How to run

-> Run main.py

-> All outputs are in /outputs directory. You can generate a new demo file if you change the constants mentionned bellow.

-> Downloaded links are stored in the /paths directory. Delete the files within the directory to regenerate them.

The constants at the top of the file allow setting upper bounds on crawling. Each further block in the file is a requirement of the assignment (naming should be clear).

Libraries used:

- I used python 3.8
- Libraries are in requirement.txt

Url crawling

The logic for this section lives in link_crawler directory. My code is inspired from this library file (I made a lot of modifications): https://github.com/x4nth055/pythoncode-tutorials/blob/master/web-scraping/link-extractor/link_extractor.py

Like in a sitemap generation, the algorithm recursively visits urls from the anchor tags in a given page, starting from the concordia main page and stops when the upper bound is reached. The upper bound on the urls to retrieve is set by constant URL_TO_FETCH_COUNT in main.py and consumed by then class LinkCrawler. To reduce the charge on Concordia website and to improve performance, if the file containing the fetched urls is present (under paths/) this step will be skipped.

Robots.txt

This file gives crawling directives to remote servers, llike crawlable paths and requests delays.

I used robotparser library from urllib to parse robots.txt file. It gives a nice wrapper were you can easily access values like the crawl delay and the scrappabled paths. I created my own class wrapper over this library to fit my needs (robot_parser). This class is given to the the classes fetching links so that we can only keep urls that are scrappable. It is also given to the html_crawler class so that we fetch at a rate and in a frequency that is expected in the robots.txt.

Implemented in method: get_is_crawlable()

Html crawling

The upper bound on the number of HTML pages to be retrieved from the bank of pre-fetched URLs in the paths/ directory is set in main.py by the PAGES_TO_FETCH constant and consumed by the method fetch_concordia_internal_links_html of the class HtmlCrawler. As stated in the assignment, I used BeautifulSoup library to parse the pages retrieved. Once parsed into a Soup object, HTML docs are returned as text and without HTML tags. The logic for this section lives in the html_crawler directory.

Clustering

I am using sklearn KMeans class to cluster the previously retrieved and transformed HTML documents. The text is vectorized with the KMeans TfidfVectorizer library. English and french stopwords are removed. IDF and DF values are kept to defaults. From main.py, two clustering runs are made, one with 3 clusters and the other with 6 clusters. The maximum number of iterations of the kmeans algorithm is set to 100. The logic for this section lives in the k_means directory.

Clusters assessment in the *Good and bad clusters examples* section.

Finding the clusters' name

The algorithm:

1. For each cluster find terms ranked in order of tf-idf frequency. Luckily, using TfidfVectorizer allows us to easily retrieve terms ranked by their tf_idf frequency for each cluter.
2. Keep of a black list of non-relevant frequent terms like "Concordia"
3. Assign the cluster's term with the best tf-idf score that is not in the black list as cluster title

Example for three clusters: `['accelerating', '2020', 'letters']`

Sentiment analysis

https://github.com/fnielsen/afinn library is used. The logic is nested in ClusterAfinn class.

Formula description:

1. Assign each TfidfVectorizered document to a cluster with k-mean
2. Calculate distance to centroid center for each document
3. Break down each document into an array of sentences
4. For each sentence of each document calculate the Afinn score
5. Each document is assigned a sentiment score that is the average of its sentences Afinn scores
6. Each centroid is assigned a sentiment score that is the weighted average of its document score. The documents are weighted by their distance from the centroid.

The closer the document, the more weight. The sum of the weights for each cluster is one.

Example results for three clusters (in order): `{0: 0.17, 1: 0.19, 2: 0.11}`

Good and bad clusters examples

Using SSE as the metric:
For 500 documents: 6 clusters slightly improves the error score 87 to 80
For 100 documents: 6 clusters slightly improves the error score 51 to 47
For 20 documents: 6 clusters slightly improves the error score 7 to 4

The more documents, the better the SSE drop, but the lower the drop in percentage. When increasing the number of documents the marginal gain of adding clusters is then going down (for a fixed amount as in this assignment). The more the documents, the logarthmic is the increase in SSE. The best cluster will maxime the SSE for a givent number of documents. In my case this is represented by the low and hight document counts clusters. The cluster count for a high number of documents did not make a lot of difference on that ratio.

SSE values can be found in the demo.txt file. Since the file is regenerated on every run, please change te constants in main.py to experiment with new values.
URL_TO_FETCH_COUNT, PAGES_TO_FETCH

Most informative terms per cluster

Cluster title is top tf-idf score term per cluster. To generate this list, I simply took the top 50 from the same algorithm. See section on cluster title for more information.

-> Files are located at: outputs/cluster_count_50_most_informative_terms.txt

Sentiment values analysis

Sentiment values that were assigned to clusters during most runs are in the interval of 0.15 to 0.5 (in Demo file). As -6 is maximum negativity, 0 is neutral and 6 is maximum positivity we can say the results of this experiment is that most documents have a slightly positive sentiment. This is not surprising, as most educational institutions will try to keep a neutral tone. Furthermore, when talking of oneself, it's understandable that the sentiment is slightly leaning toward the positive side (Concordia wants to promote itslef !).

Your experience with crawling and scraping web pages

I previously built a commercial application with an Instagram integration for a local marketing agency. As Instagram's API is very strict, many features relied on scraping the HTML from Instagram profiles. From this previous experience, I would say that the main challenge when scraping is staying on good terms with the server. When respecting the robots.txt is an option to meet business requirements this task is more or less trivial. As stated, scrape the

right paths with the right frequency and there should not be issues. The real complexity appears when the business needs exceed permissions given in the robot.txt. Distributed server-side crawling, client-side crawling and human user-agent behaviour when making requests are solutions that I have put in place in the past to scrape web pages (in my case, Instagram). Nevertheless, I faced many issues like exponential backoff and IP bans.

In the case of this assignment, URLs were downloaded locally and a very low amount of HTML documents were retrieved in order to keep the charge at its minimum on Concordia's website.