ADS Project Group2–Mini Search Engine

1 How to Run it

If you use Pycharm, just install the packages as it prompts. If not, run the following command:

```
pip3 install -r requirements.txt
```

If you want to run from scratch, like, crawling URLs, you can uncomment the `# stage_texts()` in `main()`. But it takes some time because I want our result to be accurate to a tiny part of the work, so I used a lot of local files to save. You can still try it if you wish, and we will print "[Info] Saving text process xx%" to let you know where you are.

2 Code Explanation

2.1 (1) Run a word count over the Shakespeare set and try to identify the stop words (also called the noisy words) — How and where do you draw the line between “interesting” and “noisy” words?

> I divide this into 4 parts: get URLs, get texts, save texts to local, and get a noisy dictionary.
>
> I set a global variable as **STOP_POINT** to define the threshold of the noisy word. If the **frequency** of the word is more than 2000, I regard it as a stop word.

2.1.1 Get URLs

I want this search engine to be accurate to **every act and scene** (**chapter** for poems), so I crawl 2 times for works that have acts and scenes.

Pseudo Code:

```
'''
    @brief every URL that contains texts
    @param url: the base URL
    @return: a list that includes every URL that contains texts
'''
def get_all_links(url):  # DONE: get all links that contains texts
    response = requests.get(url)
    soup = BeautifulSoup(response.content, 'html.parser')
    links = soup.find_all('a')  # every sub link in the page

    valid_doc_links = []
    valid_scene_links = []

    for every link in links, href = link.get('href')
        if href is not external links and href endswith valid work names:
            valid_doc_links.append(url + href)
```

```
    for link in valid_doc_links, find sub_links in link
        for sub_link in sub_links, href = sub_link.get('href')
            if href has index, find its sub_sub_urls
                valid_scene_links.append(new url)

    return valid_scene_links
```

2.1.2 Get Text

I find all the texts that are surrounded by <blockquote> and add them to the text variable, and return it.
Pseudo Code:

```
    """
        @brief Get the text from the given URL.
        @param url: The URL to fetch the text from.
        @return: The text content retrieved from the URL.
    """
    def get_shakespeare_text(url):  # DONE: get the text from the URL
    response = requests.get(url)
    # text <blockquote> blocks
    text_blocks = soup.find_all('blockquote')
    for block in text_blocks:
        text += block.get_text() + '\n'  # add a new line after each block
    return text.strip()  # get rid of leading/trailing whitespaces
```

2.1.3 Save Texts to Local File

> Because it's slow to access all the URLs and save them to a local place, so I print its process bar to show where we are. In our file, we already saved it to the folder, so you don't need to run it unless you want to test its correctness.

Pseudo Code:

```
   """
        @brief Fetches Shakespeare's works from the web and saves them locally.
        @details
        This function fetches Shakespeare's works from a base URL, processes them, and
saves them as text files locally.
        The processed text files are then concatenated into a single file named
'0_all_texts.txt'.
        @return: None
    """
   def stage_texts():
    print("[Info] getting urls...")
    base_url = 'http://shakespeare.mit.edu/'
    works_links = get_all_links(base_url)

    directory_name = 'shakespeare_works'

    count = 0
    print("[Info] saving texts to local place... Please wait a while...")
    for link in works_links:
        if count % 50 == 0: # show process bar
```

```
            print("[Info] Saving text process " + str(int((count / 913) * 100)) + "%")
        count += 1
        file.write(texts) with filename of works and its act, scene, chapter, part

    all_texts = ''
    folder = 'shakespeare_works'
    for files in folder:
        all_text += texts in files
    file.write(all_texts)
```

2.1.4 Filter Stop Words into Dictionary

> In Python, **dictionaries** are implemented using **hash tables**, which provide an **average-case time complexity** of O(1) for both **insertion** and **retrieval** operations.
>
> In Python, the `Counter` is used to count the **occurrences** of elements in a sequence. But when we access the counts of specific elements using the `Counter` object, the time complexity for retrieval is O(1), similar to dictionary lookup, as it directly accesses the counts stored in the underlying dictionary.

```
"""
        @brief Counts words in the given text and identifies stopwords.
        @details This functioncounts the occurrence of each word, and identifies
stopwords based on a predefined threshold. Words with counts equal to or higher than
the STOP_POINT are considered stopwords
        @param text: The input text for word counting and stopwords identification.
        @return: 2 dictionaries containing dictionaries of noisy words and interesting
words, respectively.
"""
def word_count_and_stopwords_identification(text):
    word_counts = Counter(filtered_words)
    noisy_words = set()
    noisy_dic = {}
    interesting_words = set()
    interesting_dic = {}
    for word, count in word_counts.items():
        if count >= STOP_POINT:  # stop words 的阈值
            noisy_words.add(word)
            noisy_dic[word] = count
        else:
            interesting_words.add(word)
            interesting_dic[word] = count

    return noisy_dic, interesting_dic
```

2.2 (2) Create your inverted index over the Shakespeare set with word stemming. The stop words identified in part (1) must not be included.

> In this module, I have implemented three specific parts:

> 1. Preprocessing: read the text file content, tokenize the content, convert it to lowercase, and extract the word stems.
> 2. Iterate and create index: iterate over each word and its position in the text, check if the extracted stem is in noisy_dic. If not, add it to the inverted index. The index records the stem, filename, and the position of the word in the text.
> 3. Sort: sort the inverted index by the number of positions each word appears in, and then return the sorted index.
> 4. Save as file: save the file for easy debugging.

