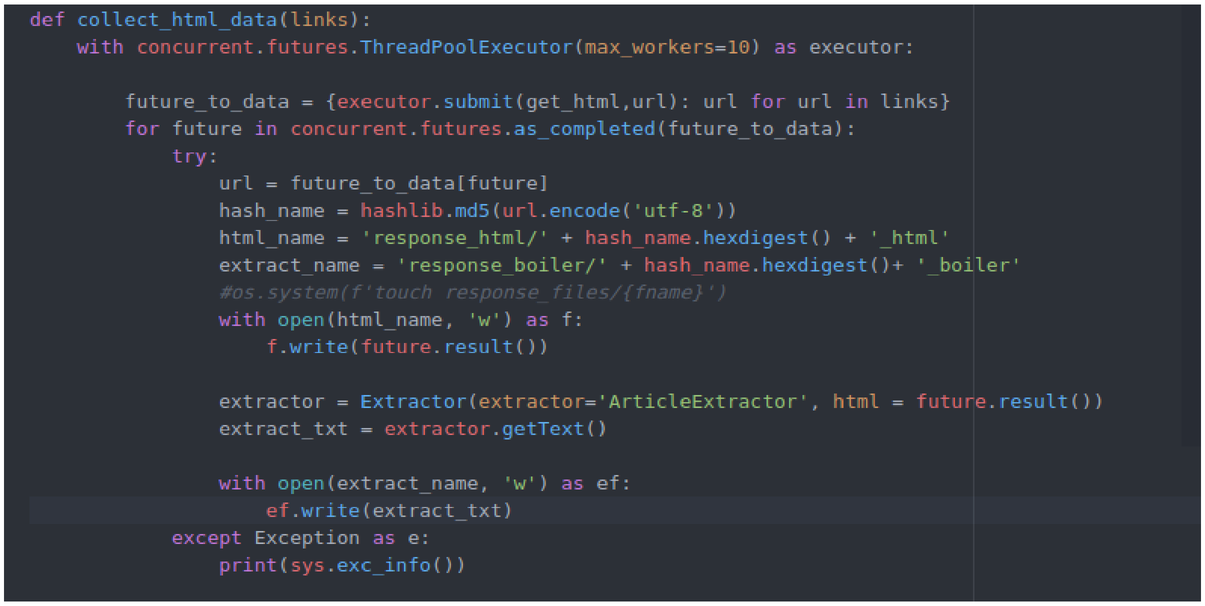
CS 432 Assignment #3

Larry Montgomery 01093132

Part 1: Download and Extract Text for Each URI

The process of downloading all the HTML for each URI collected in the previous assignment was a task that I used similar code from previous assignments. In particular using the requests library combined with the futures library to run 10 concurrent threads at a time. To collect the HTML I used the GET request method, which led me to some of the sites not allowing my unspecified user-agent header. I set the user-agent header to be equal to Mozilla and this allowed me to collect the responses. I found that 24 of the links were not valid anymore. I generated a MD5 hash for each URI and appended "_HTML" to generate unique file names. I applied this naming process to the files with extracted text, except I used the extension "_boiler".

