Project 4: Report

Christopher Di Giacomo

40133600

05 December 2023

COMP479 2232 D

Dr. Sabine Bergler

Table of Contents

Extraction

The following section covers all details pertaining to the process of extracting textual data from the crawled/scraped pages. It discusses details specific to crawling, as well as scraping, including the different tools used, design choices taken, and challenges encountered. Note that all of the following is implemented in the *extraction.py* file.

Crawling

For the purpose of crawling, the *Scrapy* tool was used. As per the assignment instructions, it is one of the scraping/crawling tools in the list linked in the project guidelines. This framework is very high-level, especially relative to the proposed *Spidy* web crawling tool that is a crawler used through a CLI. Thus, *Scrapy* very easy to integrate into the project environment, and begin using after going through the tutorials provided in its documentation. It is important to note that despite *Scrapy* 's ability to scrape websites, it was only used for the purpose of crawling in this project, as per the guidelines.

After the necessary imports, the spider is constructed by building a class called *ConcordiaCrawler* that inherits from the *scrapy.Spider* class. Within it, we define the multiple attributes such as a name attribute. However, more importantly, an attribute that contains a list consisting of all of the domains that the crawler is allowed to explore. This restricts the crawler from leaving the concordia domain to explore external domains through links, such as social media links (twitter, facebook, etc). The crawler also contains an attribute that assigns it its starting URL, which is simply the root URL of the crawling process. Since the project guidelines requires an inputted threshold on the number of downloaded pages, the download count is what is used by the crawler to increment and keep track of the number of pages that have been received via response to the asynchronous requests.

There are also some customized configuration settings for the crawler. These custom settings ensure that the crawler:

1- Obeys the *robots.txt* ethical crawling requirements.
2- Allows for a download delay of 1 second in between asynchronous requests, to avoid overloading the server. It is also an effort made to avoid getting flagged as a suspicious client.
3- Identifies myself as a Concordia student, by assigning my school email as the user-agent to each request that is sent out.

These configurations are all for the purpose of "ethical crawling" by respecting the *robots.txt* standards, and to not get my IP address blocked for suspiciously spamming the server.

Scraping

As in previous projects, the markup parsing/web scraping tool of choice is *BeautifulSoup* as per the project guidelines. For scraping, we simply extract the title and the body of the valid linked pages, and add them to a dictionary where they are mapped to by their URL as a key. The choice of including the title comes from the knowledge of *zoning* gathered in the course. Titles can often be indicative of relevance due to containing pertinent information, so I have decided to

combine them with the *<main>* tag within the body. Only considering the *<main>* tag allows for a lot of "garbage" to be filtered out of the scraping process, since the *<body>* is overdiluted with many irrelevant things, even including the entire header and footer. Furthermore, the *<main>* tag is processed a step further, by removing removing all contents within *<script> or <style>* tags, which can also tend to dilute the resulting terms with meaningless data. One more thing that is scraped for is the *lang* attribute of the *html* tag, for URL filtering. The function validates a link only if the scraped URL's html tag contains a set *lang* attribute, and if so, if it is set to ***en*** indicating that the page is in English.

Unit

As a unit, the extraction module works by recursively crawling through all of the links discovered by starting from the Concordia home page. All within the *parse()* function, these links are checked against a set of conditions before being crawled, as follows:

1- Firstly, the crawler verifies that they do not violate the *robots.txt* standards since the flag is set in the crawler's configuration settings.
2- Next, the function verifies that the links are not *mailto* or *tel* links for emailing or telephoning.
3- There is also a small helper function that the *parse()* method uses, called *is_valid_url()* which validates the URL by making sure it does it begins with *https or http,* and does not lead to a resource with the following extensions: pdf, jpg, jpeg, png, gif, mp4, zip.

