Report about Project 3 - Sentiment Analysis

1. Lei Xu

2. Peter Chen

3. Yufeng Ding

Table of Content

1. Background

In the final project, We will do an experiment with web crawling, infomation extracting and indexing to make document ranking reflect sentiment with the help of **aFinn** sentiment dictionary. And then We will do a comprehensive tests and analyze results.

2. Introduction

2.1 Information Crawling & Extracting

The first and important thing of the project is to generate source data for following operations, like indexing and query. The method we apply here is to crawl webpages and then extract useful information stored in text files.

2.1.1 Web Crawlers – Scrapy

Even though **Scrapy** was originally designed for web scraping, it can also be used to extract data using APIs (such as Amazon Associates Web Services) or as a general purpose web crawler, in a fast, simple, yet extensible way.

> **Scrapy** is an application framework for crawling web sites and extracting structured data which can be used for a wide range of useful applications, like data mining, information processing or historical archival.

The following diagram shows an overview of the Scrapy architecture with its components and an outline of the data flow that takes place inside the system (shown by the red arrows). A brief description of the components is included below. The data flow is also described below.

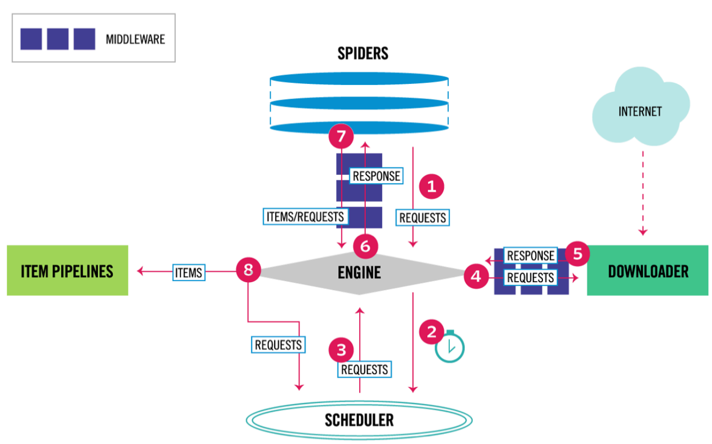


Figure 2.1 Scrapy Data Flow

The data flow in Scrapy is controlled by the execution engine, and goes like this:

1. The Engine gets the initial Requests to crawl from the Spider.
2. The Engine schedules the Requests in the Scheduler and asks for the next Requests to crawl.
3. The Scheduler returns the next Requests to the Engine.
4. The Engine sends the Requests to the Downloader, passing through the Downloader Middlewares.
5. Once the page finishes downloading the Downloader generates a Response and sends it to the Engine, passing through the Downloader Middlewares.
6. The Engine receives the Response from the Downloader and sends it to the Spider for processing, passing through the Spider Middleware.

7. The Spider processes the Response and returns scraped items and new Requests to the Engine, passing through the Spider Middleware.
8. The Engine sends processed items to Item Pipelines, then send processed Requests to the Scheduler and asks for possible next Requests to crawl.
9. The process repeats (from step 1) until there are no more requests from the Scheduler.

2.1.2 Text Extractor – Beautiful Soup

Text extraction is the task of separating boilerplate such as comments, navigation bars, social media links, ads, etc, from the main body of text of an article formatted as HTML.

In this part, we decide to introduce the **Beautiful Soup 4** framework into the project based on the three following features.

> **Bautiful Soup**, is a Python library for pulling data out of HTML and XML files. It works with your favorite parser to provide idiomatic ways of navigating, searching, and modifying the parse tree.

1. Beautiful Soup provides a few simple methods and Pythonic idioms for navigating, searching, and modifying a parse tree: a toolkit for dissecting a document and extracting what you need. It doesn't take much code to write an application
2. Beautiful Soup automatically converts incoming documents to Unicode and outgoing documents to UTF-8. You don't have to think about encodings, unless the document doesn't specify an encoding and Beautiful Soup can't detect one. Then you just have to specify the original encoding.
3. Beautiful Soup sits on top of popular Python parsers like lxml and html5lib, allowing you to try out different parsing strategies or trade speed for flexibility.

2.2 Information Indexing

2.2.1 SPIMI Algorithm

To implememnt rudimentary information retrieval, we will create an indexer using the **SPIMI** algorithm.

> **SPIMI** uses terms instead of termIDs, writes each block's dictionary to disk, and then starts a new dictionary for the next block. SPIMI can index collections of any size as long as there is enough disk space available.

The SPIMI algorithm pseudocode is shown below:

```
SPIMI-INVERT(token_stream)
    output_file = NEWFILE()
    dictionary = NEWHASH()
    while (free memory available)
    do token <- next(tolen_steam)
        if term(token) ∉ dictionary
            then postings_list = ADDTODICTIONARY(dictionary, term(token))
            else postings_list = GETPOSTINGSLIST(dictionary, term(token))
        if full(postings_list)
            then postings_list = DOUBLEPOSTINGSLIST(dictionary, term(token))
        ADDTOPOSTINGSLIST(postings_list, doc_ID(token))
    sorted_terms <- SORTTERMS(dictionary)
    WRITEBLOCKTODISK(sorted_terms, dictionary, output_file)
    return output_file
```

The part of the algorithm that parses documents and turns them into a stream of term-docID pairs, which we call tokens here, has been omitted. SPIMI-INVERT is called repeatedly on the token stream until the entire collection has been processed.