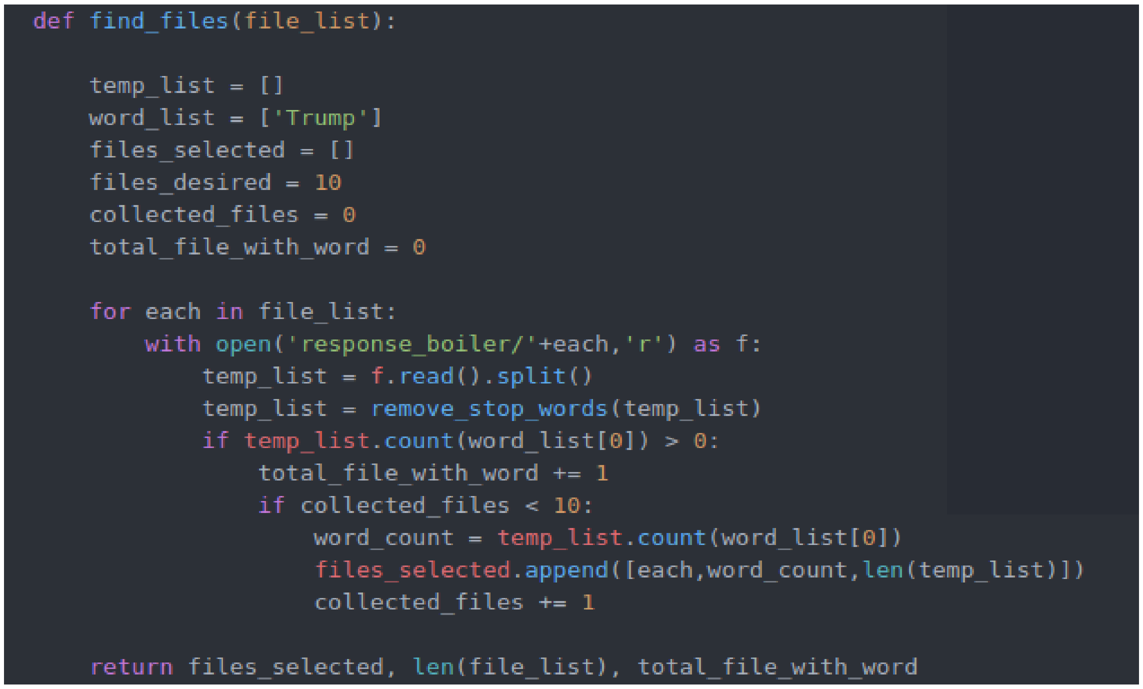
The extraction required a little more effort. I used the python wrapper library for the Java boilerpipe library to facilitate the extraction of the text from the HTML responses. I used GitHub to download the Python wrapper and then followed the instructions of installing dependencies and adjusting the JAVA-HOME environment variable. After the base setup the use of the library was quite simple. To write my own method would have taken far longer and would have missed some important conditions, given the time to complete the assignment. The sample code below is the main collection method combined with the extraction.



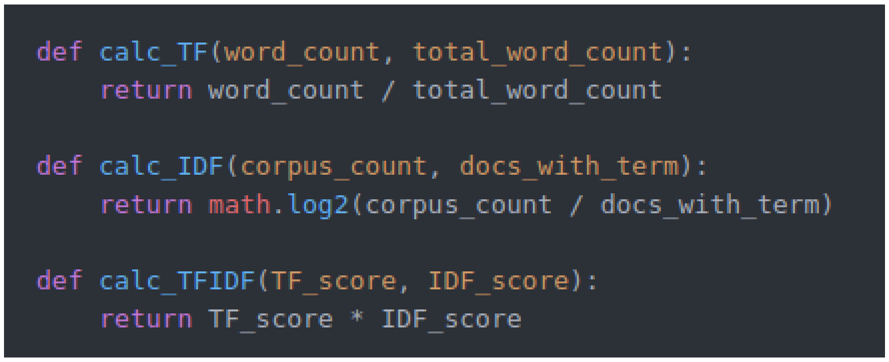
```python
def collect_html_data(links):
    with concurrent.futures.ThreadPoolExecutor(max_workers=10) as executor:

        future_to_data = {executor.submit(get_html,url): url for url in links}
        for future in concurrent.futures.as_completed(future_to_data):
            try:
                url = future_to_data[future]
                hash_name = hashlib.md5(url.encode('utf-8'))
                html_name = 'response_html/' + hash_name.hexdigest() + '_html'
                extract_name = 'response_boiler/' + hash_name.hexdigest()+ '_boiler'
                #os.system(f'touch response_files/{fname}')
                with open(html_name, 'w') as f:
                    f.write(future.result())

                extractor = Extractor(extractor='ArticleExtractor', html = future.result())
                extract_txt = extractor.getText()

                with open(extract_name, 'w') as ef:
                    ef.write(extract_txt)
            except Exception as e:
                print(sys.exc_info())
```

