OLD DOMINION UNIVERSITY

Assignment 3

David Bayard

March 3, 2019

QUESTION 1.

Download the 1000 URIs from assignment 2. "curl", "wget", or "lynx" are all good candidate programs to use. We want just the raw HTML, not the images, stylesheets, etc.

Solution:

The requests library was used to download the URIs, sending GET requests to each of the URIs and creating a response object from the response. This was repeated for each URI in the list of existing URIs from assignment 2.

```
  try:
     response = requests.get(uri, allow_redirects=True, timeout=30)

   except Exception as e:
     print('Error: ', str(e))
```

Listing 1: Download HTML

After downloading the HTML, a hashing function was used to create distinct file names for each URI. The returned hash code and URI were then appended to the “Records.txt” file. After creating each file, the .text() method of the response object is used to retrieve the HTML and write it to the new file.

```
  with open('tempFile.txt', 'rb') as afile:
       buf = afile.read()
       hasher.update(buf)

   fileName = hasher.hexdigest() + '.html'

   #Write to the specified file (with hash code)
   writeComplete = open(fileName, 'w')

   writeComplete.write(response.text)
```

Listing 2: Make Hashcode

“python-boilerpipe” was used to extract the HTML from each file. In order to use this library, a virtual environment was created, and the source code was installed in this environment. There were difficulties when installing “python-boilerpipe”, but this was solved by specifying the Python version to use (3.6) when installing.

```
with open(a['FileName'] , 'r') as read:
    html = read.read()

 #Extract html from file
 extractor = Extractor(extractor='ArticleExtractor', html=html)
 extractedText = extractor.getText()
```

Listing 3: Extract Text

The code above is responsible for extracting the majority of the HTML from each hashed document, resulting in new files that contain only the text from the HTML files.

QUESTION 2

Choose a query term (e.g., "shadow") that is not a stop word (see week 5 slides) and not HTML markup from step 1 (e.g., "http") that matches at least 10 documents (hint: use "grep" on the processed files). If the term is present in more than 10 documents, choose any 10 from your list.

Solution:

The query term used was “Congress”. This query term is relevant within the context of the downloaded documents, and it was expected to have been found in multiple documents. With the query term selected, a search was conducted on each of the document files containing extracted text, using grep as the searching tool.

```
process = subprocess.Popen(['grep', '–rwn', arg, completeName], stdout=subprocess.PIPE)
    stdout, stderr = process.communicate()
    if len(stdout) != 0:
      num_words = 0
```

Listing 4: Query Term Search

In the above code, there is a list “stdout”, which contains all of the information retrieved from grep search. Moreover, the if condition filters out any documents that did not match the query term.

Two very important segments of code are listed below. This code is responsible for counting the number of words that each document contains, as well as the number of times the query term appears in the document. This information is then stored in a file for later use.

```
with open(completeName, 'r') as readFile:
        for line in readFile:
          words = line.split()

          ## Keeps track of number of words in file ##
          num_words += len(words)

      with open(completeName) as toCount:

        ## Code from Joshua on slack, Counts occurrance of query term in document##
        wordcount = Counter(toCount.read().split())
        wordOuccurance = wordcount['Congress']

        if(wordOuccurance != 0):
          fileCounter += 1
```

Listing 5: Word Count and Count for Query Term

The following equations listed below were used to calculate the TFIDF, TF, and IDF:

$$TF = \frac{Occurrances\ of\ Query\ Term}{Total\ Word\ Count}$$

$$IDF = \frac{Total\ Number\ of\ Documents}{Documents\ Containing\ Querty\ Term}$$

$$TFIDF = TF * IDF$$

With regards to normalization for question 3, the following equation was used:

$$Normalized\ Score = \frac{x - x_{min}}{x_{max} - x_{min}}$$

All values were rounded to four significant figures (after the decimal) because most of the values for the TF, IDF, and TFIDF were smaller than those of the assignment example.

Moreover, a logarithmic function was not required for the IDF because the total number of documents was used as the corpus, instead of using a search engine. All of the results for the TF, IDF, and TFIDF calculations are stored in the “CalculationRecords2.txt” file for future reference.

