PubText: A PubMed Text Search Tool Documentation

By Damee Moon, Max Anderson, Keerti Chalasani, Andy Lu

Contents

1) An overview of the function of the code (i.e., what it does and what it can be used for).
2) Documentation of how the software is implemented with sufficient detail so that others can have a basic understanding of your code for future extension or any further improvement.
3) Documentation of the usage of the software including either documentation of usages of APIs or detailed instructions on how to install and run the software, whichever is applicable.
4) Example of a search case and exploration of the webapp interface
5) Brief description of the contribution of each team member in case of a multi-person team.

Overview function of the code

The main function for our project is to provide users with the ability to query a scientific database with a keyword and get resulting a histogram with terms that appear with the most frequency in relation with the query. The database in question is the National Center for Biotechnology Information (NCBI) PubMed database which is a database filled with research articles from various field, such as biology, medicine, chemistry, etc.. This functionality can allow for users to find relations between keywords and terms that would otherwise go unnoticed. An example would be to search with the term “antibiotics resistance” and analyze the resulting histogram to see if there are specific terms like names of antibiotics or types of bacteria that come up frequently. This tool can be used by researchers, students, and medical professionals to provide an avenue of research into topic relations in the a field of interest.

Implementation

The structure of our code is:
Frontend:

- App.py - the code for the webapp

Backend/Scraping:

- extract.py - code that ties scraper.py and count.py together
- Scraper.py - code that contains scraping logic for ncbi articles
- Count.py - code used for tokenizing word, counting word for histogram

This documentation will go through each file one by one for implementation details.

App.py

App.py contains a html layout that determines the look of our webpage.

```
colors = {
    'background': '#111111',
    'text': '#7FDBFF'
}

app.layout = html.Div(style={'backgroundColor': colors['background']}, children=[
    html.H1(children='CS410 Project PubMed Search ',
    style={
            'textAlign': 'center',
            'color': colors['text']
        }
    ),

    html.P([
        'Search Phrase: ',
        dcc.Input(value='', type='text',id='search-text-id',

            style={
                'textAlign': 'center',
                'width': '80%',
                'margin': 20
            }
        )], id = 'search-text-p', style={

            'color': colors['text']

        } ),
```

The search bar callback is written by the function below which calls a function from extract.py as the main point of entry.

```
@app.callback(Output('intermediate-value', 'children'),
              [Input('search-button', 'n_clicks')],
              [State('search-text-id', 'value'),
               State('no-docs-id', 'value')])
def update_cache(n_clicks, search_text, no_docs):

        with open('config.toml', 'r') as fin:
            cfg_d = pytoml.load(fin)

        extract = ext.Extract(cfg_d)

        dff = extract.extract_abstracts (search_text, int(no_docs))
        return dff.to_json(date_format='iso', orient='split')
```

Then update_graph is the function that takes the json data returned by update_cache and display the results on the graph.

```
@app.callback(Output('search-output-graphic', 'figure'),
              [Input('intermediate-value', 'children')])
def update_graph(json_data):
```

This function allows for users to click on the graph for more information.

```
@app.callback(
    Output(component_id='abstract-html', component_property='children'),
    [Input('search-output-graphic', 'clickData')],
    [State('intermediate-value', 'children')])
def update_output_div(clickData, json_data):
```

Extract.py

```
def extract_abstracts (self,term, no_of_docs):
    base_url = self.config_d['base-url']
    pubmed_url = self.config_d['pubmed-url']
    page_limit=max(int(no_of_docs/20),1)

    extract_file =  self.config_d['extract-file']
    original_term = term
    term = term.replace(' ', '+')

    scraper = sc.Scraper(base_url)
    #ncbi_url = pubmed_url+ term + '&ncbi_sortorder=relavance'
    ncbi_url = pubmed_url+ term

    print (ncbi_url)
    print (page_limit)

    article_links= scraper.scrape_ncbi_articles(ncbi_url, page_limit)
    article_abstracts = {}
    article_words={}
    count = cnt.Count(self.config_d)
    link_l = []
    abstract_l = []
    term_l = []
    frequency_l = []


    for link in article_links:
        abstract=scraper.scrape_study_page(link)
        article_abstracts[link]=  abstract
        unigrams = count.analyze(abstract)
        term_unigrams = count.analyze(original_term)
        term_count = 0;
        for term in term_unigrams:
            if term in unigrams:
                term_count += unigrams[term]
        print (len(unigrams))
        print (term_count)
        article_words [link] = len(unigrams)
        link_l.append(link)
        abstract_l.append(abstract)
        term_l.append(len(unigrams))
        frequency_l.append(term_count)
    dict = {}
    dict['links'] = link_l
    dict['abstracts'] = abstract_l
    dict['terms'] =  term_l
    dict['frequency'] =  frequency_l
    df = pd.DataFrame(dict)
    return df
```

The main function for extract.py is extract_abstract which collects the list of articles and then scrape them by calling methods from scrapper.py. This function returns a dataframe which app.py reads to display necessary information.

