COMP472 – Project 4 DEMO

Rhina Kim – 40130779

December 12, 2022

This program is composed of 3 python files:

- ***/P4/main.py***: pipeline execution of crawling, index creation, clustering, and sentiment scoring. It also includes the code script for index creation, clustering, and sentiment scoring except crawling part, since crawling part is a code script directly modified from the crawling scripts inside the *spidy* package.
- ***/P4/crawler.py***: the modified version of *spidy* package *crawler*.py. The variables, inputs, and the structure of the code is more adjusted to this project in order to achieve seamless execution.ß
- ***/P4/globals.py***: all the global variables and methods that are shared between *main*.py and *crawler*.py resides here.

The output files:

- ***/P4/crawled_pages.zip***: overview of crawled and saved webpage files. The number of the files inside this zip file depends on the MAX_CRAWL_PAGES configuration which sets the maximum pages to crawl and save.
- ***/P4/crawler_done.txt***: All the links that *spidy* crawler has already visited
- ***/P4/crawler_todo.txt***: All the links that *spidy* has found but yet to be crawled.
- crawler_words.txt (not really necessary for this project, it is just a output generated from original script of *spidy* package)
- ***/P4/logs/***: log files for each execution are stored here. There will be 4 log files: *crawler_log_{*hh-mm-ss}*.txt*, *crawler_error_log_*{hh-mm-ss}*.txt* for any logging information when executing *spidy* crawler, and *scrape_log_*{hh-mm-ss}*.txt* and *scrape_error_log_*{hh-mm-ss}*.txt* for any logging information during extracting texts from downloaded webpages using *bs4*.
- ***/P4/clusters/***: the output of top 20 terms from each cluster and its corresponding *Afinn* score resides here. The files inside this folder are respectively: *k-3.txt* and *k-6.txt* with numeric part being parameter used for KMean clustering algorithm.
- ***/P4/config/***: All the configuration files with different parameters to try out different results. The default is ***/P4/config/concordia.cfg***. */P4/config/blank.cfg* denotes the description of variables used in the configuration files, which was the script brought from original *spidy* package and modified.
- ***/P4/saved/***: accumulates the posts crawled from /*P4/crawler_todo.txt*. Set OVERWRITE = True in the configuration file in order to overwrite the accumulated files for each execution.

Input Prompt (** Important **):

In order to process proper code execution, the user must run from ***/P4/main.py***. At the beginning of the execution, the program will ask to configure the initial variables. It is desirable to edit any configurations directly inside ***/P4/concordia.cfg*** file instead of entering the input one by one in the prompt as there will be a lot of configuration variables to be asked.
To load the configuration file at the beginning of the execution, the program will ask the following set of questions:

```
Do you want to run the crawler? y/n (crawled data is going to be saved in ./saved):
```
{input}

`Should spidy load settings from an available config file? (y/n):` {input}
`… [spidy] [WORKER #0] [INIT] [INPUT]: Config file name:` {input}
For all the yes or no prompt, it is recommended to enter "y" to observe crawl process and to load an existing configuration file. It is also recommended to type "concordia.cfg" or press enter key to import the default configuration suited for this project.

Dependencies Installations

Install virtual environment
To manage Python packages:
- UNIX/MAC OS: `python3 -m pip install --user virtualenv`
- WINDOWS: `py -m pip install --user virtualenv`

Create virtual environment
- UNIX/MAC OS: `python3 -m pip install --user virtualenv`
- WINDOWS: `py -m pip install --user virtualenv`

Activate virtual environment
Before you can start installing or using packages in your virtual environment, you'll need to activate it. Activating a virtual environment will put the virtual environment-specific python and pip executables into your shell's PATH.
- UNIX/MAC OS: `source env/bin/activate`
- WINDOWS: `.\env\Scripts\activate`

Install necessary Python packages
- Install necessary Python packages to execute the project script using **pip**
- List of necessary dependencies file (requirements.txt) are already created for this project
- `pip install -r requirements.txt` to install packaged/dependencies
- All you need to do is install all the packages written in requirements.txt to run the application

Implementations

First, the overall design of this project is as following: Spidy → BeautifulSoup4 → vectorize → KMeans → AFINN
Each step will be described in detail in execution order.

