COMP 479 – Final Project Report

François Crispo-Sauvé – 27454139
Roger Shubho Madhu – 40076461
Vartan Benohanian – 27492049

Sentiment Web Crawler

December 4th, 2018

OVERVIEW

The team agreed to use Python (more specifically, Python 3) to build the system, thanks to its simple syntax and ability to implement ideas quickly. It also has an abundance of open-source, community-driven, useful libraries. The ones used in this project are:

- scrapy to crawl web pages and parse its contents.
- nltk to tokenize web pages' contents.
- afinn to provide sentiment analysis on terms, queries, and documents.
- tabulate to generate aesthetic tables for query results.

GOALS

1. Experiment with web crawling and scraping.
2. Experiment with indexing a set of documents.
3. Use the sentiment dictionary to associate sentiment values to the index.
4. Conduct queries to the index, while making document ranking reflect sentiment.

SPECIFICATIONS

Scrapy helped a ton in scraping desired content from web pages, especially links. It provides a very clean way to get the information we want from web pages, all while keeping the code we write to a minimum. After some practice playing around and experimenting with the scraping framework, it essentially boiled down to specifying 2 types of information:

1. Number of links to scrape.
2. Which HTML tags to extract text from.

The rest was taken care of by the framework. Our crawler inherits from the parent *CrawlSpider* class, which provides a way to extract and follow links based on a set of rules, like domains to deny (more on that later). It never scrapes the same link twice, which is very convenient as we

don't want duplicate data. Scrapy writes the results of its crawl to a JSON file (*results.json*). After a crawl, the file contains an array of JSON objects, one object for each page scraped. For each object, we assigned a *url* property (the URL of the page), and a *content* property (the text scraped from the page).

There were still some bumps on the way to getting it to work fully as intended. For example, it took us a while to fully figure out how to control the upper bound on links, as well as other configurations in the crawler.

The *CrawlerProcess* object helps us define these configurations. It comes with a predefined set of default settings, but the ones of interest to us are:

- FEED_FORMAT: the format under which our data will be returned. We used JSON.
- CLOSESPIDER_ITEMCOUNT: the limit, or "upper bound", on links scraped. Once the spider crawls through *x* links, it will stop.
- CONCURRENT_REQUESTS: the maximum number of concurrent requests. The default is 16, but we set it to 1 so that the upper bound is respected. That was confusing at first.
- ROBOTSTXT_OBEY: whether or not the spider will respect robot exclusion of websites. The default is true.

Although it's not in the requirements of the project, the application is built in a way so that the user can specify the URL that the spider will begin crawling from, as well as whether it will respect robot exclusion.

Running

Make sure to use Python 3 and install the packages in the *requirements.txt* file of the submission. The script to run is located in the *src/* directory. To run the program, enter the following in the command line: **python3 main.py [arguments]**. The arguments are optional. They are the following:

- -m or --max: maximum number of links to scrape. Default is 10.
- -url or --start-url: URL the crawler will begin scraping links from. Surround it with quotes in the command line for best results. Default is the about page of Concordia University.
- -ign or --ignore-robots: the crawler will not respect robot exclusion. Default is false.
- -rs or --remove-stopwords: stopwords will be removed from the index and ignored in queries. Default is false.
- -skip or --skip-crawl: the crawler won't run, and you will use the data set from its most recent run to conduct queries. Default is false.

Examples: python3 main.py, python3 main.py -m 20, python3 main.py --max 57

Note 1: It's important to note that only the first argument was a requirement to have for the project. The others were added mostly for fun and experimenting.

Note 2: The *max* argument can be a bit deceiving. Let's say we set an upper bound of 23. It will scrape the starting page, as well as 23 *other* pages, more or less. We're not too sure why, but sometimes it scrapes an additional 1 or 2 links. After failing to figure out why it does that, we figured it's not of utmost important to have that hard of a cap on the upper bound.

What is Happening?

