Project 4: Demo

Christopher Di Giacomo

40133600

05 December 2023

COMP479 2232 D

Dr. Sabine Bergler

Table of Contents

Environment/Setup

The project directory is organized into 2 subdirectories:

- /collection : This subdirectory contains the results obtained from crawling + scraping 100 web pages.
- /src: Contains the implementation files for this project.

NOTE: The */collection* subdirectory is created (and updated) with every run of *extraction.py.* You should only run the *extraction.py* file if you wish to crawl/scrape a new set of pages. Otherwise, the *clustering.py* and *sentiment.py* files can be ran independently, as they depend only on the contents of */collection*, and not the execution of *extraction.py.*

NOTE: There may be modules missing when attempting to run the implementation files. Please be sure to verify the import statements at the top of each file and install the required modules/libraries.

First-Run Walkthrough

extraction.py

The following covers the demonstration when running the *extraction.py* file.

The *extraction_pipeline()* function calls every function within the file successively, after first prompting the user via terminal for a batch size (number of links to be crawled to). A batch size of 100 is what was used to generate the results for this project.

```
Please enter a download limit: 100
```

A crawler process is created and is assigned to crawl with the *ConcordiaCrawler* with the specified batch size as the download limit.

```
def extraction_pipeline():
    """
    Main function to start the crawling and writing process.
    """
    limit = int(input('Please enter a download limit: '))
    process = CrawlerProcess()
    process.crawl(ConcordiaCrawler, download_limit=limit)
    process.start()
    write2File()
```

The specific details regarding the structure to the *ConcordiaCrawler* class can be seen in the image below.

```python
class ConcordiaCrawler(scrapy.Spider):
    """
    A Scrapy Spider for crawling the Concordia University website.

    Attributes:
        name (str): Name of the spider.
        allowed_domains (list): List of allowed domains for scraping.
        start_urls (list): List of starting URLs for the spider.
        download_count (int): Counter for tracking number of downloads.
        custom_settings (dict): Custom settings for the spider.
    """

    name = 'concordia_crawler'
    allowed_domains = ['concordia.ca']
    start_urls = ['https://www.concordia.ca']
    download_count = 0   # Counter for tracking number of downloads.

    custom_settings = {
        'ROBOTSTXT_OBEY': True,
        'DOWNLOAD_DELAY': 1,
        'USER-AGENT': 'c_digi@live.concordia.ca'
    }

    def parse(self, response):
```

The *parse()* function is the name functional method of the *ConcordiaCrawler* object. Specific details with regards to the design of the implementation of this function can be found in the report.

```python
    def parse(self, response):
        """
        Parses the response from a web page.

        Args:
            response (TextResponse): The response object to process.

        Returns:
            None
        """
        if self.download_count >= self.download_limit:
            self.logger.info('Reached download limit, stopping spider.')
            self.crawler.engine.close_spider(self, 'download_limit_reached')
            return

        self.logger.info(f'Processing {response.url} with status {response.status}')
        if response.status == 200:
            content_type = response.headers.get('Content-Type', '').decode()
            if 'application/pdf' in content_type or 'video/mp4' in content_type:
                self.logger.info(f"Skipping non-text response at {response.url}")
                return

            if not (response.url in url_text_dict) and self.download_count < self.download_limit:
                soup = BeautifulSoup(response.text, 'html.parser')
                html_tag = soup.find('html')
                if html_tag and 'lang' in html_tag.attrs and html_tag.get('lang', '').startswith('en'):
                    # Focus on the main content of the page
                    main_content = soup.find('main')
                    if main_content:
                        # Remove script or style elements within the main content
                        for script_or_style in main_content.find_all(["script", "style"]):
                            script_or_style.extract()

                        # Get cleaned text from the main content
                        full_text = main_content.get_text(separator=' ', strip=True)
                        url_text_dict[response.url] = full_text
                        self.download_count += 1
                else:
                    self.logger.info(f"Skipping non-English or undefined language page at {response.url}")
                    return

            links = response.css('a::attr(href)').getall()
            for link in links:
                absolute_url = response.urljoin(link)
                if not absolute_url.startswith(("mailto:", "tel:")) and self.is_valid_url(absolute_url):
                    if self.download_count < self.download_limit:
                        yield scrapy.Request(absolute_url, callback=self.parse)
                    else:
                        self.crawler.engine.close_spider(self, 'download_limit_reached')
                        return
```