Part 2: Calculate TFIDF Score

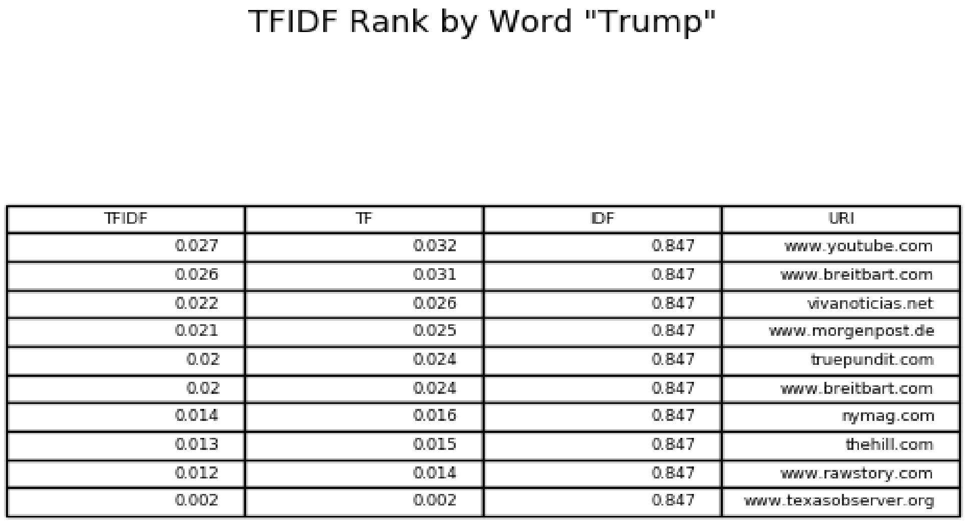
When calculating the term frequency(TF) and inverse document frequency(IDF) the first step I took was splitting the large text blocks from the extractor into tokens. I performed this by using the split method when reading the text from the file. This method combined some forms of punctuation together with other text, which I attempted to remedy at the same time I removed stop words from the list of word tokens. After the list had been reduced to the significant words. I counted the frequency of the search word, Trump in this case. I counted the number of words in each document when I found the search term. After finding 10 documents I ended the search and moved on to generating the TF and IDF scores for each document. I used the equations from the class slides as reference for writing the methods. After I calculated the scores for all the URIs I used matplotlib to generate a table to represent the ranking between the URI's TFIDF scores. The example of the code used to search the files for the search term is below. There also is the example that shows the base methods for calculating the TF, IDF and TFIDF.



```python
def find_files(file_list):

    temp_list = []
    word_list = ['Trump']
    files_selected = []
    files_desired = 10
    collected_files = 0
    total_file_with_word = 0

    for each in file_list:
        with open('response_boiler/'+each,'r') as f:
            temp_list = f.read().split()
            temp_list = remove_stop_words(temp_list)
            if temp_list.count(word_list[0]) > 0:
                total_file_with_word += 1
                if collected_files < 10:
                    word_count = temp_list.count(word_list[0])
                    files_selected.append([each,word_count,len(temp_list)])
                    collected_files += 1

    return files_selected, len(file_list), total_file_with_word
```



```python
def calc_TF(word_count, total_word_count):
    return word_count / total_word_count

def calc_IDF(corpus_count, docs_with_term):
    return math.log2(corpus_count / docs_with_term)

def calc_TFIDF(TF_score, IDF_score):
    return TF_score * IDF_score
```


TFIDF Rank by Word "Trump"

