Daniel Kantor

Final Project Report

Libraries Used

requests – to get crawl and get the HTML content of a website

re – to use regular expressions

os – to iterate through all the txt files we save for the program

json – to save a dictionary as a json and to open a json and save it as a dictionary

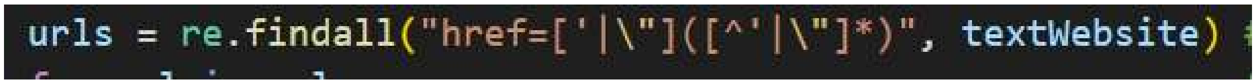
BeautifulSoup – to grab the text elements from a HTML file

nltk – to lemmatize words and to download the set of stopwords

math – to find the log of numbers and the square root of numbers

Step 1: Web Crawler

The web crawler was used to go through all the webpages on the muhlenberg.edu website and extract all the links located on the page by using a regular expression to parse the HTML code on the page. First the URL is popped off the stack. Then it is important to check that the URL that the program was trying to access worked, which is why the loop only runs on a given webpage if it returns the 200-status code.

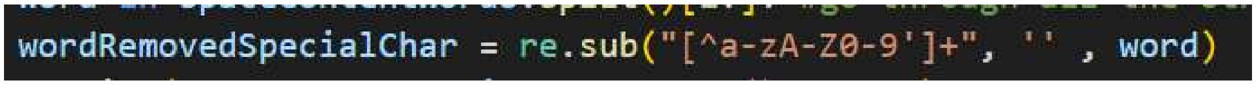


```
urls = re.findall("href=['|\"]([^'|\"]*)", textWebsite)
```

This simple regular expression is then used to get all the links in href tags (clickable links) that are located on the page. To explain the regex simply, it looks for a href= tag and then either a ‘ or “ to denote the beginning of the URL. It will then keep gathering all characters following the quotes until it finds the closing quote. After gathering all the URLs on a given page I iterate through them and check if it is either a relative or absolute URL. We can check if it’s a relative link by checking if the first character in the string is a /. If it is a relative URL I then append the www.muhlenberg.edu URL to it. There is then a filter to check if the URL is any webpage is not a HTML webpage as we don’t want to crawl those. The crawler then checks if the URL has already been seen which is kept track of in a seen list and if it has yet to be seen the URL is appended to the stack. After all the URLs on a given page have been iterated through, the HTML of the page we were looking at is then saved to a text file to be cleaned and processed later.

Step 2: Inverted Index

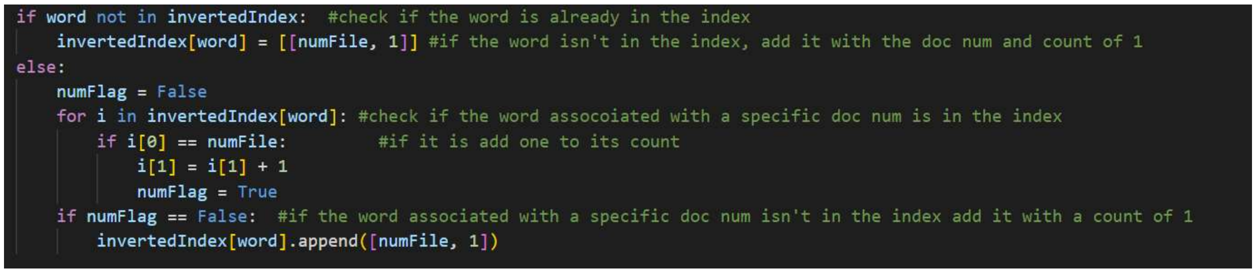
Before creating the inverted index it is important to clean the files of the HTML code and other characters that may not be useful for the program. To remove all HTML, CSS, and other non-text characters in the files BeautifulSoup is used. This leaves us with only the words that were in the raw files. For each word in the file, I remove all special characters such as periods, commas, colons, etc. To do this I used the regular expression seen below



```
wordRemovedSpecialChar = re.sub("[^a-zA-Z0-9']+", '' , word)
```

I did this because while these characters might sometimes have meaning, I felt it far more important to only be looking at the words themselves. I also saved each word as lowercase as a way to aggregate the counts of the words more easily. After this cleaning has been done, the cleaned words for each document are saved in a file that only comprises the cleaned words to be used to create the inverted index.