This *parse()* function uses the *is_valid_url()* function seen below, which deems whether or not a link should be crawled.

```python
def is_valid_url(self, url):
    """
    Checks if a URL is valid and not of an ignored file type.

    Args:
        url (str): The URL to check.

    Returns:
        bool: True if the URL is valid, False otherwise.
    """
    if not (url.startswith("http://") or url.startswith("https://")):
        return False
    parsed_url = urlparse(url)
    _, ext = os.path.splitext(parsed_url.path)
    ignored_extensions = ['.pdf', '.jpg', '.jpeg', '.png', '.gif', '.mp4', '.zip', '.xml', '.svg', '.mp3']
    return ext not in ignored_extensions
```

The below function is what is used by the module to open the */collection* subdirectory and write all of the crawled/scraped documents to file.

```python
def write2File():
    """
    Writes extracted text data to files in a specified directory.
    """
    if not os.path.exists("collection"):
        os.makedirs("collection")

    for i, (url, text) in enumerate(url_text_dict.items()):
        file_name = f"./collection/doc{i}.txt"
        with open(file_name, 'w', encoding='utf-8') as file:
            file.write(text)
```

Running this file gives output that is similar to the following. This is all crawler log output, indicating the state of various asynchronous crawls, along with whether they were successful or not.

```
2023-12-05 01:31:17 [scrapy.core.engine] DEBUG: Crawled (200) <GET https://www.concordia.ca/health/topics/mental-health.html> (referer: https://www.concordia.ca/health/topics/quitting-smoking.ht
2023-12-05 01:31:17 [concordia_crawler] INFO: Processing https://www.concordia.ca/health/topics/mental-health.html with status 200
2023-12-05 01:31:17 [scrapy.spidermiddlewares.offsite] DEBUG: Filtered offsite request to 'toronto.cmha.ca': <GET https://toronto.cmha.ca/understanding-mental-illness/>
2023-12-05 01:31:17 [scrapy.spidermiddlewares.offsite] DEBUG: Filtered offsite request to 'psychcentral.com': <GET http://psychcentral.com/lib/types-of-mental-health-professionals/>
2023-12-05 01:31:17 [scrapy.spidermiddlewares.offsite] DEBUG: Filtered offsite request to 'www.nimh.nih.gov': <GET http://www.nimh.nih.gov/health/topics/mental-health-medications/mental-health-m
2023-12-05 01:31:18 [scrapy.core.engine] DEBUG: Crawled (200) <GET https://www.concordia.ca/cunews/offices/provost/health/topics/quitting-smoking/strengths-skills.html> (referer: https://www.con
2023-12-05 01:31:19 [concordia_crawler] INFO: Processing https://www.concordia.ca/cunews/offices/provost/health/topics/quitting-smoking/strengths-skills.html with status 200
2023-12-05 01:31:19 [scrapy.core.engine] DEBUG: Crawled (200) <GET https://www.concordia.ca/cunews/offices/provost/health/topics/quitting-smoking/coping-strategies.html> (referer: https://www.co
2023-12-05 01:31:19 [concordia_crawler] INFO: Processing https://www.concordia.ca/cunews/offices/provost/health/topics/quitting-smoking/coping-strategies.html with status 200
2023-12-05 01:31:21 [scrapy.core.engine] DEBUG: Crawled (200) <GET https://www.concordia.ca/cunews/offices/vprgs/gradproskills/blogs/2020/01/22/Concordia-Library-hidden-gems.html> (referer: http
.html)
2023-12-05 01:31:21 [concordia_crawler] INFO: Processing https://www.concordia.ca/cunews/offices/vprgs/gradproskills/blogs/2020/01/22/Concordia-Library-hidden-gems.html with status 200
2023-12-05 01:31:21 [scrapy.spidermiddlewares.offsite] DEBUG: Filtered offsite request to 'www.eventbrite.ca': <GET https://www.eventbrite.ca/e/3d-modelling-101-registration-82833833225>
2023-12-05 01:31:21 [scrapy.spidermiddlewares.offsite] DEBUG: Filtered offsite request to 'www.gettyimages.ca': <GET https://www.gettyimages.ca/>