Tokens are processed one by one during each successive call of SPIMI-INVERT. When a term occurs for the first time, it is added to the dictionary (best implemented as a hash), and a new postings list is created.

2.2.2 Natural Language Toolkit – NLTK

NLTK will be used to preprocess documents crawled by the scrapy, like tokenization, stemming and removing stopwords.

> **NLTK** is a leading platform for building Python programs to work with human language data. It provides easy-to-use interfaces to over 50 corpora and lexical resources such as WordNet, along with a suite of text processing libraries for classification, tokenization, stemming, tagging, parsing, and semantic reasoning, and wrappers for industrial-strength NLP libraries.

2.3 Probabilistic Search Engine

2.3.1 TF-IDF Algorithm

To reflect how relevant a term is in a given document, we will appy the **TF-IDF** algorithm as a part of the scoring function in the project.

> **TF-IDF** is an information retrieval technique that weighs a term's frequency (**TF**) and its inverse document frequency (**IDF**). Each word or term has its respective TF and IDF score. The product of the TF and IDF scores of a term is called the TF*IDF weight of that term.

Put simply, the higher the TF-IDF score (weight), the rarer the term and vice versa.

The TF-IDF algorithm is used to weigh a keyword in any content and assign the importance to that keyword based on the number of times it appears in the document. More importantly, it checks how relevant the keyword is throughout the web, which is referred to as corpus.

For a term t in a document d, the weight $W_{t,d}$ of term t in document d is given by:

$$W_{t,d} = tf_{t,d} \times idf_t$$

We can see a few common components like $tf_{t,d}$ and idf_t. Here's what each of these is all about:

1. $tf_{t,d}$ is the number of occurrences of t in document d.

   For example, when a 100 word document contains the term "sentiment" 12 times, the TF for the word 'sentiment' is

   $$tf_{sentiment} = 12/100 = 0.12$$

   But relevance does not increase proportionally with term frequency. We here decide to apply **Log frequency weighting** instead of raw frequency.

   $(1 + \log tf_{t,d})$ if tf_{t, d} > 0

2. idf_t is a measure of the informativeness of the term t.

   The IDF component of our formula measures how often a term occurs in all of the documents and "penalizes" terms that are common. The actual formula tf-idf uses for this part is:

   $$IDF(q_i) = \log[\frac{N}{df_t}]$$

   - **N** is the number of documents in the collection
   - df_t is the document frequency of the ith query term.

So, the final **tf-idf** formula that we will use in the project is:

$$W_{t,d} = (1 + \log tf_{t,d}) \times (\log \frac{N}{df_t})$$

With words having a high TF*IDF weight in our content, our content will always be among the top search results, so benifits that we have are:

1. Stop worrying about using the stop-words,
2. Successfully hunt words with higher search volumes and lower competition,
3. Be sure to have words that make our content unique and relevant to the user, etc.

2.3.2 Sentiment Dictionary - aFinn

To associate sentiment values to each term in our index, we will apply **aFinn**, specifically **AFINN-111.txt**, as the sentiment dictionary.

> **AFINN** is a list of English words rated for valence with an integer between minus five (negative) and plus five (positive). The words have been manually labeled by Finn Årup Nielsen in 2009-2011. The file is tab-separated.

Word associated with sentiment score between -5 **(most negative)** and $+5$ **(most positive)**, like:

1. abandon -2
2. abhor -3
3. ability 2
4. aboard 1

2.3.3 Sentiment Aggregation Function

To associate sentiment to individual documents, we have to develop a simple sentiment aggregation function to calculate their sentiment values.

Our idea is to take some text (e.g. document text, query text) as argument. It then splits all the text into lowercased words, and if some of the words appear in the AFINN-111.txt file, the associated values to the words are summed up to provide the sentiment score. Here is the code:

Python

```python
def sentiment_aggr_function(some_text):
    afinn = dict(map(lambda p: (p[0], int(p[1])), [line.split('\t') for line in open("AFI
    sentiment_aggr = sum(map(lambda word: afinn.get(word, 0), some_text.lower().split()))
    return sentiment_aggr
```

2.3.4 Ranking Function

To rank retrieved documents, we have to develop a simple ranking function to sort them by a partial order on (w, s)

Before formulating the partial order function, it is necessary to obtain the values of the arguments needed in the function (i.e. w and s). The approaches taken for obtaining these values are as follows:

1. w is tf-idf based cosine distance to the query. The function for finding the cosine similarity between a query and document is

   $$\cos(\vec{q}, \vec{p}) = SIM(\vec{q}, \vec{p}) = \frac{\vec{q} \times \vec{p}}{|\vec{q}||\vec{p}|} = \frac{\sum_{i=1}^{|V|} q_i \times d_i}{\sqrt{\sum_{i=1}^{|V|} q_i^2}\sqrt{\sum_{i=1}^{|V|} d_i^2}}$$

   1. q_i represents the tf-idf weight of the term i in the query.
   2. d_i represents the tf-idf weight of the term i in the document.
2. s is a sentiment bias. As mentioned in the project instruction, the sentiment bias should obey the following principle:
3. If the query has **overall positive** sentiment, set $s_1 \leq s_2$ if s_1 is **more positive** than s_2.
4. If the query has **overall negative** sentiment, set $s_1 \leq s_2$ if s_1 is **less positive** than s_2

The approach taken to make this principle apparent is to define S as the difference between the query sentiment S_q and the document sentiment S_d (i.e. $S = S_q - S_d$). This value S will be used as the denominator in the partial order function which will be described below.