(1) Set crawler configuration option via loading .cfg file or command prompt
Some of the important configuration variables are:

```
THREAD_COUNT = 5
DOMAIN = 'www.concordia.ca'
RESPECT_ROBOTS = True
MAX_CRAWL_PAGES = 100
START = ['https://www.concordia.ca/ginacody.html']
```

- THREAD_COUNT sets the number of threads for speeding up the crawling time
- DOMAIN determines the domain address within which to restrict crawling
- RESPECT_ROBOTS sets whether to obey the rules in crawled domain's robots.txt or not

- MAX_CRAWL_PAGES sets the maximum number of pages to crawl and save
- START sets the starting link for crawling

(2) Execute crawler
Crawler.spidy_main() from the main.py is going to call spidy's main function loop.

/P4/main.py

```
if __name__ == "__main__":
    # Crawl Concordia Websites using Spidy Package
    spidy_main()
```

(3) Look up and fetch Robots Exclusion during crawling
The class RobotsFetcher() fetches the robots.txt file for given URL and returns Robots instance. The content of the robots are then being read through class RobotsIndex() which is one of the pre-existing crawler functions from original *spidy* package.

/P4/crawler.py

```
class RobotsFetcher(object):
    def __init__(self, url, urlparsed):
        self.url = url
        self.urlparsed = urlparsed

    def fetch_robots(self):
        # Replace domain name with robots.txt, Create path for robots.txt
        if self.urlparsed.path == '/':
            robots_url = self.url + '/robots.txt'
        else:
            robots_url = self.url.replace(self.urlparsed.path, '/robots.txt')

        # Fetch the robots.txt file for given URL, and create Robots instance
        robots = Robots.fetch(robots_url)

        return robots

class RobotsIndex(object):
    """
    Thread Safe Robots Index
    """
    def __init__(self, respect_robots, user_agent):
        self.respect_robots = respect_robots
        self.user_agent = user_agent
        self.lock = threading.Lock()
        self.index = {}

    def is_allowed(self, start_url):
        if self.respect_robots:
            return self._lookup(start_url)
```

```
        else:
            return True

    def size(self):
        ...

    def _lookup(self, url):
        hostname = urllib.parse.urlparse(url).hostname
        if hostname not in self.index.keys():
            with self.lock:
                # check again to be sure
                if hostname not in self.index.keys():
                    self._remember(url)

        return self.index[hostname].allowed(url)

     def _remember(self, url):
        # Parse the link given
        urlparsed = urllib.parse.urlparse(url)
        # Fetch robots.txt file for given URL
        robots = RobotsFetcher(url, urlparsed).fetch_robots()
        # Create Robots.Checker instance for the specified user agent
        checker = robots.agent(self.user_agent)
        # Adds the checker to the index using the hostname as the key
        # Which allows Spidy to quickly look up the rules for a given hostname
without having to fetch the robots.txt file every time.
        self.index[urlparsed.hostname] = checker

# Spawn threads here
robots_index = RobotsIndex(RESPECT_ROBOTS, HEADER['User-Agent'])
spawn_threads(robots)   # Starts the threat activity
```

(4) Extract text from saved pages

Crawled pages are saved into ./saved folder after crawling. The program will go through files in ./saved and then extract only html files and <body> tag / text part from those html files. The text retrieval is done using some of the methods from *BeautifulSoup4*.

/P4/main.py

```
# Extract text from html files with BeautifulSoup
def get_text_from_pages(folder):
    extracted_texts = []

    # Go through files in "/saved"
    for subdir, dirs, files in os.walk(folder):
        for filename in files:
            # only extracting text from html files
            if filename.endswith((".html")):
```

```
                filename = os.path.join(folder, filename)
                if path.exists(filename):
                    # need to force encoding
                    file = open(filename, 'r', encoding="utf8")
                    soup = BeautifulSoup(file.read(), 'lxml')
                    extracted_texts.append(soup.get_text())
    return extracted_texts
```

(5) Vectorize and Apply K-means clustering algorithm

The input is a list of texts from the step (4), and k numbers of clusters to create. The program first uses a *sklearn.feature_extraction.text.**TfidfVectorizer()*** to vectorize the input texts, which converts them into numerical representations that can be used as input to the k-means algorithm. The K-means algorithm is then applied to the vectorized input data which generates resulting clusters.