2023-12-05 01:31:21 [scrapy.spidermiddlewares.offsite] DEBUG: Filtered offsite request to 'epe.lac-bac.gc.ca': <GET http://epe.lac-bac.gc.ca/100/205/301/ic/cdc/E/Alphabet.asp>
2023-12-05 01:31:21 [scrapy.spidermiddlewares.offsite] DEBUG: Filtered offsite request to 'siarchives.si.edu': <GET http://siarchives.si.edu/collections>
2023-12-05 01:31:22 [scrapy.core.engine] DEBUG: Crawled (200) <GET https://www.concordia.ca/news/topic.html?topic=topics:research_subjects/technology> (referer: https://www.concordia.ca/news/sto
2023-12-05 01:31:23 [concordia_crawler] INFO: Processing https://www.concordia.ca/news/topic.html?topic=topics:research_subjects/technology with status 200
2023-12-05 01:31:24 [scrapy.core.engine] DEBUG: Crawled (200) <GET https://www.concordia.ca/cunews/main/stories/2022/12/20/concordia-art-hives-offer-creative-community-and-care.html> (referer: h
-for-anti-colonial-solidarity.html)
2023-12-05 01:31:24 [concordia_crawler] INFO: Processing https://www.concordia.ca/cunews/main/stories/2022/12/20/concordia-art-hives-offer-creative-community-and-care.html with status 200
2023-12-05 01:31:24 [scrapy.spidermiddlewares.offsite] DEBUG: Filtered offsite request to 'www.arthives.org': <GET https://www.arthives.org/>
2023-12-05 01:31:24 [scrapy.spidermiddlewares.offsite] DEBUG: Filtered offsite request to 'arthives.org': <GET https://arthives.org/arthives/concordia-university-art-hives-loyola-campus>
2023-12-05 01:31:24 [scrapy.spidermiddlewares.offsite] DEBUG: Filtered offsite request to 'hivecafe.ca': <GET https://hivecafe.ca/>
2023-12-05 01:31:25 [scrapy.core.engine] DEBUG: Crawled (200) <GET https://www.concordia.ca/news/stories/2023/05/26/concordia-students-and-alumni-work-to-indigenize-urban-spaces-through-a-week-l
-a-plan-to-decolonize-and-indigenize-its-curriculum-and-pedagogy.html)
2023-12-05 01:31:25 [concordia_crawler] INFO: Processing https://www.concordia.ca/news/stories/2023/05/26/concordia-students-and-alumni-work-to-indigenize-urban-spaces-through-a-week-long-moose-
2023-12-05 01:31:25 [scrapy.spidermiddlewares.offsite] DEBUG: Filtered offsite request to 'www.buckskinbabes.ca': <GET https://www.buckskinbabes.ca/about>
2023-12-05 01:31:26 [scrapy.core.engine] DEBUG: Crawled (200) <GET https://www.concordia.ca/students/life/all-groups.html> (referer: https://www.concordia.ca/students/life/dean-of-students.html)
2023-12-05 01:31:26 [concordia_crawler] INFO: Processing https://www.concordia.ca/students/life/all-groups.html with status 200
2023-12-05 01:31:27 [scrapy.spidermiddlewares.offsite] DEBUG: Filtered offsite request to 'asacconcordia.com': <GET https://asacconcordia.com>
2023-12-05 01:31:27 [scrapy.spidermiddlewares.offsite] DEBUG: Filtered offsite request to 'mobile.twitter.com': <GET https://mobile.twitter.com/asacconcordia>
2023-12-05 01:31:27 [scrapy.spidermiddlewares.offsite] DEBUG: Filtered offsite request to 'launchlab.ai': <GET http://launchlab.ai/>
2023-12-05 01:31:27 [scrapy.spidermiddlewares.offsite] DEBUG: Filtered offsite request to 'ascendleadership.ca': <GET https://ascendleadership.ca/page/student-affairs/>
2023-12-05 01:31:27 [scrapy.spidermiddlewares.offsite] DEBUG: Filtered offsite request to 'm.facebook.com': <GET https://m.facebook.com/bestbuddiesconcordia/>
2023-12-05 01:31:27 [scrapy.spidermiddlewares.offsite] DEBUG: Filtered offsite request to 'www.casacares.com': <GET https://www.casacares.com/>
2023-12-05 01:31:27 [scrapy.spidermiddlewares.offsite] DEBUG: Filtered offsite request to 'communicationscase.wixsite.com': <GET https://communicationscase.wixsite.com/case-concordia>
2023-12-05 01:31:27 [scrapy.spidermiddlewares.offsite] DEBUG: Filtered offsite request to 'www.ceedconcordia.org': <GET https://www.ceedconcordia.org>
2023-12-05 01:31:27 [scrapy.spidermiddlewares.offsite] DEBUG: Filtered offsite request to 'genderadvocacy.org': <GET https://genderadvocacy.org/>
```