After obtaining necessary arguments above, now we can define the partial order function below:

$$f(w, s) = \frac{w}{|S|}$$

Based on this function, a document with a superior weight and sentiment bias will be ranked higher than others. For this project, the emphasis is put on the sentiment bias. In other words, if a document d_1 has a slightly inferior weight than a document d_2, but the sentiment for d_1 is significantly higher, then d_1 will be ranked higher than d_2. For instance, a query has a sentiment value of 10. If a document d_1 with weight 3 has a sentiment value of 5 and document d_2 with weight 2 has a sentiment value of 8, by applying the partial order function:

$$f_1(w_1, s_1) = \frac{2}{|8-10|} = \frac{2}{2} = 1$$

$$f_2(w_2, s_2) = \frac{3}{|5-10|} = \frac{3}{5} \leq 1$$

In this case, d_1 **should be ranked higher than** d_2

3. Design & Implementation

According to the description above, we will separate the program into three parts, and implement it in the following six steps:

1. Crawling Information
    1. Web Crawling
    2. Infomation Extracting
2. Creating Indexer
    1. Data Preprocessing
    2. Indexing
    3. Ranking
3. Query
    1. Query and Rank retrieved documents

3.1 Web Crawling

It is the module responsible for crawling webpages using **Scrapy** framwork.

3.1.1 Creating a scrapy project

First, create a scrapy project named **crawler** in the commmand line.

```
> scrapy startproject crawler
```

This will create a tutorial directory with the following contents:

```
crawler/
    scrapy.cfg            # deploy configuration file
    crawler/              # project's Python module, I will import my code from here
     __init__.py
        items.py          # project items definition file
        middlewares.py    # project middlewares file
        pipelines.py      # project pipelines file
        settings.py       # project settings file
        spiders/          # a directory where I will later put my spiders
            __init__.py
```

3.1.2 Customizing scrapy settings - settings.py

The Scrapy settings allows you to customize the behaviour of all Scrapy components, including the core, extensions, pipelines and spiders themselves.

Here is how we customize my crawler (**web_crawler/settings.py**):

Python

```python
BOT_NAME = 'comp479project3'

SPIDER_MODULES = ['comp479project3.spiders']
NEWSPIDER_MODULE = 'comp479project3.spiders'

# Obey robots.txt rules
ROBOTSTXT_OBEY = True
# Set to BFS
DEPTH_PRIORITY = 1
```

3.1.3 Define the spider – concordia_about_spider.py

Spiders are classes that you define and that Scrapy uses to scrape information from a website (or a group of websites). They must subclass **scrapy.Spider** and define the initial requests to make, optionally how to follow links in the pages, and how to parse the downloaded page content to extract data.

This is the code for my Spider named **about_crawl** to crawl the Concordia university about webpage. Save it in a file named **concordiaaboutspider.py** under the **web_crawler/spiders** directory in my project:

Python

```python
import scrapy
from scrapy.exceptions import CloseSpider
from urllib.parse import urlparse
import re
import os
from pathlib import Path

class ConcordiaAboutSpider(scrapy.Spider):
    name = 'about_crawl'
    allowed_domains = ['concordia.ca']
    start_urls = ['https://www.concordia.ca/about.html']
    max_document_size = 200  # Number of links the crawler will extract
    document_counter = 0
    def parse(self, response):
        if self.document_counter <= self.max_document_size:
            self.document_counter += 1
            soup = BeautifulSoup(response.body.decode('utf-8'), 'html.parser')
            self.extract_content(response.url, soup)
            links = soup.find(id='content-main').find_all('a', href=re.compile(r'.*html$')
            for link in links:
                yield response.follow(link.get('href'), callback=self.parse)
        else:
            raise CloseSpider('max_document_exceeded')
```

3.2 Information Extracting

This part is responsible for extracting all useful information from requests sent by the scrapy engine, coded in the **about_crawl** spider file **(spiders/concordia_about_spider.py)**.

Python

```python
from bs4 import BeautifulSoup
from urllib.parse import urlparse
@staticmethod
def extract_content(url, soup):
    sub_titles = urlparse(url).path.split('.')[0].split('/')
    title = '_'.join(sub_titles)[1:]
    tags = ['p', 'span', 'h1', 'h2', 'h3', 'h4', 'h5', 'h6', 'li', 'th', 'td']
    content = ''
    for tag in tags:
        content += '\n' + '\n'.join([txt.text for txt in soup.find(id='content-main').fin
    ConcordiaAboutSpider.write_content_to_file(url, title, content)
```

3.3 Data Preprocessing

In the preprocerssing phase (**preprocessor.py**), we will implement the **Lossy Dictionary Compression** techniques, known as **Normalization**, by removing numbers, blanks, punctuations etc. And then we will tokenize documents extracted by crawler for the next step - indexing.