In order to calculate the TF, IDF, and TFIDF, the snippet of code supplied below was used. This code takes the data from the “MatchedFormatted.json” file and runs it through the formulas supplied above. In this case, the corpus is 1130, and the query term was found in 375 files, found in "fileOccurrance.txt". The TF, IDF, and TFIDF, are then rounded to four significant figures, and are written to a dictionary for later use.

```
if(checkURI in collectionURI):
        ## Word Count (Frequency) / Number of words found in document
        TF = entry["WordCount"] / entry["NumberWords"]
        roundedTF = round(TF,4)

        ## Had a total of 1130 documents, Query Term was found in 375 documents out of 1130
        IDF = 1130/375
        roundedIDF = round(IDF,4)

        TF_IDF = TF * IDF
        roundedTF_IDF = round(TF_IDF, 4)

        calcData["TFIDF"] = roundedTF_IDF
        calcData["TF"] = roundedTF
        calcData["IDF"] = roundedIDF
        calcData["URI"] = entry["URI"]

        collectionCalc.append(calcData)
```

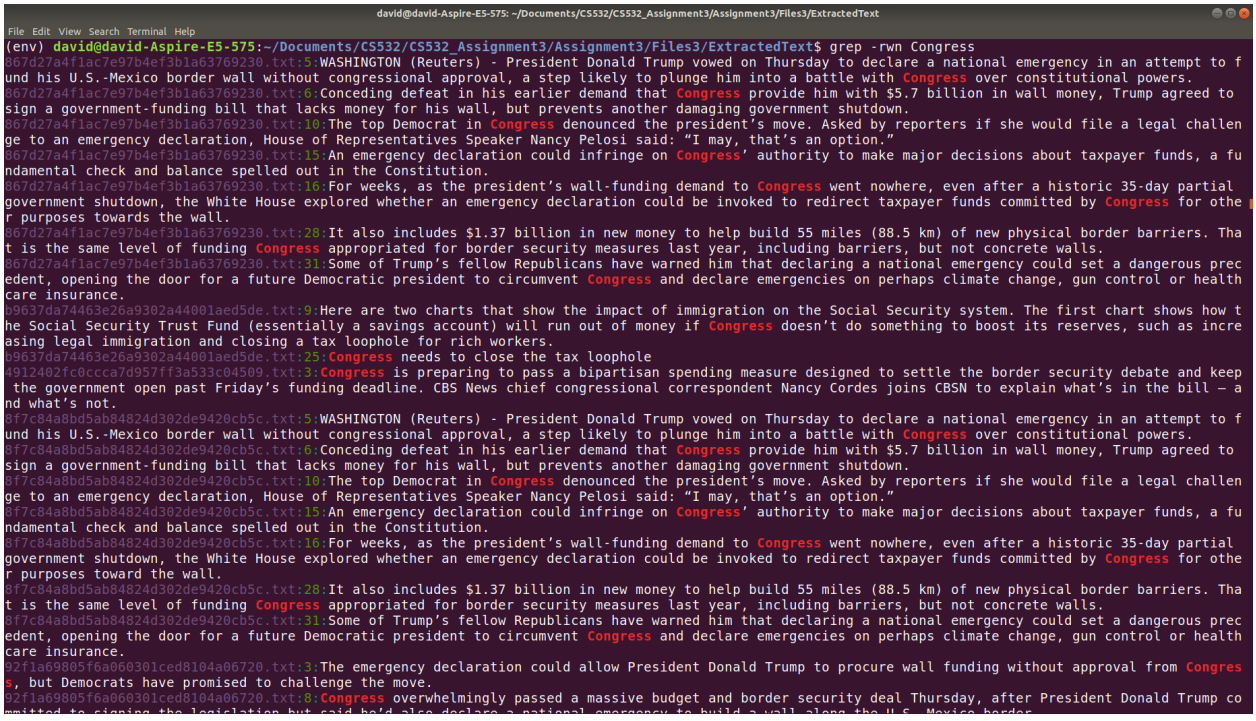
Listing 6: Calculations

The URIs in the table below were shortened to fit within the coulumn of the table. The full URI and data is provided in the “CalculationRecords2.txt” file.