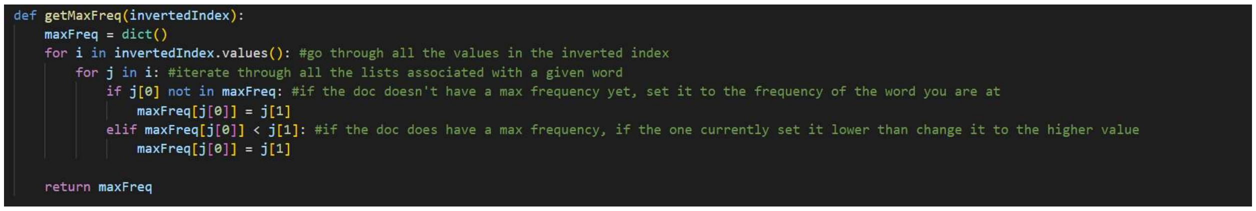
The inverted index is represented as a dictionary with words as the keys and a list of lists containing the document number and the word count for that document as values. First the file is opened and depending on which optimizations are selected it will enter a different if statement but the logic for adding words to the inverted index is the same regardless of which optimizations are selected as stop word removal and lemmatization occurs before the word is added to the dictionary.



```
if word not in invertedIndex:  #check if the word is already in the index
    invertedIndex[word] = [[numFile, 1]] #if the word isn't in the index, add it with the doc num and count of 1
else:
    numFlag = False
    for i in invertedIndex[word]: #check if the word assocoiated with a specific doc num is in the index
        if i[0] == numFile:         #if it is add one to its count
            i[1] = i[1] + 1
            numFlag = True
    if numFlag == False:  #if the word associated with a specific doc num isn't in the index add it with a count of 1
        invertedIndex[word].append([numFile, 1])
```

First the program checks if the word is in inverted index. If it is not then it is added a key and initialized with a list and inside this list is another list containing the document number the word was found and setting the count to one. If the word in the inverted dictionary the program then loops through all the lists that are values to the key. If the document number associated with the word is already in the list then the words count is increased by one, if the document associated with the word is not in the list then the document number is appended to the list with the words count being set to one. After this is done the inverted index is written to and saved as a json file to be used to calculate the rankings.

Additionally, there is the function used to get the max frequency word in each document that can be seen below



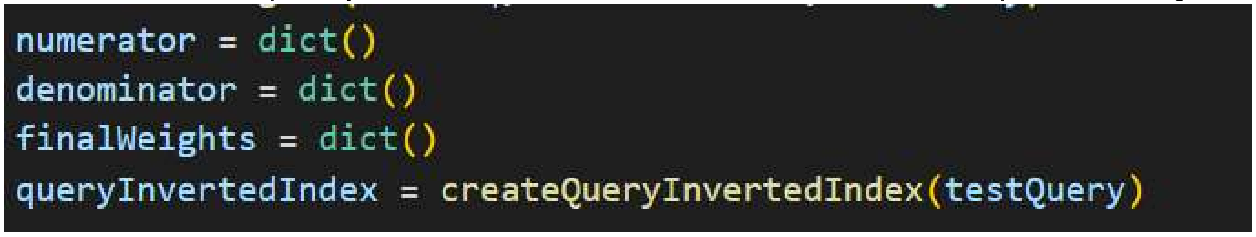
```python
def getMaxFreq(invertedIndex):
    maxFreq = dict()
    for i in invertedIndex.values(): #go through all the values in the inverted index
        for j in i: #iterate through all the lists associated with a given word
            if j[0] not in maxFreq: #if the doc doesn't have a max frequency yet, set it to the frequency of the word you are at
                maxFreq[j[0]] = j[1]
            elif maxFreq[j[0]] < j[1]: #if the doc does have a max frequency, if the one currently set it lower than change it to the higher value
                maxFreq[j[0]] = j[1]

    return maxFreq
```

Using a dictionary to store the values of max frequencies, it iterates through all the values in the inverted index. Checking each individual list that is associated with a given word, it checks if we have a max frequency associated for a given document, if there isn’t it sets the max frequency to the one that is currently being looked at in the dictionary. If the document already has a max frequency of and the one currently being looked at is higher then the max frequency is updated. This dictionary is then also saved as a json file to be used to calculate rankings.

Step 3: Retrieval

To calculate the rankings first the user is asked to enter a search query and choose which optimizations they want to be used. After this, the correct json files for inverted index and max frequency are loaded and stored as dictionaries to compute the weights.