1. Remove special characters, like punctuations, linefeed/carriage return and other non-alphanumeric characters.

   Python

   ```python
   def remove_special_characters(words):
       '''Removes punctuations, linefeed/carriage return and other non-alphanumeric cha
       special_characters = '!?"#$%&\'()*+,./:;<=>?@[\\]^_`{|}~'
       transtable = str.maketrans('', '', special_characters)
       processed_words = [term.translate(transtable) for term in words]  # remove punct
       useless_words = ['','s','-','--']
       processed_words = [term for term in processed_words if term not in useless_words
       return processed_words
   ```

2. Remove digits

   Python

   ```python
   def remove_numbers(words):
       '''Remove numbers'''
       processed_words = [term for term in words if not term.isdigit()]
       return processed_words
   ```

3. Case Folding

   Python

   ```python
   def case_folding(words):
       '''Case Folding'''
       processed_words = [term.lower() for term in words]
       return processed_words
   ```

4. Remove stopwords

Python

```python
def remove_stopwords(stopwords, words):
    '''Remove stopwords'''
    processed_words = [term for term in words if not term in stopwords]
    return processed_words
```

5. Stemming

Python

```python
def stemming(terms):
    '''Stemming'''
    stemmer = PorterStemmer()
    stemmed_terms = [stemmer.stem(term) for term in terms]
    return stemmed_terms
```

6. Tokenize

Python

```python
def tokenize(file_list):
token_id_pairs = []
for file_index, file in enumerate(file_list):
    print('Processing ' + file_list[file_index] + '...')
    with open(file, 'r') as file_obj:
        data = file_obj.read()
    if data.__len__() == 0:
        continue
    info = ast.literal_eval(data)
    newid = info['title']
    body = info['content']
    tokens = nltk.word_tokenize(body)
    # Preprocess tokens, aka lazy compression
    tokens = remove_special_characters(tokens)
    tokens = remove_numbers(tokens)
    tokens = case_folding(tokens)
    tokens = remove_stopwords(stop_words_30, tokens)
    tokens = stemming(tokens)
    for token in tokens:
        token_id_pairs.append((token, newid))
    print('Tokenization for ' + file_list[file_index] + ' completed...')
print('Tokenization for all files completed...')
return token_id_pairs
```

3.4 Indexing – SPIMI

After preprocessing the data, we will perform the SPIMI algorithm **(spimi.py)** to generate an index of the data crawled by the scrapy.

To generate the final index file, the first thing we have to is to:

1. Iterate terms to create their posting lists.
2. Merge block files.

3. Write them into the final index file.

1. Create posting lists

Python

```python
def add_to_postings_list(postings_list, newid, term):
    """
    :param postings_list: specific postings_list from the disk
    :param newid: id of the document
    :param term: term from a token tuple
    """
    sentiment_value = 0
    term_frequency = 1
    afinn = dict(map(lambda p: (p[0], int(p[1])), [line.split('\t') for line in open("AFI
    for key in afinn.keys():
        if key == term:
            sentiment_value = afinn[key]
            break
    for item in postings_list:
        if item[0] == newid:
            item[1] += 1
            item[3] = (1 + math.log(item[1]))*(math.log(N/len(postings_list)))
            return
    document_frequency = len(postings_list) + 1
    tf_idf_weight = (1 + math.log(term_frequency))*(math.log(N/document_frequency))
    postings_list.append([newid, term_frequency, sentiment_value, tf_idf_weight])

def spimi_invert(token_id_pairs, path='blocks', block_size=float(1)):
    block_count = 0
    disk = {}
    for index, token in enumerate(token_id_pairs):
        token_id_tuple = token_id_pairs[index]

        if token_id_tuple[0] not in disk:
            postings_list = add_to_dictionary(disk, token_id_tuple[0])
        else:
            postings_list = get_postings_list(disk, token_id_tuple[0])

        add_to_postings_list(postings_list, token_id_tuple[1], token_id_tuple[0])
        # check if the size of the current disk is reached to the limit or if reaching la
        if sys.getsizeof(disk)/1024/1024 >= block_size or index + 1 == len(token_id_pairs
            print('Sorting terms in ' + str('BLOCK' + str(block_count + 1)))
            sorted_terms = sort_terms(disk)
            write_block_to_disk(sorted_terms, disk, str('BLOCK' + str(block_count + 1)),
            disk.clear()
            block_count += 1
```

2. Merge & Write into disk

Python

```python
def merge_blocks(path='blocks', output_name='FINAL_INDEX.txt'):
    ...
    for block_index, block_file in enumerate(block_files):
        print('Started reading from ' + block_files[block_index].name + '...')
        current_line = block_files[block_index].readline()
        current_line_splited = current_line.rsplit(':', 1)
        posting_list = ast.literal_eval(current_line_splited[1].strip())
        current_line_dictionary = {current_line_splited[0]: posting_list}
        lines_holder[block_index] = current_line_dictionary
        terms_holder.append(current_line_splited[0])
    while not no_more_lines:
        current_term_to_write = sorted(terms_holder)[0]
        posting_list_to_write = []
        for file_index in lines_holder:
            if current_term_to_write in lines_holder[file_index]:
                occurrence_holder.append(file_index)
        for occurrence in occurrence_holder:
            posting_list_to_write += lines_holder[occurrence][current_term_to_write]
        final_index_file.write(str(current_term_to_write) + ': ' + str(posting_list_to_wr
        terms_holder = list(filter(lambda term: term != current_term_to_write, terms_hold
        for occurrence in occurrence_holder:
            current_line = block_files[occurrence].readline()
            if not current_line == '':
                current_line_splited = current_line.rsplit(':', 1)
                posting_list = ast.literal_eval(current_line_splited[1].strip())
                current_line_dictionary = {current_line_splited[0]: posting_list}
                lines_holder[occurrence] = current_line_dictionary
                terms_holder.append(current_line_splited[0])
            else:
                print('All content from ' + block_files[occurrence].name + ' is merged in
                del lines_holder[occurrence]
                del block_files[occurrence]
        occurrence_holder.clear()
        if len(block_files) == 0 or len(lines_holder) == 0:
            print('All block files are merged into ' + output_name + '...')
            no_more_lines = True
```