Here is a detailed step-by-step demonstration of what's happening in the *main.py* file:

1. First, we make sure there's no *results.json* file (generated by the crawler), in the current directory. If there is, we delete it, so that the spider can generate a new one.
2. We initialize the spider, and instruct it to crawl, passing in the upper bound on links to crawl. We also pass in the start URL and a robot exclusion obedience boolean, but for the purpose of this project, those are always the same.
    a. We were getting lots of links containing insignificant content, such as password reset links, repeatedly in different languages. We decided to exclude the following domains, by including them in the *deny_domains* property of the crawler's rule: *facebook.com, twitter.com, youtube.com, google.com, stm.info, apple.com*.
    b. We were getting a lot of French links, as well as Concordia campus maps links. We decided to avoid those by including the following regular expressions in the crawler's rule's *deny* property: *.*/fr/.*, .*concordia.ca/maps/.**
3. The crawler appends objects to the *results.json* as it crawls through links:
    a. First, it stores the URL of the page in the object.
    b. Second, it stores the page's contents in the object. We parse the following HTML tags, as we feel they contain the most relevant textual information on a webpage, all while making sure to exclude *<script> and <style>* tags:
        i. header, h1, h2, h3, h4, h5, h6, p, span, footer
    c. Using nltk's *word_tokenize* method, we tokenize the content of the page. Then, we convert everything to lowercase. We also make sure to exclude strings that are only made up of punctuation marks. If specified, we also remove stopwords.
    d. In the end, the file contains an array of objects, with 2 properties each:
        i. url: the URL of the page.
        ii. content: the list of tokens extracted.
4. Using this *results.json*, our *DocumentParser object* parses it using Python's *json* package, part of Python's standard library. Assuming one web page corresponds to one document, for each document, it calculates and writes to a *url_stats.txt* file:

a. the total number of tokens extracted from the document.
b. the total Afinn score of the tokens.
c. The *DocumentParser* also outputs at the end a total tally of some interesting stats: total #documents, total and average #tokens, total and average Afinn score.
d. The stats are written for each URL in the *url_stats.txt* file in the following order: URL, #total tokens in page, total Afinn score.

5. Even though the statistics are written in a file, we still store it in a dictionary object. We initialize our *IndexBuilder* object with it, since we'll need some of those stats to store term frequency-inverse document frequency of terms. The object builds the index, storing **distinct** terms. Most of the time, we decided to keep stopwords in the index for more consistent results. What if a user wants to search for "the"? They can. Still, the option to remove them exists. For each term, we store:
    a. the term's Afinn sentiment value.
    b. the term's document frequency (df_t), i.e. the number of documents it appears in.
    c. the term's collection frequency (cf_t), i.e. the number of times it appears in the corpus.
    d. the term's inverse document frequency (idf_t), i.e. $\log_{10}$(total #documents / df_t).
    e. the scraped page's URL that the term appears in, and for each page:
        i. the term's frequency (tf), i.e. the number of times it appears in the page.
        ii. the term's frequency-inverse document frequency (tf-idf), i.e. the number representing how important it is to the collection of documents (in this case, the collection of pages crawled).
    f. Note: For the tf-idf calculation, we use our *TFIDF* class, which takes in the index and document statistics objects as parameters.
    g. The index is then written to the *index.txt* file, each line in the following order: term, cf_t, df_t, idf, sentiment score, [URL, tf, tf-idf for each page the term is in].
6. Finally, we initialize our *Query* object, which asks the user if they would like to conduct queries against the resulting index. The user first chooses if they would like to conduct an *AND* query (*AndQuery* object) or an *OR* query (*OrQuery* object). Once they submit a query, they get its overall Afinn sentiment score, as well as a table showcasing results matching their query (if there are any at all). The results are partially sorted: first by cosine similarity, descending, then the top 10 are sorted by Afinn score of the document, ascending if the query's score is less than 0, descending if it's greater than 0, unsorted by Afinn score otherwise. The table of results has 3 columns:
    a. The cosine similarity between the query and the document.
    b. The overall Afinn score of the document.
    c. The URL of the document.

SAMPLE RUNS

The following queries were conducted with an upper bound of 100 links. The sample queries provided in the project description didn't return anything, so we came up with our own.