Scraper.py

This file contains functions for scraping the article links and article abstract.

```
class Scraper:
    def __init__(self, base_url):
        options = Options()
        options.headless = True
        self.browser = webdriver.Chrome('./chromedriver', options=options)
        self.base_url = base_url

    def get_js_soup(self, url):
        self.browser.get(url)
        res_html = self.browser.execute_script('return document.body.innerHTML')
        soup = BeautifulSoup(res_html, 'html.parser')
        return soup
    def remove_script(self, soup):
        for script in soup(["script", "style"]):
            script.decompose()
        return soup

    def is_absolute(self, url):
        return bool(urlparse(url).netloc)

    def scrape_ncbi_articles(self, ncbi_url, sz):
        article_links = []

        self.browser.get(ncbi_url)
        index = 0
        while index < sz:
            index += 1
            print ('-'*20,'Scraping directory page','-'*20)
            soup = BeautifulSoup(self.browser.page_source, 'html.parser')
            for rprt in soup.find_all('div', class_='rprt'):
                for link_holder in rprt.find_all('div', class_='rslt'):
                    if link_holder is not None and link_holder.find('a') is not None:
                        rel_link = link_holder.find('a')['href'] #get url
                        #url returned is relative, so we need to add base url
                        if not bool(urlparse(rel_link).netloc):
                            rel_link = self.base_url+rel_link
                        article_links.append(rel_link)
                        print(rel_link)

            try:
                 next_link = self.browser.find_element_by_partial_link_text('Next')
                 if next_link != 'None':
                     next_link.click()
                 else:
                     break
            except:
                break


        return article_links
```

It uses libraries like beautifulsoup to do the web scraping. In regards to the chrome driver, there may be compatibility issues depending on the operating system this code is being run on. If compatibility issues

arise, testers can try to download the version of chromedriver most compatible with their operating system. Sometimes issues may arise where chromedriver is not found, in that case replace the 'chromedriver' with the full path.

The function scrape_ncbi_articles is returning a list of article links for which another function in the same python file will scrape for the actual abstract.

Count.py

```
class Count:
   def __init__(self, cfg_d):
       self.cfg_d = cfg_d



   def analyze(self):
       # Read the datafile from config
       datafile = (self.cfg_d['extract-file'])
       f = open(datafile, "r")

       # read file content into content
       content = f.read()

       # construct a metapy doc
       doc = metapy.index.Document()
       doc.content(content)

       # tokenizers. Use ICUTokenizer
       tok = metapy.analyzers.ICUTokenizer(suppress_tags=True)

       # Filter stop words using the stopwords.txt
       stop_words_file=self.cfg_d['stop-words']
       tok = metapy.analyzers.ListFilter(tok, stop_words_file, metapy.analyzers.ListFi

       #Removes short words. lower cases it
       #lowercases, removes words with less than 2 and more than 25  characters
       tok = metapy.analyzers.LengthFilter (tok, min=2,max=25)
       tok = metapy.analyzers.LowercaseFilter(tok)

       #performs stemming.
       tok = metapy.analyzers.Porter2Filter(tok)

       ana = metapy.analyzers.NGramWordAnalyzer(1, tok)
       unigrams = ana.analyze(doc)

       return unigrams
```

This python file is mainly used for analyzing the words. The analyze function applies various filters like tokenizer, stopwords, stemming to return the unigrams in the desired form for this application.

Usage

To use this piece of software, git clone the directory. Then run “python3 app.py”. An optional argument is the name of a customized stopword file: “python3 app.py file.txt”. This custom stopword file and the included lemur stopword file will be combined together to create a stopword file that will be used for analysis. The default search upon running the program uses our included antibiotic resistance stopwords, and every subsequent search uses the user’s custom stopwords. If no custom stopword file is passed, then the default stopword file is used (a combination of lemur stopwords and our curated stopwords for antibiotic resistance).