3.5 Ranking

The partial order function is defined as

$f(w, s) = \frac{w}{|s|}$

As discussed in section 2.3.4 Ranking function, the arguments **w** and **s** are obtained as follows:

1. $\cos(\vec{q}, \vec{p}) = SIM(\vec{q}, \vec{p}) = \frac{\vec{q} \times \vec{p}}{|\vec{q}||\vec{p}|} = \frac{\sum_{i=1}^{|V|} q_i \times d_i}{\sqrt{\sum_{i=1}^{|V|} q_i^2}\sqrt{\sum_{i=1}^{|V|} d_i^2}}$

2. $s = s_q - s_d$

For the tf-idf cosine based argument, the **function 1** is divided into two portions, namely the **numerator** and the **denominator** portion.

Here is code for the numerator portion (**rank_document.py file**).

Python

```python
def q_d_sum_function(query_dictionary, d_title):
    """
    :param query_dictionary: dictionary containing the terms of a query
    :param d_title: title of a document
    :return: the numerator of the cosine similarity function
    """
    q_d_sum = 0.0
    for key in query_dictionary.keys():
        q_d_sum += query_dictionary[key] * get_document_tf_idf(key, d_title)
    return q_d_sum
```

Here is the code for the denominator portion (**rank_document.py file**).

Python

```python
def q_d_square_sum_function(query_dictionary, d_body, d_title):
    """
    :param query_dictionary: dictionary containing the terms of a query
    :param d_body: body content of a document
    :param d_title: title of a document
    :return: the denominator of the cosine similarity function
    """
    q_square_sum = 0.0
    d_square_sum = 0.0
    for key in query_dictionary.keys():
        q_square_sum += query_dictionary[key] * query_dictionary[key]
    for d_term in nltk.word_tokenize(d_body):
        d_square_sum += get_document_tf_idf(d_term, d_title)*get_document_tf_idf(d_term,
    return math.sqrt(q_square_sum)*math.sqrt(d_square_sum)
```

The overall cosine similarity function is given below (**rank_document.py file**):

Python

```python
def cosine_similarity(query_dictionary, d_body, d_title):
    """
    :param query_dictionary: dictionary containing the terms of a query
    :param d_body: body content of a document
    :param d_title: title of a document
    :return: the cosine similarity measure
    """
    return q_d_sum_function(query_dictionary, d_title) / q_d_square_sum_function(query_di
```

3.6 Query & Rank retrieved documents

The part **(rank_document.py)** mainly focuses on ranking retrieved documents by a partial order discussed above.

Firstly, the program asks the user for a query:

Python

```python
query = input('Please enter a query: ')
```

Secondly, the query is then splitted into tokens and stored in a dictionary (hashmap) which consists of a query token (i.e. query term) and the query token's tf-idf tuple.

Python

```python
def find_q_tf_idf(term, term_frequency):
    """
    :param term: a term in the query
    :param term_frequency: the frequency of the term in the query
    :return: the tf-idf of the term
    """
    if term in final_index_dictionary.keys():
        return (1 + math.log(term_frequency)) * (math.log(N / ((final_index_dictionary[te
    return 0.0
```

Thirdly, after collecting all the pieces required for generating the w and s arguments (i.e. query tf-idf, document tf-idf, query sentiment value, and document sentiment value), we will do the ranking of the retrieved documents.

Python

```python
ranked_document_file = open('RANKED_DOCUMENT_Q1.txt', 'w+')
    sorted_by_value = sorted(document_w_s_weight.items(), key=lambda kv: kv[1], reverse=T
    rank_number = 0
    ranked_document_file.write("Query: " + query + "\n\n")
    for elements in sorted_by_value:
        rank_number += 1
        ranked_document_file.write("Rank: " + str(rank_number) + " url: " + document_url_
```

Finally, a text file is generated with the contents being a list of documents sorted by their respective weight obtained through the partial order function.

4. Analysis & Test

We are planning to test our program from three main parts:

1. Crawling
2. Indexing
3. Ranking

4.1 Crawling Test

1. **Case 1**

   Purpose:

   Check whether the spider can crawle **correct pages** and download the **correct total number of files** into the **right directory** we set, which is ***extracted_files***

   Steps:

1. Open a termianl and go to the project directory.
2. Run the **scrapy crawl about_crawl** command.
3. Check the result.