clustering.py

Running the *clustering.py* file triggers the call of the function below via the *main()* method. This pipeline function calls all necessary functions in the file successively.

```python
def cluster_pipeline(k):
    """
    The pipeline function for clustering.

    Args:
        k (int): The number of clusters to use in KMeans.

    Returns:
        tuple: A tuple containing the KMeans object, the matrix, and the original documents.
    """
    docs = load_documents()  # Load the documents.
    matrix, vectorizer = vectorize(docs)  # Vectorize the documents.
    km = cluster(matrix, k)  # Apply clustering.
    topTerms(vectorizer, km, k)  # Print the top terms for each cluster.
    return km, matrix, docs  # Return the KMeans object, matrix, and documents.
```

The first function of interest is the *load_documents()* function, which extracts the content of all of the files saved to */collection* and returns them in a list.

```python
def load_documents():
    """
    Loads documents from a specified directory.

    Returns:
        documents (list): A list containing the content of each document.
    """
    documents = []
    directory = "./collection"  # Path to the directory containing the documents.
    for filename in os.listdir(directory):
        file_path = os.path.join(directory, filename)  # Combine directory and filename to form the full path.
        with open(file_path, 'r', encoding='utf-8') as file:
            documents.append(file.read())  # Read and append the content of each file to the documents list.
    return documents
```

The next function in the pipeline is the *vectorize()* function, which vectorizes the resulting list of documents into a tf-idf matrix with the help of the *TfIdf Vectorizer* library.

```python
def vectorize(documents):
    """
    Converts a collection of raw documents to a matrix of TF-IDF features.

    Args:
        documents (list): A list of documents to be vectorized.

    Returns:
        tuple: A tuple containing the transformed documents and the vectorizer.
    """
    vectorizer = TfidfVectorizer(
        max_df=0.5,  # Maximum document frequency for the given term.
        min_df=5,    # Minimum document frequency for the given term.
        stop_words="english",  # Remove common English stop words.
        token_pattern=r'(?u)\b[A-Za-z][A-Za-z]+\b'  # Regex pattern to identify tokens.
    )
    return vectorizer.fit_transform(documents), vectorizer  # Transform documents and return the matrix and vectorizer.
```

The next function in the pipeline is the *cluster()* function responsible for creating clusters from the resulting matrix using the KMeans library.

```python
def cluster(matrix, num_clusters):
    """
    Applies KMeans clustering to the given matrix.

    Args:
        matrix: The matrix (e.g., TF-IDF) to apply clustering on.
        num_clusters (int): The number of clusters to form.

    Returns:
        KMeans: The fitted KMeans object.
    """
    km = KMeans(n_clusters=num_clusters, n_init=10, random_state=42)  # Initialize KMeans with specified parameters.
    return km.fit(matrix)  # Fit the KMeans model to the data.
```

