CS121-Project2-Web Crawler

05/03/2024

Group member:

…,...

Jixuan Luo, 31754916

Chengjun Wu, 39263968

Ron Luo, 57585853

…,...

Global variables:

not_allowed: the url that we know and not allowed by robots.txt

visited_page: the number of the page we have visited and crawl

visited: the set for all the page we have visited in order to avoid repetitive crawl.

longest_number: count the number of words in the longest page.

longest_url: the url of the longest page.

WordCount: the dictionary to count the word frequency.

domain: the dictionary to count the domain frequency.

depth: the dictionary to detect a trap.(more specific explanation below in extract_next_links)

finger_print: the dictionary to store all the fingerprint we have and check similarity.

stop_words: words we do not want to count.

Functions:

scraper(url, resp):

1. Try to extract_next_links and return it
2. Exception process: if we encounter some unknown Exception, we just skip this url.

printall():

1. Print the url with the longest words
2. Print the number of the words in that url
3. In the result.txt, we write the total number of pages we have visited, the longest url and the words number of it, the sorted words_count dictionary and the domain dictionary.
4. We write all the urls that is not allowed by robots.txt into not_allowed.txt
5. We write all the urls we visited in the visited.txt

extract_next_links(url, resp):

1. Check the status code, if it is not 200, the result of this url will be empty list [].
2. Check the depth. Depth works this way: if we get url B from url A, and we got url C from url B, then depth of A = 0, B = 1, C = 2. If this url is obtained by a certain amount of crawls(we set it to be 20), then we consider it as a trap. If it is a trap, we should get nothing with it, making it return an empty list []. We write this url to our ING_trap.txt.
3. Then, we have parsed url = urlparse(resp.url) where resp.url is the actual url we are crawling.

4. Then, as we have seen, different url have different encoding ways. As a result, we need to find the way to decode it. We can get how it encode by getting Content-Type in the content of the url and we set our default decode to utf-8. If it has a specific decode, we use it. If not, we use the default utf-8.
5. Then we use the library BeautifulSoup3 to parse the html in order to get all the urls and text in the html.
6. And then we want to check the length of html. We first try to directly get the content length from the title Content-Length. If the html does not have Content-Length, we just set it to the length of the text from soup.get_text(). If the length is 0, which means it is an empty file, we should just return the empty list []. We write this url to our ING_empty_file.txt. If it is too big so that the length is bigger than a number(we set it to be 1000000), it is too large to crawl and we just return the empty list[]. We write this url to our ING_too_large_file.txt.
7. If it has a proper length, we begin to check the similarity. We use fingerprint similarity check. First, we get all the text from the html and for every 3 words, we combine them and do hash. If the hash % 4 is 0, we use it as our fingerprint and add it the fingerprint of this url. Then, we loop through all the fingerprint we've already had and find the intersection/union. If intersection/union is bigger than a threshold(we set it to be 0.95), then we recognize this url is similar and just return empty list []. We write this url to our ING_similar.txt file. If it is not similar, we add the fingerprint of this url to the fingerprint list in order to make comparison with the following urls.

8. We get all the links by using the bs3 library and we initialize our return list to a set to avoid repetition.
9. For every word in the text, we update our word count dictionary.
10. We update the longest page by comparing the number of words in this url.
11. We update the domain dictionary to count the number of domains.
12. We update the visited_page, which is the number of total pages we have visited.
13. We loop through every url in the page, we do such a few things to get the right url based on resp.url and url(AI tutor told us to do):

    a. We use urllib.parse.urljoin to get the absolute url by adding the base url and relative url.

    b. We make the scheme lower case and delete the default:80 and :443.

    c. We use posixpath to normalize the path.

    d. We use unquote, urlencode, parse_qsl to sort the query.

    e. We get our new path by using urlunparse combining all those updated part together with defragment.
14. For every url, we check whether it has been already visited and whether is_valid. If it is not visited and is valid, we add it to the return_list(which is a set but finally will be a list) and add it to the visited, which will help us avoid the future repetition. Also, the depth of new_url will be the depth of current url + 1.
15. Finally return the list of all the url found in this page.

is_valid(url):

1. We check whether it has been banned first by checking not_allowed. If it is not allowed, return False.
2. We check the scheme, if it is neither https nor http, return False.
3. if the domain does not end up with ics.uci.edu, stat.uci.edu, informatics.uci.edu or cs.uci.edu, return false.
4. No calendar and stayconnected, if found, return False.
5. If the length of url is too long(we set the threshold to be 300), return False.
6. Check the robots.txt by using urllib.robotparser. We create a robotparser first and set url and check whether we are allowed to crawl it. If not, we add the url to not_allowed and return False.
7. Return not match all these symbol(given by the original code).
8. Exception process: if we encounter a ssl urlopen error in robots.txt part, we just return False to be polite.