There will be various libraries that you are missing. Install those modules using “pip install” until the application runs. Navigate to the localhost url in your web browser. http://127.0.0.1:8050/

```
andy@andy-XPS-15-9550:~/Documents/cs410/Course_project$ python3 app.py
Running on http://127.0.0.1:8050/
Debugger PIN: 031-532-339
 * Serving Flask app "app" (lazy loading)
 * Environment: production
   WARNING: Do not use the development server in a production environment.
   Use a production WSGI server instead.
 * Debug mode: on
Running on http://127.0.0.1:8050/
Debugger PIN: 435-577-544
```

Troubleshooting Scenarios

- If you don’t have the necessary libraries, use “pip install” or “sudo pip install” to install all the libraries
- Different operating systems display file systems differently. If issues arise where “chromedriver” cannot be located, user would have to edit the init function in scraper.py. Change ‘chromedriver’ to the path of your chromedriver or ‘./chromedriver.’ The file we committed runs on Mac and Windows, Linux might require a tweek.

```
def __init__(self, base_url):
    options = Options()
    options.headless = True
    self.browser = webdriver.Chrome('chromedriver', options=options)
```

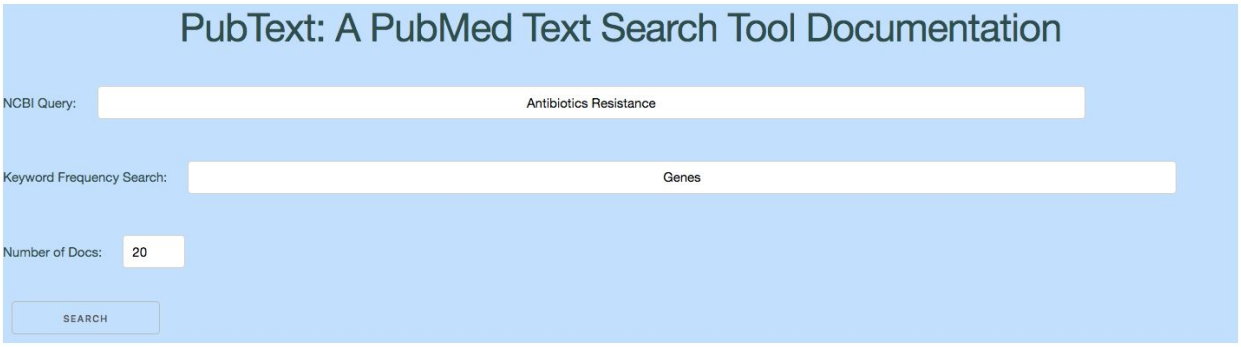
- The version of the chromedriver might be incompatible with your operating system. Download the appropriate version and replace your local one from http://chromedriver.chromium.org/

Example

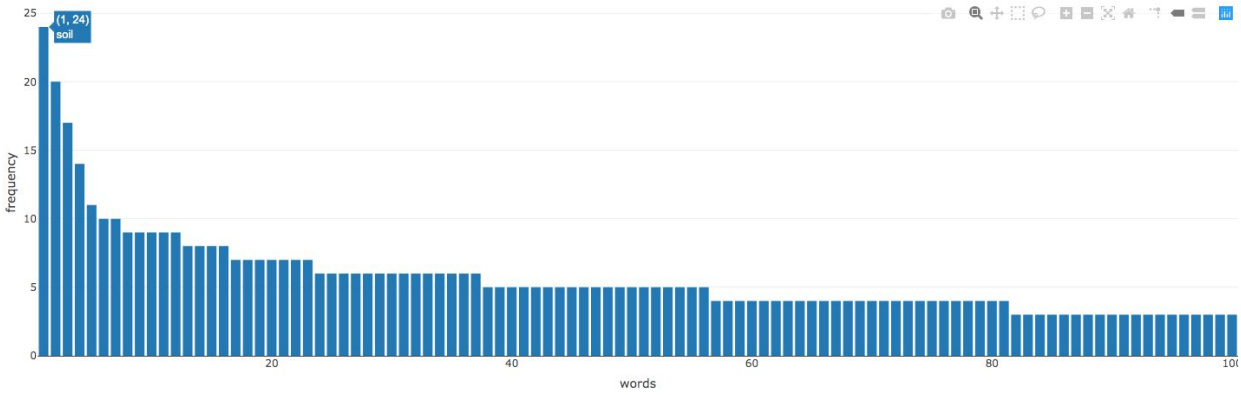
Note: The results for the following examples are subject to change due to the ever changing nature of NCBI.