If any of the above are violated then the link is not crawled. However if not, then an asynchronous request is sent to the server to retrieve the page from this URL. Once the page is retrieved, there are further verifications performed at the top of the *parse()* function:

1- First the function checks if the threshold for the batch size has been reached (download limit).
2- It then verifies if the server's response status code was indeed a *200 OK.*
3- Following this, it makes sure that the page is not an html page and not a pdf or mp4.
4- It then makes sure that this link has not already been visited, and scraped before.
5- If not, then it is consequently scraped, and its contents logged in the global *url_text_dict* dictionary.

If any of the above conditions are violated, then the function returns and this page is disregarded. If not, then after scraping the page, it is further processed for links, and the cycle begins again at the link verification process.

Supplementary

Using the global *url_text_dict* dictionary this function writes every crawled/scraped page to file in a subdirectory called *collection/*. This was done to ensure that the rest of the modules could function independently and did not require to crawl the website with every run. The *pipeline()* function calls all of the above functions in logical order, beginning with crawling and ending with writing the crawled pages to file.

Clustering

The following section details all with regards to the process of clustering the gathered data from the above steps. All that pertains to this section is implemented within the *clustering.py* file.

Loading Documents

The first function in this module, *load_documents()*, does as it is named. It loads all documents (scraped web pages) saved to the */collection* subdirectory, into a local list variable *documents.* This function is particularly useful to the functioning of this module, as well as the sentiment module, as it allows for them to run based on the locally stored web pages (documents), and does not require them to re-crawl and re-scrape with every run.

Vectorizing the Documents

The function following the previous is the *vectorize()* function, which allows for the scraped documents to be converted into a tf-idf matrix by using the *TfIdfVectorizer* of the *SciKit-Learn* library. The vectorizer creates a sparse document-term matrix, where the values are tf-idf weights representing terms in each respective document in which they occur.

Tf-Idf Vectorizer Configurations

To further improve the vectorizer's resulting matrix, its settings have been configured in order to produce meaningful results, and extract meaningful features. It's ***max_df*** parameter has been set to 0.5 in order to ignore terms that appear in at least 50% of the documents. Furthermore, the ***min_df*** parameter has been set to 5, which only retains terms that appear in at least 5 documents. These two settings allow for filtering outliers or anomalies, which can skew results and affect the overall evaluation of the clusters, as well as their sentiment values. The ***stop_words*** parameter has also been set to *english,* in order to filter out all pre-established stopwords of the English language. Finally, the default vectorizer's regular expression was overwritten to be slightly more strict in its term criteria. The ***token_pattern*** attribute was set to **r'(?u)\b[A-Za-z][A-Za-z]+\b'**, which accepts terms that are at least two characters long, and contain no numbers.

KMeans Clustering

Following this, the module now has the necessary input to create clusters from the scraped data. This is achieved by using the imported *KMeans* algorithm from the *SciKit-Learn*'s *cluster* library. The algorithm is parameterized by the matrix created using the *vectorizer()* function from earlier, as well as the desired number of clusters (3 and 6 for this project), and finally the random state seed. This seed is set to an arbitrary number, 42 in this case, and ensures consistent and reproducible results. Had this not been set, then running the implementation code for the same batch size would yield different results everytime.

Top-N terms

The next function in place is the *topTerms()* function, which returns the top 20 most important terms for a cluster. It does so by retrieving the feature names of the terms closest to the centroid

of the cluster. The centroid is found by using the *cluster_centers_*attribute of the *KMeans* object that we created.

Unit

Finally, the entire module is ran in a pipeline function called *cluster_pipeline()*. This function returns the vectorized matrix, the *KMeans* object that is fit to this matrix, and the list of documents. When called in the following module, these values are retrieved in order to perform the sentiment analysis.

Sentiment Analysis

This section covers the module responsible for producing the sentiment scores of the clusters. It does so in multiple steps, all of which are elaborated on in the following subsections. Note that the implementation details of this module can be found in the *sentiment.py* file.

