over', 0.13], ['poised', 0.24],
['rahu', 0.21], ['restrict', 0.24], ['statues', 0.19], ['steps', 0.2], ['the', 0.04], ['to', 0.03], ['two', 0.13], ['walkwa
y', 0.24], ['while', 0.14]]
[[',', 0.02], ['.', 0.0], ['a', 0.05], ['bridge', 0.35], ['i', 0.1], ['of', 0.06], ['the', 0.03], ['couple', 0.39], ['foreig
n', 0.37], ['overhear', 0.44], ['rebirth', 0.39], ['talking', 0.44], ['“', 0.13], ['”', 0.13]]
[['.', 0.0], ['a', 0.09], ['of', 0.05], ['to', 0.05], ['arms–lie', 0.37], ['blocking', 0.37], ['desire–enslaved', 0.37], ['i
```

6.Document Similarity

- **Creating similarity measure object:** Now, we are going to create similarity object. The main class is Similarity, which builds an index for a given set of documents. The Similarity class splits the index into several smaller sub-indexes, which are disk-based.

Code Snippet:

```
In [11]: sims = gensim.similarities.Similarity('~/',tf_idf[corpus],
                                                num_features=len(dictionary))
```

- **Create Query Document:** Now we are going to calculate how similar is this query document to our our original document(article). Here we have used an article named “How I Returned to India Before the Lockdown” by SHRENIK AVLANI. So, we have created second text file which will include query documents or sentences and tokenize them as we did before.

Code Snippet:

```
In [12]: dtext = nltk.data.load('corpora/your_corpus/dtext.txt', format='raw')
         dtext = dtext.decode("utf-8")

In [13]: file2_docs = []


         tokens = sent_tokenize(dtext)
         for line in tokens:
             file2_docs.append(line)


         for line in file2_docs:
             query_doc = [w.lower() for w in word_tokenize(line)]
             query_doc_bow = dictionary.doc2bow(query_doc)

In [14]: query_doc_tf_idf = tf_idf[query_doc_bow]
```

- **Document similarities to query:** Now we are going to see the similarity between the out article and the query document sentence by sentence using ccosinr similarity.

Code Snippet:

```
In [17]: sum_of_sims =(np.sum(sims[query_doc_tf_idf], dtype=np.float32))
         print(sum_of_sims)

         34.083164

In [18]: percentage_of_similarity = round(float((sum_of_sims / len(file_docs)) * 100))
         print(f'Average similarity percentage: {float(sum_of_sims / len(file_docs)) * 100}')
         print(f'Average similarity rounded percentage: {percentage_of_similarity}')

         Average similarity percentage: 3.197294954511059
         Average similarity rounded percentage: 3
```

- We can conclude that average similarity is approximately 3%.

7. Readability

Below is the program through flesch index to determine the readability of our text file.

```
In [61]: sentence = text.count('.') + text.count('!') + text.count(';') + text.count(':') + text.count('?')
         words = len(text.split())
         syllable = 0
         for word in text.split():
             for vowel in ['a','e','i','o','u']:
                 syllable += word.count(vowel)
             for ending in ['es','ed','e']:
                 if word.endswith(ending):
                     syllable -= 1
             if word.endswith('le'):
                 syllable += 1
         G = round((0.39*words)/sentence+ (11.8*syllable)/words-15.59)
         if G >= 0 and G <= 30:
              print ('The Readability level is high')
         elif G >= 50 and G <= 60:
              print ('The Readability level is mediuml')
         elif G >= 90 and G <= 100:
              print ('The Readability level is low')
         print ('This text has %d words' %(words))

         The Readability level is high
         This text has 20108 words
```

Contributions

Reading: Harshith Chukka and Sanjoi Sethi
Tokenizing: Shubham Varade and Sarthak Banerjee
Stemming: Jeswin Jacob J and Samartha Pal
Dictionary: Harshith Chukka and Guneshwar Singh
TFIDF: Harshith Chukka and Abhishek Dhall
Document Similarity: Harshith Chukka and Rakshit Kumar
Finding Readability Index: Harshith Chukka and Keerthivasan
Documentation: Sanjoi Sethi and Harshith Chukka

Team 5-Slot(E2+TE2)

RAKSHIT KUMAR
18BCE0999

HARSHITH CHUKKA
18BCE0941

SARTHAK BANERJEE
18BCE2341

SAMARTHA PAL
18BCE2434

ABHISHEK DHALL
18BCE2278

KEERTHIVASAN
18BCE2133

GUNESHWAR SINGH MANHAS
18BCE0960

JESWIN JACOB J
18BCE2121

SHUBHAM VARADE
18BCE2388

SANJOI SETHI
18BCE2261

THANK YOU!