Results:

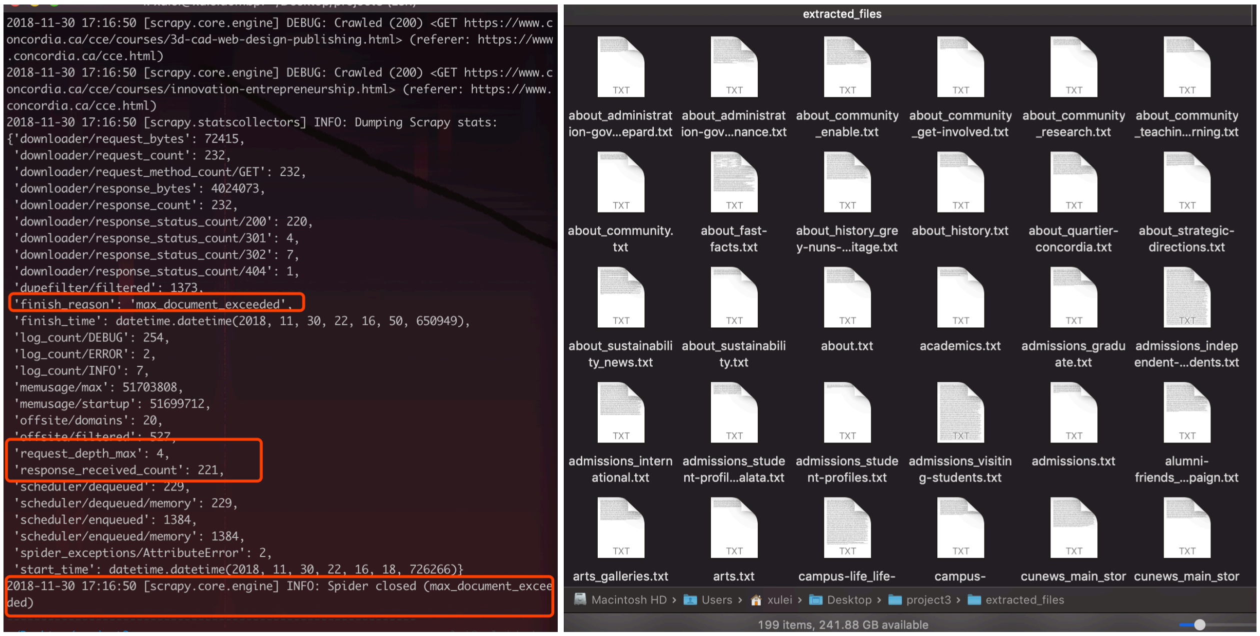


Figure 4.1 Crawling Testing Result

4.2 Indexing Test

1. **Case 1**

   Purpose:

   Check whether the **SPIMI** algorithm works through the following two parts

   1. Block fils
   2. Final index file.

   Steps:

   1. Open a termianl and go to the project directory.
   2. Run the **python3 spimi.py** command.
   3. Check the result.

   Note:

   (**How big is the index?**)

   The size of the final index is **4.6 MB**

   (**How did you define what constitutes a document in our index?**)

   The structure of item in the final index is {term : [[doctitle, tf, sentimentvalue, tfidfweight], [posting#2], [posting#3],…]}

   Results:

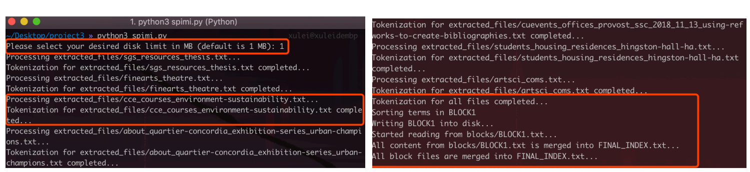
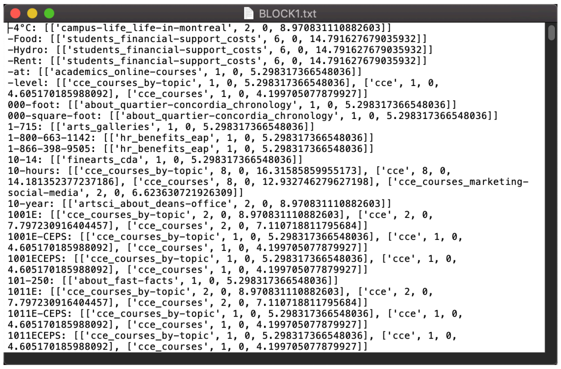


Figure 4.2 Indexing Testing Result - Console



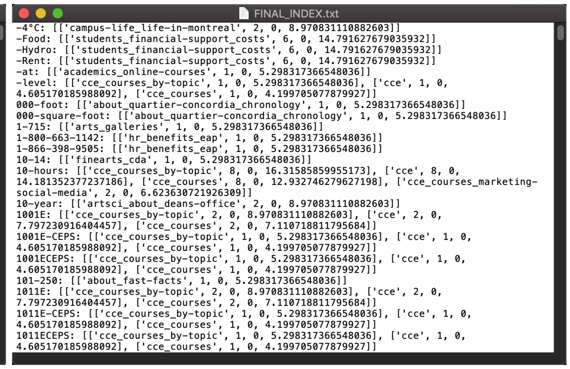


Figure 4.3 Indexing Testing Result - Documents

4.3 Ranking Test

4.3.1 Scenario 01 – Multiple Keyword Query

1. **Case 1**

   Purpose:

   Check whether the program can return the correct result when doing a single keyword (**Positive**) query.

   Steps:

   1. Open a termianl and go to the project directory.
   2. Run the **python3 rank_document.py** command.
   3. Enter positive keywords, like **amazing creativity powers**
   4. Check the result.