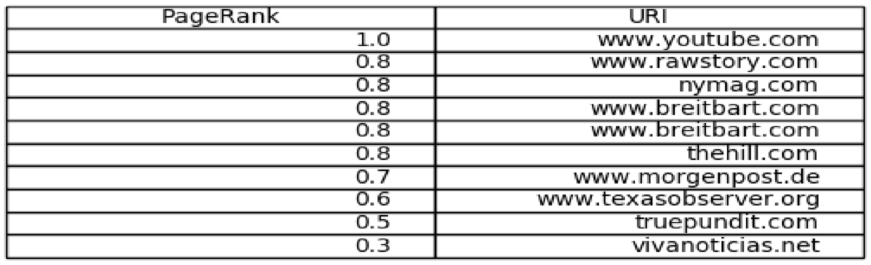
TFIDF	TF	IDF	URI
0.027	0.032	0.847	www.youtube.com
0.026	0.031	0.847	www.breitbart.com
0.022	0.026	0.847	vivanoticias.net
0.021	0.025	0.847	www.morgenpost.de
0.02	0.024	0.847	truepundit.com
0.02	0.024	0.847	www.breitbart.com
0.014	0.016	0.847	nymag.com
0.013	0.015	0.847	thehill.com
0.012	0.014	0.847	www.rawstory.com
0.002	0.002	0.847	www.texasobserver.org

Part 3: Calculate Page Rank Score

For calculating the page ranks, I used the website www.checkpagerank.net. I tried using one or two other sites but they didn’t have part of the domains from my sample set. I have to say that the the ranks seem quite unreliable since most of my results returned an 8 out of 10. After calculating the normalized page rank which I did by dividing each score by 10, I sorted a table that I made with matplotlib based on the page rank. The table can be seen below.

After completing both tables I noticed that the raw page rank from these sites was not very good at predicting the TFIDF score and should not be used when looking for specific content. I take away from this that page rank can be associated with the overall popularity of a domain, but the TFIDF is the better predictor of overall content quality. YouTube was the only URI that did well in both tables and could be seen as the best overall choice when trying to deliver the best resource in response to the query. There are only base level domains because adding the full paths to the tables would have made the tables unreadable when using matplotlib’s table generator.

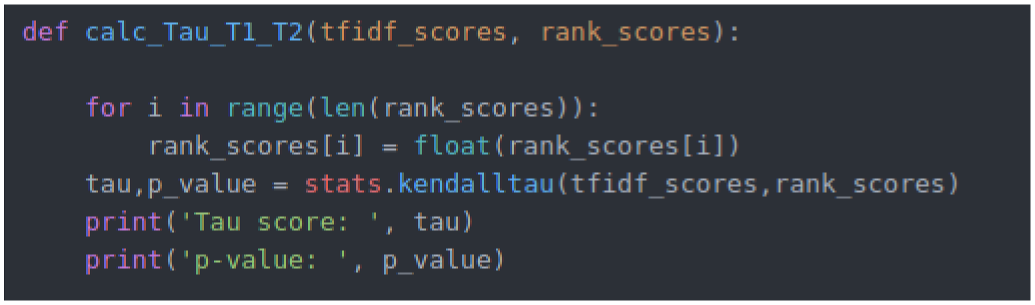
Page Rank for Domains



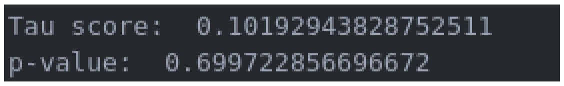
PageRank	URI
1.0	www.youtube.com
0.8	www.rawstory.com
0.8	nymag.com
0.8	www.breitbart.com
0.8	www.breitbart.com
0.8	thehill.com
0.7	www.morgenpost.de
0.6	www.texasobserver.org
0.5	truepundit.com
0.3	vivanoticias.net

Extra Credit: Calculate Kendall Tau-b Score

To calculate the Kendall Tau-b score I used the scipy.stats method. This was suggested by David Bayard from the slack channel. I used the TFIDF scores from table 1 and the rank scored from table 2 to calculate the correlation between the two ranking methods. From the result you seen bellow There is no significant correlation in either way. This leads me to say there is no correlation between the two ranking methods.



```python
def calc_Tau_T1_T2(tfidf_scores, rank_scores):

    for i in range(len(rank_scores)):
        rank_scores[i] = float(rank_scores[i])
    tau,p_value = stats.kendalltau(tfidf_scores,rank_scores)
    print('Tau score: ', tau)
    print('p-value: ', p_value)
```



```
Tau score:  0.10192943828752511
p-value:  0.699722856696672
```