Lastly the top 20 terms of the clusters are printed to the terminal with the help of the *topTerms()* function seen below.

```python
def topTerms(vectorizer, km, num_clusters):
    """
    Prints the top terms in each cluster.

    Args:
        vectorizer: The vectorizer used to transform the documents.
        km: The fitted KMeans object.
        num_clusters (int): The number of clusters.
    """
    print(f"\n*** TOP TERMS PER CLUSTER FOR k={num_clusters} ***")
    order_centroids = km.cluster_centers_.argsort()[:, ::-1]  # Sort cluster centers by proximity to centroid.
    terms = vectorizer.get_feature_names_out()  # Retrieve the feature names from the vectorizer.

    for i in range(num_clusters):
        print(f"CLUSTER {i}: ", end='')
        for ind in order_centroids[i, :20]:  # Loop through the top 20 terms in each cluster.
            print(f'{terms[ind]} ', end='')  # Print each term.
        print()  # New line after each cluster.
```

The resulting output of running this pipeline for k=3 and k=6 can be seen via the terminal as is shown below.

```
*** 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
```

sentiment.py

The following demonstrates the functioning of the *sentiment.py* module of the project.

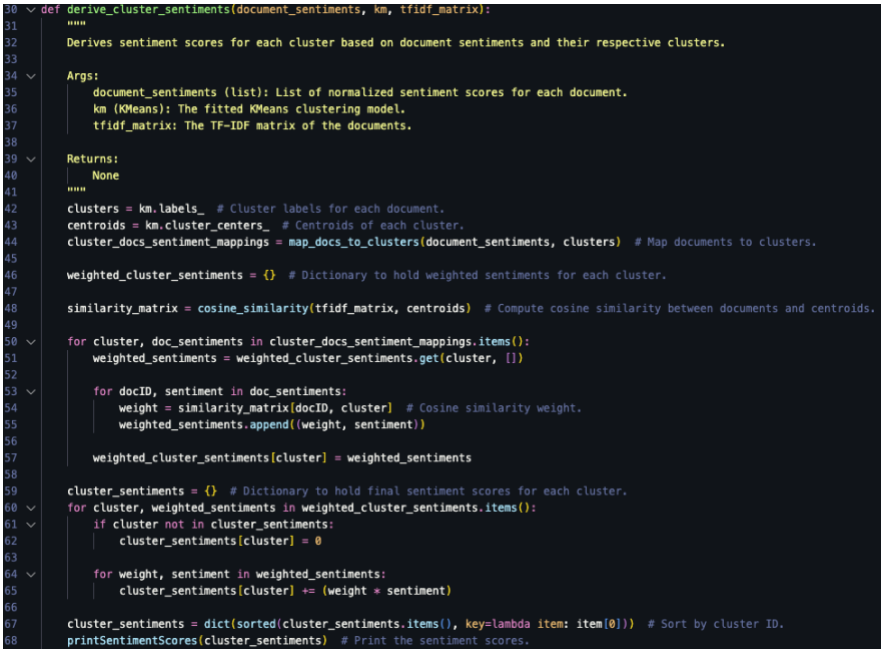
When first ran, the *sentiment_pipeline()* is triggered, which is responsible for calling the necessary functions successively. The first function called is the pipeline of the previous module, in order to retrieve the matrix, KMeans object and the list of documents.

```python
def document_sentiment(documents):
    """
    Calculates normalized sentiment scores for a list of documents using Afinn sentiment analysis.

    Args:
        documents (list): A list of documents (strings).

    Returns:
        list: A list of normalized sentiment scores for each document.
    """
    afinn = Afinn()
    normalized_sentiment_scores = []

    for doc in documents:
        word_count = len(doc.split())  # Count words in the document.
        sentiment_score = afinn.score(doc)  # Get sentiment score using Afinn.
        # Normalize by the log of word count, adding 1 to avoid division by zero.
        normalized_score = sentiment_score / (math.log(word_count + 1) if word_count > 0 else 1)
        normalized_sentiment_scores.append(normalized_score)

    return normalized_sentiment_scores
```

```python
def sentiment_pipeline(k):
    """
    Runs the entire sentiment analysis pipeline for a specified number of clusters.

    Args:
        k (int): Number of clusters to use in the KMeans model.
    """
    km, matrix, documents = cluster_pipeline(k)  # Run the clustering pipeline.
    normalized_doc_sentiments = document_sentiment(documents)  # Get document sentiments.
    derive_cluster_sentiments(normalized_doc_sentiments, km, matrix)  # Derive cluster sentiments.
```

Following this, the pipeline calls the *document_sentiment()* function which iterates through all of the documents in the list, calculating a normalized sentiment score for each, and returning it. More in depth details are covered in the report.