/P4/main.py

```
def cluster_collection(extracted_texts, k):
    cluster_terms = []

    # Vectorize
    vectorizer = TfidfVectorizer(stop_words='english')
    X = vectorizer.fit_transform(extracted_texts)

    # Apply k-means
    kmeans = KMeans(n_clusters=k, init='k-means++', max_iter=100, n_init=1)
    kmeans.fit(X)

    cluster_ids, cluster_sizes = np.unique(kmeans.labels_, return_counts=True)
    print(f"Number of elements asigned to each cluster: {cluster_sizes}\n")

    # Get cluster terms
    order_centroids = kmeans.cluster_centers_.argsort()[:, ::-1]
    terms = vectorizer.get_feature_names_out()
    for i in range(k):
        current_words = ""
        for ind in order_centroids[i, :]:
            current_words += terms[ind] + ' '
        cluster_terms.append(current_words)

    return cluster_terms
```

(6) Retrieve top 20 cluster terms

The resulting clusters obtained from step (5) are used to acquire top 20 terms for each cluster. These top terms are returned as list of strings in order to later insert into Afinn() function as an input.

/P4/main.py

```
# Get top cluster terms
def get_top_cluster_terms(cluster_terms):
    cluster_top_terms = []
    k = len(cluster_terms)
    for i in range(k):
        current_words = ""
        lst_terms = cluster_terms[i].split(" ")
        for term in lst_terms[:NUMBER_OF_INDEX_TERMS]: # index slicing
            current_words += term + ' '
        cluster_top_terms.append(current_words)

    return cluster_top_terms
```

The variable NUMBER_OF_INDEX_TERMS is set to 20 which is used to retrieve top cluster terms.

As an output example, the resulting top cluster terms will look like (for k=3):

```
cluster_top_terms = [
       [school, concordia, science, services, academic, …, schools, colleges, class],
       [concordia, event, school, calendar, …, news, research, students, computer],
       [concordia, school, calendar, …, schools, colleges, news, academic, current]
]
```

(7) Compute *Afinn* sentiment score

To perform sentiment analysis on the clusters, *AFINN* lexicon and the *afin* 0.1 script is used. *Afinn* scores for each cluster are returned as list of scores for the output step in (8). The function takes a single argument, *cluster_terms*, which is expected to be a list of strings from step (6). It iterates over the list, calculates the AFINN score for each string using the *Afinn().score()* method, and appends the result to a new list called *afinn_score_cluster_terms*. Finally, it returns the *afinn_score_cluster_terms* list.

/P4/main.py

```
def compute_afinn_score(cluster_terms):
    afinn_score_cluster_terms = []
    k = len(cluster_terms)

    for i in range(k):
        # Calulate AFINN score of each term
        afinn_score = Afinn().score(cluster_terms[i])
        num_words = len(cluster_terms[i].split())

        afinn_score_cluster_terms.append(afinn_score)

    return afinn_score_cluster_terms
```

Here is the ***main()*** function from */P4/main.py*:

```
if __name__ == "__main__":
    # Crawl Concordia Websites using Spidy Package
    ans = input('Do you want to run the crawler? y/n (crawled data is going to be
saved in ./saved)')
```

```
    if (ans in ['y', 'Y', 'yes', 'Yes', 'YES']):
        write_log(LOG_FILE, 'INIT', 'Creating variables...')
        spidy_main()

    # After execution, all the downloaded pages are in ./saved/
    print("\n------ EXCTRACTING TEXT WITH BeutifulSoup ------")
    extracted_texts = get_text_from_pages(CRAWL_FOLDER)

    print("\n------ CLUSTERING AND SCORING THE EXTRACTED TEXT WITH K=3 ------")
    cluster_terms_k3 = cluster_collection(extracted_texts, 3)
    top_cluster_terms_k3 = get_top_cluster_terms(cluster_terms_k3)
    afinn_score_k3 = compute_afinn_score(top_cluster_terms_k3)
    result = list(zip(top_cluster_terms_k3, afinn_score_k3))
    write_clusters_scores_to_file(result, 3)

    print("\n------ CLUSTERING AND SCORING THE EXTRACTED TEXT WITH K=6 ------")
    cluster_terms_k6 = cluster_collection(extracted_texts, 6)
    top_cluster_terms_k6 = get_top_cluster_terms(cluster_terms_k6)
    afinn_score_k6 = compute_afinn_score(top_cluster_terms_k6)
    result = list(zip(top_cluster_terms_k6, afinn_score_k6))
    write_clusters_scores_to_file(result, 6)
```

This project has experimented clustering and sentiment scoring with two cluster values: k=3 and k=6. The result output are under */clusters/k-3.txt* and */clusters/k-6.txt* respectively.