   Results:

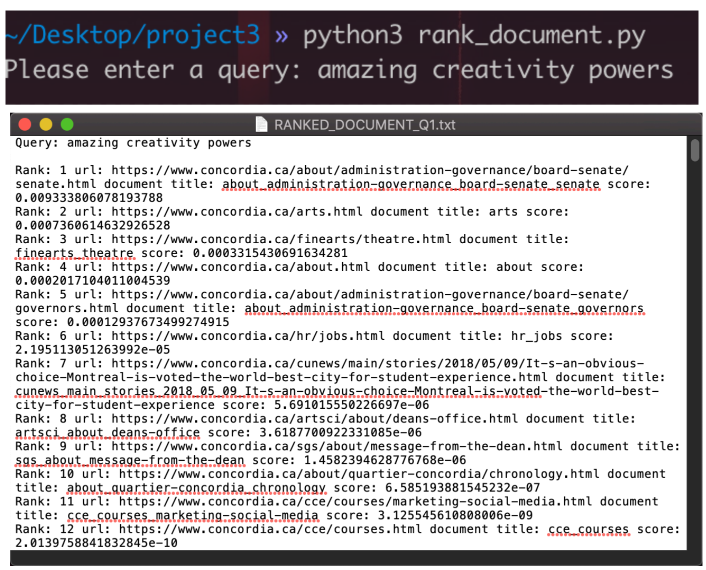


Figure 4.4 Multiple Positive Keywords Query

2. **Case 2**

   Purpose:

   Check whether the program can return the correct result when doing a single keyword (**Neutral**) query.

   Steps:

1. Open a termianl and go to the project directory.
2. Run the **python3 rank_document.py** command.
3. Enter neutral keywords, like **creativity powers**
4. Check the result.

Results:

```
~/Desktop/project3 » python3 rank_document.py
Please enter a query: creativity powers
```

```
RANKED_DOCUMENT_Q2.txt
Query: creativity powers

Rank: 1 url: https://www.concordia.ca/about/administration-governance/board-senate/
senate.html document title: about_administration-governance_board-senate_senate score:
0.0012613251457018633
Rank: 2 url: https://www.concordia.ca/arts.html document title: arts score:
0.00032995858699325813
Rank: 3 url: https://www.concordia.ca/finearts/theatre.html document title:
finearts_theatre score: 0.00015992077453765352
Rank: 4 url: https://www.concordia.ca/about.html document title: about score:
0.00014018348499355635
Rank: 5 url: https://www.concordia.ca/about/administration-governance/board-senate/
governors.html document title: about_administration-governance_board-senate_governors
score: 7.584153430609432e-05
Rank: 6 url: https://www.concordia.ca/hr/jobs.html document title: hr_jobs score:
1.8705122303746772e-05
Rank: 7 url: https://www.concordia.ca/cunews/main/stories/2018/05/09/It-s-an-obvious-
choice-Montreal-is-voted-the-world-best-city-for-student-experience.html document title:
cunews_main_stories_2018_05_09_It-s-an-obvious-choice-Montreal-is-voted-the-world-best-
city-for-student-experience score: 4.879406887595807e-06
Rank: 8 url: https://www.concordia.ca/artsci/about/deans-office.html document title:
artsci_about_deans-office score: 3.3008372489871803e-06
Rank: 9 url: https://www.concordia.ca/sgs/about/message-from-the-dean.html document title:
sgs_about_message-from-the-dean score: 1.360242362377497e-06
Rank: 10 url: https://www.concordia.ca/about/quartier-concordia/chronology.html document
title: about_quartier-concordia_chronology score: 6.09306620528972e-07
Rank: 11 url: https://www.concordia.ca/cce/courses/marketing-social-media.html document
title: cce_courses_marketing-social-media score: 3.0919378987013824e-09
Rank: 12 url: https://www.concordia.ca/cce/courses.html document title: cce_courses score:
2.008950369764693e-10
```

Figure 4.5 Multiple Neutral Keywords Query

3. **Case 3**

Purpose:

Check whether the program can return the correct result when doing a single keyword (**Negative**) query.

Steps:

1. Open a termianl and go to the project directory.
2. Run the **python3 rank_document.py** command.
3. Enter negative keywords, like **disappointing creativity powers**
4. Check the result.

Results:

```
~/Desktop/project3 » python3 rank_document.py
Please enter a query: disappointing creativity powers
```

```
RANKED_DOCUMENT_Q3.txt
Query: disappointing creativity powers

Rank: 1 url: https://www.concordia.ca/about/administration-governance/board-senate/
senate.html document title: about_administration-governance_board-senate_senate score:
0.0007179850829379837
Rank: 2 url: https://www.concordia.ca/arts.html document title: arts score:
0.00024286291936051994
Rank: 3 url: https://www.concordia.ca/finearts/theatre.html document title:
finearts_theatre score: 0.00012029438792655355
Rank: 4 url: https://www.concordia.ca/about.html document title: about score:
0.00011947691409347418
Rank: 5 url: https://www.concordia.ca/about/administration-governance/board-senate/
governors.html document title: about_administration-governance_board-senate_governors
score: 6.0757030245213683e-05
Rank: 6 url: https://www.concordia.ca/hr/jobs.html document title: hr_jobs score:
1.734568249284711e-05
Rank: 7 url: https://www.concordia.ca/cunews/main/stories/2018/05/09/It-s-an-obvious-
choice-Montreal-is-voted-the-world-best-city-for-student-experience.html document title:
cunews_main_stories_2018_05_09_It-s-an-obvious-choice-Montreal-is-voted-the-world-best-
city-for-student-experience score: 4.5372106761609715e-06
Rank: 8 url: https://www.concordia.ca/artsci/about/deans-office.html document title:
artsci_about_deans-office score: 3.157357242994016e-06
Rank: 9 url: https://www.concordia.ca/sgs/about/message-from-the-dean.html document title:
sgs_about_message-from-the-dean score: 1.3149073535480818e-06
Rank: 10 url: https://www.concordia.ca/about/quartier-concordia/chronology.html document
title: about_quartier-concordia_chronology score: 5.867436504761961e-07
Rank: 11 url: https://www.concordia.ca/cce/courses/marketing-social-media.html document
title: cce_courses_marketing-social-media score: 3.075336949830863e-09
Rank: 12 url: https://www.concordia.ca/cce/courses.html document title: cce_courses score:
2.0064446635235218e-10
```

Figure 4.6 Multiple Negative Keywords Query

5. Conclusion

5.1 what is the hardest step?