The default search upon initialization of the search tool utilizes the following parameters:
NCBI Query: Antibiotics Resistance

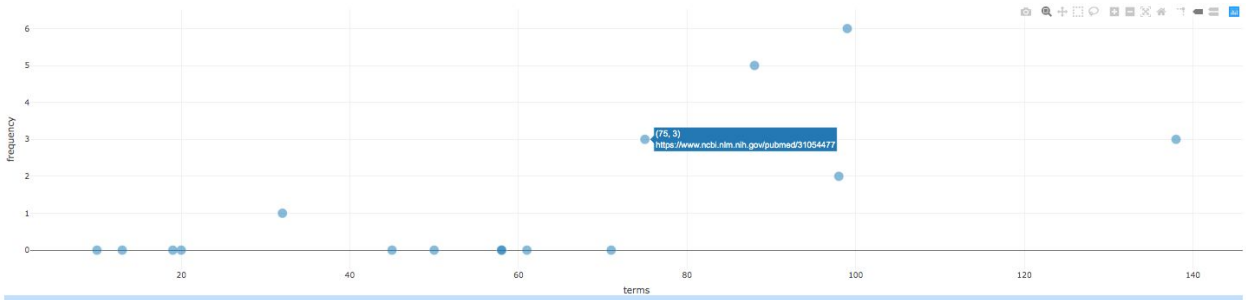
Keyword Frequency Search: Genes
Number of Documents: 20



The first plot is a histogram of the top 100 most frequently present words in the corpus ranked from most frequent to least frequent. In this case, the corpus is the first 20 PubMed abstracts when "antibiotics resistance" is searched. This is not affected by the Keyword Frequency Search. When hovering over each bar of the histogram with a cursor, the word and number of times it appeared in the corpus pops up. Stopwords are discluded from this. In this example, "soil" is the most frequently occuring word in the antibiotic resistance corpus. This may be of interest to researchers, especially because soil bacteria are one of the most unexplored areas of microbiology and many antibiotic resistances have been traced to originating in soil bacteria.



The second plot is a scatter plot, where each point is a PubMed abstract, the X-axis represents the number of words in the abstract (abstract length), and the Y-axis represents the frequency of the selected Keyword Frequency Search word (in this example, "genes"). Hovering over each datapoint shows the values for each axis as well as the URL of the abstract. Clicking on the point reveals the abstract for the user to read directly on the app interface. This is an important feature of our application for users to easily access the information in one interface rather than flipping back and forth between different websites to gather the data they need.



Abstract

"Molecular biology techniques have assisted in the investigation of antibiotic resistance genes (ARGs) from various environments. However, their accuracy relies on primer quality and data interpretation, both of which require a full-coverage sequence database for ARGs. Here, based upon the abandoned Antibiotic Resistance Genes Database (ARDB), we created an updated sequence database of antibiotic resistance genes (SDARG). A total of 1,260,069 protein sequences and 1,164,479 nucleotide sequences, 56 times more sequences than ARDB, from 448 types of ARGs (enabling resistance to 18 categories of antibiotics) were collected and integrated with different hierarchical credibility and full-scale taxonomic information. Based on this comprehensive sequence database, an online pipeline - ARG analyzer (ARGA, http://mem.rcees.ac.cn:8083/) was developed to assess current ARGs primers, as well as annotate ARGs from environmental metagenomes. Thereafter, a list of 658 published primer pairs, targeting 173 ARGs, was evaluated using ARGA and integrated in ARGA as ARGs primer database. The results showed that 65.05% primers are of high specificity (≥90%), while only 29.79% primers cover >50% of targeted sequences, indicating a divergence in the quality of current ARG primers. Hence, primer assessment or redesign is highly recommended to improve the accuracy of ARGs studies. ARGs primer database was attached in ARGA to provide researchers alternatives to better survey ARGs in the environment."

The two plots work together to help the user gain an understanding of their field of interest and the word trends in it. The scatter plot allows accessibility of information, and the histogram can help the user refine their Keyword Frequency Search. All subsequent searches can be done from the webapp interface itself, and no further interaction with the Terminal/Command Line is necessary.

Teammate contribution

Damee Moon - Introduced the idea to the team, provided expertise on biology related topics.
Keerti Chalasani - Coded the webapp up, also helped with coding the backend.
Max Anderson - Helped with various coding tasks, developed scraper logic.
Andy Lu - Helped with various coding tasks, did the documentation.