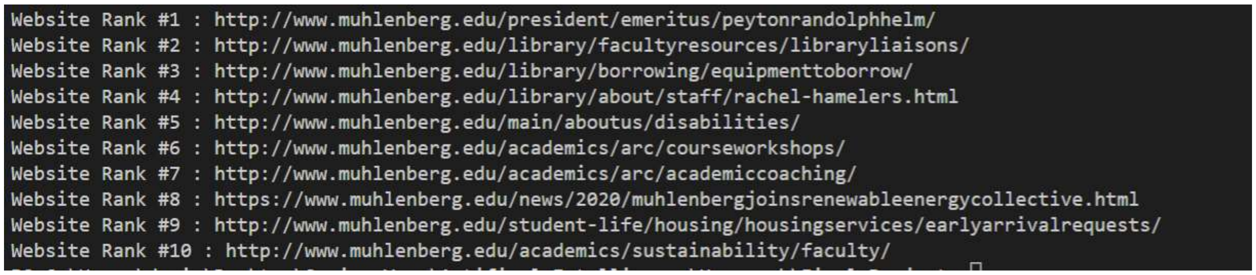


```python
numerator = dict()
denominator = dict()
finalWeights = dict()
queryInvertedIndex = createQueryInvertedIndex(testQuery)
```

This part of the program uses four dictionaries, a dictionary to represent the numerator, the denominator, the final weights of each page, and the inverted index that is used to represent the query which is created in a very similar process as the one described above. The program iterates through each word in the query. It first checks if the word in the query even exists in the inverted index. If it is then using the formulas taught in class, the values for the numerator and denominator for each document are calculated and stored in a dictionary. The denominator value for the query is also calculated at this time and stored in the denominator dictionary. This approach also uses the log value of the IDF to avoid it skewing the values too much. After all the words in the query have been iterated through, all the values in the numerator and denominator dictionaries are iterated through and the final relevance weights are stored in the finalWeights dictionary which is then returned so the most relevant websites can be found.

Step 4: Results

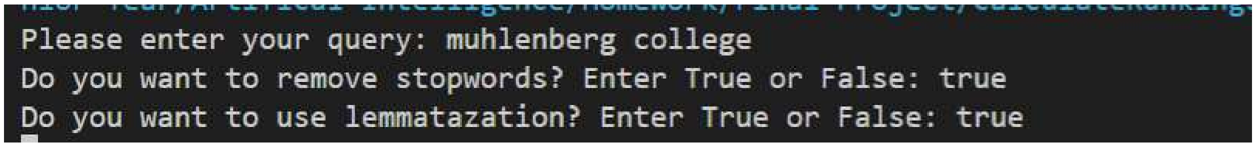
This method works by receiving that dictionary of weights that was calculated in the previous step. It also utilizes a text file listing the websites that were crawled in order so we can associate the weights with the website link. This text file is loaded and stored in a list. Then the dictionary of weights is sorted in reverse order so that the value that is at index zero is the highest value. Then the keys in the sorted dictionary of weights is iterated through and it gets the website link from the list based on the value that is stored in the dictionary. The rank of the website and the link is then printed to the terminal for the top ten websites, if there were less than ten websites crawled then the amount of weights that are stored in the dictionary are printed. The output looks like this:



```
Website Rank #1 : http://www.muhlenberg.edu/president/emeritus/peytonrandolphhelm/
Website Rank #2 : http://www.muhlenberg.edu/library/facultyresources/libraryliaisons/
Website Rank #3 : http://www.muhlenberg.edu/library/borrowing/equipmenttoborrow/
Website Rank #4 : http://www.muhlenberg.edu/library/about/staff/rachel-hamelers.html
Website Rank #5 : http://www.muhlenberg.edu/main/aboutus/disabilities/
Website Rank #6 : http://www.muhlenberg.edu/academics/arc/courseworkshops/
Website Rank #7 : http://www.muhlenberg.edu/academics/arc/academiccoaching/
Website Rank #8 : https://www.muhlenberg.edu/news/2020/muhlenbergjoinsrenewableenergycollective.html
Website Rank #9 : http://www.muhlenberg.edu/student-life/housing/housingservices/earlyarrivalrequests/
Website Rank #10 : http://www.muhlenberg.edu/academics/sustainability/faculty/
```

Step 6: GUI

My program runs in terminal and as such there is no formal GUI. It first asks for the user to enter their query then asks for a user to enter whether they want stopwords removed or not and then asks whether they want to lemmatize the words. It looks like this:



```
Please enter your query: muhlenberg college
Do you want to remove stopwords? Enter True or False: true
Do you want to use lemmatazation? Enter True or False: true
```

Difficulties

I want to note that for some reason the crawler was only able to find ~6700 websites before it ran out of websites to check and wasn't able to get to 10,000. I'm not sure why this was happening and could not solve it. Perhaps that is all the websites that existed after filtering out the non-HTML ones?

Also, while writing this report a discovered a small error in my code. When creating the inverted index for the query I make all the words lowercase, except when I am iterating through all the words to compute the weights I am trying to access the query inverted index at a given word except I forgot to convert that word to lowercase. So, if the search term includes any uppercase letters the program will crash but this is simply fixed by putting word.lower() when iterating through the words in the query for the calculateWeights() method. This will also happen with any special characters with the query so the regular expression I used to filter those out would also have to be added.