The hardest step in this project consists of implementing the ranking for the document. This step is dependent on the elements which are implemented in the inverted-index (the term tf-idf weight and the document title in particular) and it requires the handling of the query to obtain the necessary arguments which will be used for the partial order function (i.e. $f(w, s) = \frac{w}{|s|}$). We also need to make thoughtful decisions on how to retrieve the necessary elements from the query and the inverted-index in a way which would minimize the complexity of the function.

5.2 Observations & Experiences

5.2.1 Lei Xu

My contribution mainly foucuses on indexing and testing cases, its data structure, procedures, and its revelant document in the report. In addition, I also am responsible for other parts of the report, like background introduction and principle explaination.

5.2.1.1 What observations do I make during my experiments?

According to my observation, the **SPIMI** algorithm has two advantages:

1. It is faster because there is no sorting required.
2. It saves memory because we keep track of the term a postings list belongs to, so the termIDs of postings need not be stored.

As a result, the blocks that individual calls of SPIMI-INVERT can process are much larger and the index construction process as a whole is more efficient.

In addition to constructing a new dictionary structure for each block and eliminating the expensive sorting step, SPIMI has a third important component: Compression. Both the postings and the dictionary terms can be stored compactly on disk if we employ compression. Compression increases the efficiency of the algorithm further because we can process even larger blocks, and because the individual blocks require less space on disk.

5.2.1.2 What do I learn from my experience?

I've deepened my understanding of indexing methodology, SPIMI algorithm and developed a greater appreciation for the technology and techniques behind indexing. I've also improved my rusty Python skills and used them to find fast ways to calculate things like the intersection of two postings lists, list comprehensions, etc.

5.2.2 Peter Chen

I contributed in making the ranking of the documents for this project. The overall structure and procedures taken to accomplish this task are explained in section 2.3.4 Ranking Function and 3.5 Ranking and 3.6 Query & Rank retrieved documents of this report.

5.2.2.1 What observations do I make during my experiments?

I observed that the different methods I used for retrieving elements from the inverted index could make a huge impact on the speed for generating the ranked documents file. Initially, I tried to retrieve the tf-idf and the df of a term by simply parsing through the entire final merged index file generated by SPIMI. The number of web pages obtained from the crawling process was set to twenty. With the use of this simple method, the time it takes for ranking the documents is around five to ten minutes. This method of retrieving the elements is obviously tedious and has an exponential complexity. The other approach taken is to convert the final merged file into a dictionary. From this approach, the performance increases dramatically. The generation of the ranked document file could be done in less than a minute. I have tested this latter approach with an increased number of web pages (200), and observed that the performance remains constant.

5.2.2.2 What do I learn from my experience?

This project serves as a great review for many of the early materials which have been taught in the course. The ranking of the document task solidified my understanding of the tf-idf cosine based function and length normalization. I was able to make practical use of the cosine similarity formula. This formula brought me to a lot of the essential technical terms in the world of information retrieval. I feel more confident in distinguishing the weight of a document and the weight of a term, document frequency and term frequency. Lastly, aside from gaining more insight about the previously taught subjects, I have also acquired some knowledge about sentimental bias and the impacts it has on document ranking.

5.2.3 Yufeng Ding

Worked on the web crawling part of the project, implemented the web spider using Scrapy library. The spider mainly used to extract the title, the link and the main content of an article into a text file for further process..

The spider starts crawling from a giving starting page, which in our case, is the about page of Concordia university, after extracting its contents, it looks for all other links on the page. Then it will navigate to the first link found and recursively does the same process in a breadth first search (BFS) order until it reaches the maximum number of pages that we defined in our program.

5.2.3.1 What observations do I make during my experiments?

I had observed that there's a tricky part during the process of extracting links from the current page. Sometime the article includes some external references or some social media resources, and those are something that we don't want to crawl. Since in most of the time, they barely contain contents that are related to our school. To solve this situation, I limited our program to extract only links that is from our school's domain name, which is "concordia.ca".

Another observation was that, if we extract the text contents from the whole page without specifying the tag, that would lead our program to exact many duplicated contents, such as all the information from the navbar and footer of the page. That would not make much sense to keep that information, since it's on every single link of our school's website. So, before crawling, we should take a look at the HTML structure of the website and specify those tags that might contain useful information in the program and ignore others that may lead to duplication or confusion.

5.2.3.2 What do I learn from my experience?

By doing this project, I have learned that, how the HTML structure of a website could affect the difficulty of information retrieving process. To make a website crawling-friendly or do search engine optimization (SEO) on a website, we should seriously consider to make a good front-end design to provide a good page structure.