REPORT

INFORMATION RETRIEVAL AND WEB SEARCH

(COMP 6791)

CONCORDIA UNIVERSITY

PROJECT 4

SUBMITTED BY:

MANJIT SINGH

(40185580)

ABSTRACT

We have created a crawler for scraping the data from our university website i.e, "*concordia.ca*". In order to create a crawler I have used the *scrapy* package to successfully crawl on domain "concordia.ca" and download the pages. I have designed a special parameter "*file_count*" as per the project requirements that will limit the crawler on how many pages it should crawl and download.

Once the pages are successfully downloaded then I have treated each page as one document and extracted the text from it using *BeautifulSoup*. Then we do text preprocessing such as removal of punctuation and numbers, lower casing the text , stemming the text etc. Once the preprocessing is done, I have applied the k-means clustering algorithm from the scikit-learn package. Once we did clustering with 6 clusters and then with 3 clusters to notice the behavior. At the end for both the cases I have calculated the *AFINN* score.

DESIGNING OF WEB CRAWLER

TOOL USED

Package – Scrapy
Version – 2.4.1

WHAT IS SCRAPY

Scrapy is a Python framework which helps us efficiently crawl any website and scrape the relevant data. It also gives us the power to process the extracted data according to one's requirements and also gives the choice on in which format we want to store the data.

WHY SCRAPY

- Scrapy is faster than other python frameworks.
- Scrapy can make requests to multiple pages in parallel instead of one at a time.
- Scrapy is memory and CPU efficient.

CODE DESIGN

I have created a class in python called "*ConcordiaSiteSpider*". In scrapy we create a class and we pass the *spider* object which will help us to crawl. In the class the only allowed domain to crawl on is "*concordia.ca*". We have a built-in "*parse*" function in scrapy, which we have overridden. In the parse function we are using the LinkExtractor in-built class which will help us in extracting link from the web pages and we will keep on iterating/crawling on links until the upper bound on how many pages should be crawled is satisfied. The upper bound can be decided at run time.

Now sometimes when we request the url of the pages we might get a negative response, we might not get a page or some error occurred. But we want to anyhow satisfy the upper bound and only consider those pages which we were able to recover successfully. So I have created a function *process_value(self , value).* In this

I have added extract functionality that is when I am requesting the page *url* and getting a response. So, right there I am checking the status code of the response if it is "*200*" that means the page has been successfully crawled upon and being downloaded. If this is the case, I increment the counter of *file_count*. If this is not the case, I don't increment the *file_count*. This way only when the pages are successfully retrieved the file_count variable is being incremented. Hence the crawler will stop when the file_count variable becomes equal to the upper bound being set by the user. This way we have made sure that we have successfully crawled upon and downloaded the pages equal to upper bound no more or no less. I am saving the pages as ".html" files in the folder called "*HTML*", which is present in the same directory as "*.py*" file of the code.

TEXT EXTRACTION AND TEXT PREPROCESSING OF THE DOWNLOADED PAGES

Now we have successfully downloaded the pages. So, the next step is to extract the text from these pages. In my case, I have used "*BeautifulSoup*" to extract the text. I have used the "*soup.get_text()*" function and In this text I have discarded the text of javascript code and css style code using "*script.decompose()*" and "*style.decompose()*". As, this text is irrelevant to us and would result in wrong clustering results. So all these things are happening in the file called "Clustering.py" inside the function named "*crawl_and_extract_data()*".

Once the text of all the pages has been extracted we need to correct it and convert it to the right form so that we can successfully run clustering algorithms on them to get the most optimized results. So, I have applied the following preprocessing techniques on the extracted text.

- Removal of punctuations.
- Removal of numbers.
- Lower-Casing the text.
- Removal of stop words.
- Stemming.

Now the extracted text is ready for next step.

CLUSTERING OF DOCUMENTS

Clustering Algorithm.

"*KMeans*" from sci-kit learn package.

What is KMeans?

It is an unsupervised machine learning algorithm. It takes all the data points and groups them into fixed number of clusters. Basically on the plane it randomly takes n-points(n-number of clusters) and start assigning the data points to these n points based on their distance from them. In the beginning when the mentioned first step is done then it keeps on computing the centroids again and keeps on assigning the data points to these n-centroids until no new result or no more optimization can be achieved. At the end with good certainty we can safely say documents in each cluster and are similar to each other.

CODE DESIGN

First, we compute the TF-IDF scores of each document because Kmeans cannot make clusters based on text data it needs numeric data to compute distance from centroids and assign the cluster and also we don't want to assign random numbers to our documents so the TF-IDF scores weighs each document based on some predefined generalized rules and assigns the score to each document.

"compute_tf_idf_scores()" this function computes the tf-idf score of each document using the Tfidfvectorizer package from python and returns the data in the relevant structure that KMeans can use.

Now, "k_means_clustering()" function takes that data and runs the clustering alogrithims on those set of documents, assigns them a cluster and then we find the top most 50 terms with highest tf-idf score from each cluster to see which cluster can be given which name.

We can easily get top terms using the "*get_feature_names()*" in-built function.

SENTIMENT SCORE USING AFINN

At this point when clusters have been already made. We want to find out the sentiment analysis of each cluster. In order to do that we again try to compute some score and assign it to each cluster. Based on that score we try to find whether a particular cluster has a positive sentiment or a negative sentiment.

In order to achieve that we have used the *AFINN* python package. Basically, each word already has a score between -5 and +5. And then we try to compute the AFINN score for the whole document based on the score of each word.

"*afinn_score(cluster_number,centroids_term)*" this function basically computes the AFINN score for each cluster but instead of using all the documents text from each cluster we only use the top 50 terns with the highest *tf-idf* score of each cluster.

STRENGTHS

- Easy To Understand.
- Highly Modular.
- Time and space complexity are optimized.
- Easy to test the code with the help of the main Function.
- Code is able to extract all the pages.

SHORTCOMINGS

- Time Complexity can be optimized even more.
- Space Complexity can be optimized even more.
- Code can be made more readable and reusable by defining more generic functions.

TAKEAWAY

- I had gained better understanding of Information Retrieval Methodology in general.
- I gained the experience on working with web crawling techniques.
- I gained better understanding of extracting data from live websites.
- I gained better perspective of how to select useful information and retrieve it from very big unstructured or partially structured data and restructure it.
- How to do unsupervised Clustering.