Calculate TF, IDF, and TFIDF

Calculation Records Using Query Term “Congress”			
TFIDF	TF	IDF	URI
0.0339	0.0113	3.0133	https://www.numbersusa.com/
0.0324	0.0108	3.0133	https://www.facebook.com/
0.0261	0.0086	3.0133	https://in.reuters.com/
0.0222	0.0074	3.0133	https://www.reuters.com/
0.0209	0.0069	3.0133	http://www.msn.com/
0.0160	0.0053	3.0133	https://www.cnn.com
0.0155	0.0051	3.0133	https://dailywn.com/
0.0143	0.0047	3.0133	https://www.upi.com/
0.0135	0.0045	3.0133	https://www.vox.com/
0.0126	0.0042	3.0133	https://whyy.org/
0.0105	0.0035	3.0133	https://www.cnbc.com/
0.0099	0.0033	3.0133	https://www.theguardian.com/
0.0088	0.0029	3.0133	https://www.nbcnews.com/
0.0074	0.0025	3.0133	https://www.nytimes.com/
0.0071	0.0024	3.0133	https://newspunch.com/
0.0060	0.0020	3.0133	https://apnews.com/
0.0034	0.0011	3.0133	https://fullmagazine.us/

As an example, the picture below provides an graphical view of what is occurring behind the scenes. This is only a segment of the program though, displaying where each occasion of the Query Term is located, and how many times it occurs.



(a) grep behind the scene

QUESTION 3

Now rank the same 10 URIs from question 2 but this time by their PageRank. Use any of the free PR estimaters on the web.

The PR estimater used for this assignment was `http://www.prchecker.info/check_page_rank.php`. There were a few issues that occurred while using the PR estimator, which resulted in less accurate results.

One issue associated with the PR estimator is that it only ranks a page based off of the home page of each URI. This resulted in a less specific PR, as it was not based off of the correct page. Another issue was that several of the chosen URIs had a page rank of zero. These URIs were mentioned in the "NoRanks.txt" file, and were not added to the list of normalized ranks in the "PageRanks.txt" file.

There was no real code involved with retreiving the page ranks, because of the anti-bot captchas. Instead, the URIs were manually run through the PR estimator. A table of each of the page ranks is listed below, displaying the pages that did not have a rank of zero.

Page Ranks for 10 URI

Page Ranks	
Page Rank	URI
1.0	https://www.cnn.com/
1.0	https://www.facebook.com/
0.9	https://www.reuters.com/
0.9	https://in.reuters.com/
0.9	http://www.msn.com/
0.8	https://www.theguardian.com/
0.8	https://www.upi.com/
0.7	https://www.vox.com/
0.7	https://whyy.org/
0.7	https://www.numbersusa.com/

The Page Rank results seem to be inaccurate, and do not produce a correlation between the page rank and the TF, IDF, and TFIDF scores. For example, https://www.numbersusa.com/ ranks lowest on the Page Rank table, but scores highest on the TFIDF value.

Moreover, https://www.facebook.com/ ranks second on both table, contradicting any possible correlation from the previous statement. This hints towards the idea that the page rank website is producing false results, and not ranking pages correctly. This is even more evident by the fact that https://www.nbcnews.com/ did not have a page rank.

QUESTION 4. EXTRA CREDIT

Compute the Kendall TauB "b" score for both lists. Report both the Tau value and the "p" value.

Solution:

Python scipy provides a function called scipy.stats.kendalltau for this exact purpose. This function was implemented, providing the list of page ranks against the list of TFIDFs, TFs, and IDFs, as shown below.

```
## Calculate Kendall Tau_b Score TFIDF ##
tauTFIDF, p_valueTFIDF = sp.stats.kendalltau(normalizedRanks, collectionTFIDF)
```

Listing 7: Word Count and Count for Query Term

Tau Calculations

Correlation		
Comparison	Tau	P Value
TFIDF VS Page Rank	0.25	0.19

As seen in the table above, the Tau b score indicates that there is a minor association between the page ranks and the TFIDF. This is evident by a Tau b score, of .25, indicating that there is an association, but that it is insignificant.