The next function called by this module's pipeline is the *derive_cluster_sentiments()* function, which based on the previously calculated document sentiments, as well as the returned results of the *cluster_pipeline(),* calculates sentiment scores for each cluster. Details regarding the design of this formula are covered in the report.



```python
def derive_cluster_sentiments(document_sentiments, km, tfidf_matrix):
    """
    Derives sentiment scores for each cluster based on document sentiments and their respective clusters.

    Args:
        document_sentiments (list): List of normalized sentiment scores for each document.
        km (KMeans): The fitted KMeans clustering model.
        tfidf_matrix: The TF-IDF matrix of the documents.

    Returns:
        None
    """
    clusters = km.labels_  # Cluster labels for each document.
    centroids = km.cluster_centers_  # Centroids of each cluster.
    cluster_docs_sentiment_mappings = map_docs_to_clusters(document_sentiments, clusters)  # Map documents to clusters.

    weighted_cluster_sentiments = {}  # Dictionary to hold weighted sentiments for each cluster.

    similarity_matrix = cosine_similarity(tfidf_matrix, centroids)  # Compute cosine similarity between documents and centroids.

    for cluster, doc_sentiments in cluster_docs_sentiment_mappings.items():
        weighted_sentiments = weighted_cluster_sentiments.get(cluster, [])

        for docID, sentiment in doc_sentiments:
            weight = similarity_matrix[docID, cluster]  # Cosine similarity weight.
            weighted_sentiments.append((weight, sentiment))

        weighted_cluster_sentiments[cluster] = weighted_sentiments

    cluster_sentiments = {}  # Dictionary to hold final sentiment scores for each cluster.
    for cluster, weighted_sentiments in weighted_cluster_sentiments.items():
        if cluster not in cluster_sentiments:
            cluster_sentiments[cluster] = 0

        for weight, sentiment in weighted_sentiments:
            cluster_sentiments[cluster] += (weight * sentiment)

    cluster_sentiments = dict(sorted(cluster_sentiments.items(), key=lambda item: item[0]))  # Sort by cluster ID.
    printSentimentScores(cluster_sentiments)  # Print the sentiment scores.
```

Within this function, there are function calls to the following two functions. The first which maps the document sentiments back to the documents within each cluster:

```python
def map_docs_to_clusters(document_sentiments, clusters):
    """
    Maps documents to their respective clusters.

    Args:
        document_sentiments (list): List of normalized sentiment scores for each document.
        clusters (list): List of cluster labels for each document.

    Returns:
        dict: A dictionary mapping each cluster to its documents and sentiments.
    """
    cluster_mapping = {}  # Dictionary to map cluster IDs to documents and sentiments.
    for i, cluster_label in enumerate(clusters):
        if cluster_label not in cluster_mapping:
            cluster_mapping[cluster_label] = []
        cluster_mapping[cluster_label].append((i, document_sentiments[i]))
    return cluster_mapping
```

The second which simply outputs the cluster sentiment scores to the terminal:

```python
def printSentimentScores(cluster_sentiments):
    """
    Prints the sentiment scores for each cluster.

    Args:
        cluster_sentiments (dict): A dictionary of cluster IDs and their sentiment scores.
    """
    i = 0  # Cluster index.

    print(f'\n*** CLUSTER SENTIMENT SCORES ***')
    for cluster, score in cluster_sentiments.items():
        print(f'CLUSTER {i}: SENTIMENT SCORE: {score}')
        i += 1
```

The final output for this module can be seen below, in conjunction with the previous module's output since they are both interconnected results.

```
*** 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

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

*** 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

*** 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
```