AND Query: *great education* *(notice sorting by Afinn score)

```
AndQuery: great education
Sentiment value: 3.0
4 page(s) found:
```

cosine similarity	Afinn score	url
0.882378	221	http://www.concordia.ca/cunews/main/stories/2014/05/28/co-op-s-moment-inthesunexperientialinstitutecelebratesa34thsucce.html
0.882378	102	http://www.concordia.ca/news/stories/topics.html?topic=topics:general_interest/teaching
0.984282	86	http://www.concordia.ca/news/stories/topics.html?topic=topics:people_places_entities/students
0.958567	39	http://www.concordia.ca/jmsb/news.html

AND Query: *technology future generation*

```
AndQuery: technology future generation
Sentiment value: 0.0
1 page(s) found:
```

cosine similarity	Afinn score	url
0.84189	156	http://www.concordia.ca/news/stories/topics.html?topic=topics:people_places_entities/alumni

AND Query: *water issue*

```
AndQuery: water issue
Sentiment value: 0.0
1 page(s) found:
```

cosine similarity	Afinn score	url
0.944769	17	https://www.concordia.ca/cunews/main/stories/2018/12/03/water-ev-building.html

AND Query: *web development*

```
AndQuery: web development
Sentiment value: 0.0
7 page(s) found:
```

cosine similarity	Afinn score	url
1	-67	http://legisquebec.gouv.qc.ca/en/ShowDoc/cs/A-2.1
1	102	http://www.concordia.ca/news/stories/topics.html?topic=topics:general_interest/teaching
0.979879	34	http://www.concordia.ca/cunews/main/stories/2014/06/11/how-concordia-researchersarefightingonlinehatespeech.html
0.919761	-55	https://www.trustarc.com/terms-of-use/
0.919761	162	https://www.trustarc.com/privacy-policy/
0.874276	26	https://www.concordia.ca/web/accessibility.html
0.874276	26	http://www.concordia.ca/web/accessibility.html

OR Query: *Amazon architecture*

```
OrQuery: Amazon architecture
Sentiment value: 0.0
6 page(s) found:
```

cosine similarity	Afinn score	url
0.77203	26	http://www.concordia.ca/web/accessibility.html
0.77203	26	https://www.concordia.ca/web/accessibility.html
0.635586	72	http://www.concordia.ca/cunews/main/stories/2015/01/12/co-op-an-essential-stepping-stone-to-amazon-and-ibm.html?c=news/stories/topics
0.635586	84	https://www.shopify.co.uk/
0.635586	82	https://www.shopify.ca/?ref=style-hatch
0.635586	102	http://www.concordia.ca/news/stories/topics.html?topic=topics:general_interest/teaching

AND Query: *social media generation*

```
AndQuery: social media generation
Sentiment value: 0.0
5 page(s) found:
```

cosine similarity	Afinn score	url
0.780848	111	http://www.concordia.ca/news/stories/topics.html?topic=topics:academic_disciplines/business
0.774277	88	http://www.concordia.ca/cunews/offices/vpaer/aar/2018/02/01/creating-a-promising-digital-future-for-quebec-.html?c=news/stories/topics
0.725331	156	http://www.concordia.ca/news/stories/topics.html?topic=topics:people_places_entities/alumni
0.538389	34	http://www.concordia.ca/cunews/main/stories/2014/06/11/how-concordia-researchersarefightingonlinehatespeech.html
0.465171	23	http://www.concordia.ca/cunews/main/stories/2015/08/19/avoid-social-media-mishaps-pitfalls-digital-footprint.html

AND Query: *great Concordia students* *(notice sorting by Afinn score)

```
AndQuery: great Concordia students
Sentiment value: 3.0
6 page(s) found:
```

cosine similarity	Afinn score	url
0.537845	221	http://www.concordia.ca/cunews/main/stories/2014/05/28/co-op-s-moment-inthesunexperientialinstitutecelebratesa34thsucce.html
0.483461	185	https://www.concordia.ca/hospitality/event-planning/food-at-events/approved-catererslist.html
0.483346	156	http://www.concordia.ca/news/stories/topics.html?topic=topics:people_places_entities/alumni
0.416209	102	http://www.concordia.ca/news/stories/topics.html?topic=topics:general_interest/teaching
0.46453	86	http://www.concordia.ca/news/stories/topics.html?topic=topics:people_places_entities/students
0.538019	39	http://www.concordia.ca/jmsb/news.html