Document-level Sentiment

The first step that is taken in this module for calculating the cluster sentiment scores is establishing document level sentiment scores. This is done in a function named *document_sentiments(),* which takes as input the list of documents that are returned by the previous clustering module. With the help of creating an *Afinn()* object, from the imported *afinn* library, a sentiment score is calculated for each of the documents by using the *afinn.score()* function. Since the magnitude of the scores are highly dependent on document length, they are all length normalized by dividing each documents score by the logarithm of the document's length. The logarithm also has add-1 smoothing to mitigate the possibility of dividing by zero. These normalized document sentiment scores are stored in a list and returned.

Remapping Documents to Clusters

At the top of the function designed to create the cluster sentiment scores, another function is called named *map_docs_to_clusters()*. The purpose of this helper function is to re-map the disassociated document sentiment scores, to their respective documents in the each of the clusters. This function returns a dictionary with the key as a cluster, and the value as a list of tuples. The tuples consist of the docID (index of the document in the document list), and its respective sentiment score.

Deriving Cluster Sentiment Scores

The most important function of this module is the *derive_cluster_sentiments()* function, responsible for calculating sentiment scores for each of clusters based on the gathered data. It does so by computing a weighted average of the document sentiment scores for each cluster. The weights for each document is calculated based on its cosine similarity to the centroid of the cluster. This is done with the help *SciKit-Learn's cosine_similarity()* function, which gets parameterized by our tf-idf matrix and the centroids of our clusters. The function then goes on to create a dictionary, similar to the one returned by the *map_docs_to_clusters()* function. However instead of the tuples consisting of (*docID, sentimentScore)* they consist of (*documentWeight,*

sentimentScore), where *documentWeight* is the cosine similarity of this document to the centroid of respective cluster.

This weighted average allows for the documents that carry more meaning to the centroid of the cluster, to gain an advantage towards other documents that lie further from the centroid. Formulas such as a simple average, or Euclidean-Distance weighted average, are too sensitive to presence of "outliers", which can heavily skew results due to their strong influence in such formulas. Furthermore, the addition of the document-level sentiment scores being logarithmically document-length normalized, allows for further mitigation of skewed results and an increased likelihood of meaningful results.

Unit

As a unit, the module's functioning depends on the *sentiment_pipeline()* function, that calls the *cluster_pipeline(),* as well as the functions mentioned in this section, successively, in order to finally output the cluster sentiment values to terminal.

Results

The following sections represents the results of running the *sentiment_pipeline()*, and the analysis of the achieved results for 100 scraped web pages.

Cluster Naming

Below can be seen the top 20 terms for 3 clusters and 6 clusters respectively.

```
*** TOP TERMS PER CLUSTER FOR k=3 ***

CLUSTER 0: health mental emergency smoking thinking services hands good building
critical quitting emotions procedures water change stress use self healthy thoughts

CLUSTER 1: campus west parking bus map sgw directions loyola montreal quebec blvd
maisonneuve maps canada kiosk commuter wheelchair atm street sherbrooke

CLUSTER 2: research graduate program student arts new academic ai faculty learning
indigenous december engineering november art science online course sign school



*** TOP TERMS PER CLUSTER FOR k=6 ***

CLUSTER 0: health mental thinking smoking good critical thoughts change healthy
services quitting behaviour social step physical self building book goal factors

CLUSTER 1: ai research indigenous gallery art says sustainable arts community
montreal new centre photo people goals world institute sdgs climate development

CLUSTER 2: west campus parking bus directions map sgw loyola quebec montreal canada
commuter wheelchair kiosk atm rack shuttle legend buildings maps
```

```
CLUSTER 3: december sign november account february research april business october
january filters mba september new march june john molson july august

CLUSTER 4: emergency ca courses time online services ext student course campus
prevention register safety groups procedures hands working training data support

CLUSTER 5: graduate program academic science faculty el engineering experiential
school student computer arts studies funding association undergraduate department
learning awards apply
```

Naming for K=3:

