Linh Tran (Skyler)

Information Retrieval – 5337

Term Project – Phase 1

1. Implement your crawler according to requirements.

a) Describe the key architecture of your web crawler.

Answer:

- I used these libraries: “requests,” “BeautifulSoup”
- The rest of the code is my personal work and not borrowed from anyone else online or in class.
- Procedure in my code: (in order)
    - Create site maps (from robots.txt) to construct ‘disallow’, ‘allow’ list of urls, and keeping track of non-text urls.
    - Send request to a url and get raw content with HTML markup (from ‘allow’ url list)
        - Check for status code of each request. If not ‘200’, push to a list of ‘broken links” and move on.
    - Parse HTML from raw content, make word vectors and other essential variables that we need to keep track of.
        - Check for meta tag of noindex. If present, move on; if not present, continue to make word vector

b) Identify and describe your major data structures. Did you use lists, arrays, etc?

Answer:

- I use ‘list’ mostly, ‘dictionary’ for some variables and look-up table in the format of {key: (value1, value2)}

c) Describe how you implemented “politeness” in your crawler.

Answer:

- I hit separate webpage every 3 or 4 seconds plus the time for my crawler to process one webpage after hitting it.

d) Describe in detail how the above is implemented in your code. [25 points]

Answer:

my_crawler = WebCrawl()

- Class WebCrawl
    - functions: (name_of_function: *what it does*)
        - create_site_map: *get allow/disallow from robots.txt, and keep track of non-text urls in non_text_url variable*
        - isNoIndexPage: *check for meta tag noindex*
        - parse_html_and_make_word_vector: *parse raw content and make word vector*
        - getTitle: *get titles of all urls*
        - isLocalRobotsTxt: *check for local robots.txt*
        - getRawContentFromTextFile: *get raw content from url*
        - crawl: *run the web crawler*

    - variables: (local variable: type – *description*)
        - N: number - *limit of number of documents needed to parse*
        - domain: list – *domain to crawl*
        - raw_content: list – *raw content with HTML markup*
        - content_dict: dict – *dictionary in the format of {url: raw content} for duplicate detection*
        - disallow: list – *disallow list to crawl*
        - allow: list – *allow list to crawl*
        - broken_links: list – *broken links, urls that don't give status code of '200'*
        - non_text_urls: list – *urls that don't have extension of text file (specified in requirements)*
        - titles: dict – *dictionary of text inside TITLE tag of all pages (in both 'allow' and 'disallow') in the format of {url: title}*
        - num_of_documents_indexed: number – *number of urls with extension of text file that give status code of '200'*
        - word_indexed: number – *number of words indexed*
        - url: string – *keeping track of current url*

2. Use your crawler to list the URL and <TITLE> of all pages in the test data and report all links going out of the test data (i.e. items you must not crawl). [10 points]

CS7337: determine if the page has a html meta tag of noindex. If so, do not index this page, but include it in your listing.

Answer:

- my_crawler.titles gives back the dictionary of titles and their urls.
- my_crawler.disallow gives back the disallow directories and urls.
- I accidentally implemented checking for noindex page. My crawler doesn't index that page but it is in my 'allow' list of urls.

3. Implement exact duplicate content detection. List the URLs of pages that contain already seen content. Do not index the duplicate pages. [10 points]

Answer: I will construct a dictionary of raw_content of indexed urls, and look up every time I crawl a webpage

```
if self.raw_content not in self.content_dict.values():
    self.content_dict[url] = self.raw_content
else: #if raw content exists in indexed pages, move on
    continue
```

4. Use your crawler to list all broken links within the test data. [10 points]

Answer:

```
my_crawler = WebCrawl(domain, N)
my_crawler.broken_links # broken links
```

5. List the URLs of non-text files that are referenced in the test data. [10 points]

Answer:

```
my_crawler.non_text_urls
```

6. Your crawler must save each word and position from each page of type (.txt, .htm, .html, .php). Make sure that you do not save HTML markup. Unless you provide a different definition, you can assume a word is a string of non-space character(s), beginning with an alphabetic character. If a token does not begin with an alphabetic character, remove those until it is alphabetic or null. You may decide if it can contain special characters, but the last character of a word is either alphabetic or numeric. Perform case insensitive matching. Start with an empty dictionary and add words as they are encountered. In this process, give each page a unique document ID. Further, the output of this step will generate a term-document frequency matrix. Your program may generate the data to be further processed in a spreadsheet (Excel or equivalent).

=> Output will display when you run my .py program

a) What is your definition of "word"?

Answer: My definition is the same as default: *a string of non-space character.*

b) How many documents are indexed?

Answer:

my_crawler.num_of_document_indexed

c) How many words are indexed?

Answer:

my_crawler.document_matrix.keys()

d) Generate the term-document frequency matrix. [25 points]

Answer:

my_crawler.document_matrix

In this variable, my data structure is a look up table:

{word: [(word count in url1, url1), (word count in url2, url2) …]}

7. Report the 20 most common words with its document frequency. [10 points]

Answer:

- I wasn't able to get the word document frequency but I was able to get the top 20 most common words

```
word_count = dict()


for word in my_crawler.document_matrix.keys():
    total_word_count = 0
    for index in range(len(my_crawler.document_matrix[word])):
        total_word_count = total_word_count + my_crawler.document_matrix[word][index][0]
    word_count[word] = total_word_count


sorted_word_count = dict(sorted(word_count.items(), key=operator.itemgetter(1), 
reverse=True))
```

```
for i in range(20):
    print(sorted_word_count[i])
```