AND Query: *bad Concordia students* *(notice sorting by Afinn score)

```
AndQuery: bad Concordia students
Sentiment value: -3.0
3 page(s) found:
```

cosine similarity	Afinn score	url
0.467752	23	http://www.concordia.ca/cunews/main/stories/2015/08/19/avoid-social-media-mishaps-pitfalls-digital-footprint.html
0.398121	62	http://www.concordia.ca/news/stories/topics.html?topic=topics:academic_disciplines/education
0.376207	156	http://www.concordia.ca/news/stories/topics.html?topic=topics:people_places_entities/alumni

CONCLUSION

What was the Hardest Step?

Before the project, none of the team members had used Scrapy. There was a bit of a learning curve in getting familiar with it, but its extensive documentation proved to be more than enough in getting the project to its current state.

In relation to the parsed contents, there was a bit of a debate as to what to keep from a web page's HTML. We also ran into issues regarding the crawler parsing *<script>* tags that were in regular tags such as *<p>*. There was some *xpath* tinkering but it was dealt with fairly quickly.

Other than that, the knowledge gained from the previous projects helped a great deal in implementing the rest of the features, such as the index and the querying.

How Big is the Index?

The size of the index varies depending on the upper bound set by the user on the number of pages visited by the spider, as well as which links the spider decides to visit. With the default upper bound (10), we notice there are about 1300 **distinct** terms in the index, with about 750 tokens per document.

What Constitutes a Document?

In this project, one web page corresponds to one document. For example, if the spider visits 25 web pages, we will have 25 documents at our disposal.

Observations and Experiments

The biggest thing we observed is how the Afinn sentiment score of a document can greatly vary depending on the subject. For fun, we decided to parse two Wikipedia articles opposing in subject matter: love (roughly 7400 tokens) and anger (roughly 10,000 tokens).

The one about love has a great positive Afinn sentiment score, at 1,615. This indicates that the attitude throughout the content of the article is overwhelmingly positive, a fact that can be derived since it talks about a positive emotion.

However, the article centered around anger has a big negative Afinn sentiment score of -1,505, indicating that the subject matter is a negative one, obviously.

We can use this kind of information when scraping data of reviews of various things, like movies, restaurants, airlines, etc. We can also use it for responses to surveys to gauge data gathered from customers of a business.

What We Learned

The biggest takeaway from the project is how sentiment analysis can be used to gauge a document's overall emotion from its contents.

We of course also learned a lot about web scraping and crawling through various links, either to build a search engine, or even extract specific types of information that we would want to manipulate or store somewhere.

Individual Contributions

When the decision was made to use Scrapy, all 3 team members agreed to practice playing around and experimenting with it. When it came to actually implementing it for the project, François took care of deciding what kind of content to extract from web pages, while Roger did some extensive research on the possible configurations of the spider. While running the crawler through some test runs, we all noticed some repetitive, low-value web pages being crawled through, so we all had a part in determining which ones to omit (e.g. *www.twitter.com*).

A lot of the rest of the code depended on some other part of the code, i.e. one's work depended on another teammate finishing first. This made it difficult to evenly separate tasks amongst ourselves.

The parsing of the web page's content to generate statistics was done by François fairly quickly. The creation of the index was done by Vartan. Roger then modified the index a little bit by adding values such as term frequency, collection frequency, etc.

Querying to the index was done by Vartan, since a lot of the logic pertaining to result retrieval, as well as the results table generation, was similar to that of his previous projects.

As the project came to a close, all teammates participated in testing the crawler on their individual machines, and reporting the smoothness of the runs.

Although not a requirement, Vartan also took care of containerizing the application with Docker, to ensure everyone can have a working environment up and running within a few seconds without any trouble.

This report was made on Google Docs.