- Cluster 0: “health”
- Cluster 1: “accessibility”
- Cluster 2: *uncertain*

Naming for K=6:

- Cluster 0: “mental health”
- Cluster 1: “news”
- Cluster 2: “transit/accessibility”
- Cluster 3: “months”
- Cluster 4: “student services”
- Cluster 5: “programs”

Discussion

For the above clusters’ top terms for k=3 and k=6, it can be seen that when the cluster size increased more relevant top terms were present in the list. This allows for easier association of a cluster name, due to the availability of more classifications. This causes the otherwise distanced documents in higher-dimensional space, to more likely be included in the correct cluster. At a low k value such as 3, these outlying or distanced document vectors will be most likely be included into a cluster that they are not necessarily associated with. However, it’s important to note that an excessively large value for k is equally detrimental in the opposite sense. This can create classifications that are meaningless, and often redundant since they would differ very insignificantly from their neighbouring clusters.

Cluster sentiment scores

Below can be seen the sentiment scores of the clusters, whose top 20 terms are listed above.

```
*** CLUSTER SENTIMENT SCORES ***
CLUSTER 0: SENTIMENT SCORE: 16.58543789079642
CLUSTER 1: SENTIMENT SCORE: 2.165374103495467
CLUSTER 2: SENTIMENT SCORE: 185.0715475536373
```

*** CLUSTER SENTIMENT SCORES ***
CLUSTER 0: SENTIMENT SCORE: 16.489862708230145
CLUSTER 1: SENTIMENT SCORE: 79.0548954943854
CLUSTER 2: SENTIMENT SCORE: 2.5889765722140132

CLUSTER 3: SENTIMENT SCORE: 24.054544270526126
CLUSTER 4: SENTIMENT SCORE: 28.893500636869224
CLUSTER 5: SENTIMENT SCORE: 106.7651722311038

Discussion

Want can be derived from these sentiment scores is that the clusters with less-neutral terms are more likely to have a sentiment score that is either higher, or in the negatives. Meaning, the clusters with positive terms such as *sustainability* or *health* are going to contribute higher scores, while terms such as *emergency* or *stress* will yield lower scores. In the middle, there are more neutral terms such as *November* or *west* which don't carry any inherent sentiment. These would contribute to a more neutral sentiment score which is why cluster 3 of k=6 is among the lowest scores.

Apart from the above technicalities, what can be noticed is that the value of k had little influence on the sentiment scores given to the nearly-equivalent clusters, for k=3 and k=6. Meaning, upon noticing the near equivalence of cluster 1 of k=3 to cluster 2 of k=6, it can be seen that they yield a near equivalent sentiment score. The same can be said for both cluster 0 of k=3 and k=6. For cluster 2 of k=3, which carried high sentiment score despite the terms having hardly any relation, it can be seen that it is almost "split" amongst clusters 1,3,4, and 5 for k=6, since they all contain a proportion of the top 20 terms found in cluster 2 for k=3. Despite some of these clusters carrying lower sentiment scores, their terms are much more in tune with one another. This suggests that the sentiment scores are not very context-sensitive or context-representative.

Overview

This project presented its challenges, mainly in the earlier stages as well as coming up with a formula for cluster sentiment scores. Although quite simple in nature, the crawling/scraping was challenging due to having to mitigate getting flagged by the concordia server, as well as finding relevant textual data to scrape. This was particularly difficult given the rather arbitrary design and structure to the html pages crawled to, within the concordia domain. The solution had to be strict enough to mitigate collecting meaningless data, yet also lenient enough to avoid being lossy due to the variance in the structure of the web pages. The clustering sentiment formula was also quite challenging to figure out, as I tried to make it simple, yet effective. A simple formula like an average was too easily influenced by anomalies, and so settling on a weighted average seemed the most effective approach. Another challenge was figuring out how to normalize the values obtained while calculating the cluster sentiment score as they could sometimes vary being either very low or very high.