2.2.1 Preprocessing

In this step, we processed the text as follows: tokenization, converting to lowercase, and extracting word stems.

We utilized the nltk.stem library to perform stemming and used the lower() method to convert.

```
stemmer = PorterStemmer()
content = file.read().lower()
......
       stemmed_word = stemmer.stem(word)
       ......
```

2.2.2 Iterate and create index

We iterated over all the words in the text, and **if the word is not in the stop word dictionary**, we added the filename and occurrence position to the inverted index dictionary.

Pseudocode:

```
Function build_inverted_index(folder_path, noisy_dic)
    Initialize inverted_index as an empty dictionary

    For each text file in folder_path
        Read the file and tokenize its content
        For each word and its position in the file
            Stem the word
            If stemmed word is not in noisy_dic and inverted_index
                Add to inverted_index with its file and position
            Else if stemmed word is in inverted_index
                Update inverted_index with new file and position

    Sort inverted_index by the number of positions
    Return inverted_index
```

2.2.3 Sort

In order to speed up intersection retrieval during search, we perform a simple sorting based on the frequency of word occurrences when constructing the inverted index.

```
sorted_index = dict(sorted(inverted_index.items(), key=lambda x: len(x[1])))
```

2.2.4 Save as file

For debugging convenience, we output the inverted index results to a txt file.

```
Function save_inverted_index(inverted_index, output_path)
    Open output_path for writing
    For each word in inverted_index
        Write the word and its file positions in a specific format to the file
```

2.3 (3) Write a query program on top of your inverted file index, which will accept a user-specified word (or phrase) and return the IDs of the documents that contain that word.

> Our ideal query process is to input a sentence, and through analysis, we can efficiently obtain the source of the sentence or a chapter containing similar sentences. Although some factors affecting the results were found in the subsequent testing process, such as the imprecise search results caused by too many stop words in a sentence; Or the excessive occurrence of interesting words has led to the appearance of similar sentences in many chapters. This will be analyzed in detail in the subsequent implementation.
>
> In conclusion, in this module, I have implemented three specific parts:
>
> 1. Read and process input, as well as print query results;
> 2. Obtain the document names where all interesting words appear and the positions where each word appears in the document;
> 3. Calculate the minimum achievable variance in different documents and find the top five with the smallest variance.

2.3.1 Input and Output

According to our expected goal, we need users to input a sentence to be queried, and then we also need to divide the sentence into different words by spaces to form a word sequence for our subsequent query operations.

In addition, we also hope to support unlimited queries until users wish to exit the query program.

Therefore, we can take the following actions:

```
## @brief This script provides a user interface for querying words or sentences in a
book collection.
# The script continually prompts the user to enter a word or sentence to search for in
the book collection. It uses an inverted index to find the intersection of words and
their positions in the book titles. The script supports querying specific chapters,
scenes, or entire books. If the query fails, it notifies the user. The user can exit
the loop by entering "-1".
while 1:
    sentece = input("input prompts")
    Split sentence and convert to lowercase
    if the user enters -1:
        exit the current loop
```

```
    Call the search function to obtain ListOfBookTitles
    if ListOfBookTitles is empty:
        return a failure prompt
    else:
        for i in range(5):
            if ListOfBookTitles[i] is None:
                break
            # In order to make the search results more aesthetically pleasing, we
attempted to split the file names into three categories:
            book_name = Split ListOfBookTitles[i]
            if book_name[1] is "txt": # <BookName>.txt
                print <BookName>.txt
            elif book_name[2] is "txt": # <BookName>.<Chapter>.txt
                print <BookName>.<Chapter>.txt
            else:
                print <BookName>.<Act>.<Scene>.txt # <BookName>.<Act>.<Scene>.txt
```

2.3.2 Obtain Possible Document Names

In this section, our function retrieves the list of words to be searched and the inverted file index passed in. We find file names where all words have appeared by querying the inverted file index for each word.

It should be noted that we also need to filter stop words.

```
## @brief Finds the intersection of word positions in files using an inverted index.
#
#  This function takes an inverted index and a list of words, then uses the
PorterStemmer to stem the words. It looks up each stemmed word in the inverted index
to find all occurrences across the files. The function then identifies files where all
the stemmed words appear and records their positions within those files.
#
#  @param inverted_index The inverted index mapping stemmed words to their positions
in files.
#  @param words A list of words to search for in the inverted index.
#  @return A dictionary where keys are filenames and values are dictionaries mapping
each word to its list of positions in that file.
#
def find_word_intersection_and_positions(inverted_index, words):

    for word in words: # use PorterStemmer to stem the words
        find stemmed_word corresponding to the word
        if stemmed_word in inverted_index:
            # traverse the file positions of the word
            for filename, position in inverted_index[stemmed_word]:
                # record the file and the position of the word
                file_word_positions[filename][stemmed_word].append(position)

    intersection_files = { # find the intersection of the files
        filename: positions for filename, positions in file_word_positions.items()
        if len(positions) == len(appear_list)
    }

    return calculate_Var(intersection_files)
```

2.3.3 Calculate Words' Variance

This is the most important part of what I have completed. If we only go to the previous step, we will only search for results, but there is no corresponding advantage or disadvantage for this result.

In order to determine which result is better (closer to the user's search objective), we designed this algorithm:

$$
\begin{aligned}
Suppose\ intersection\_files = \{&FileName_0 : POS_0,\\
&FileName_1 : POS_1,\\
&\dots,\\
&FileName_n : POS_n,\}
\end{aligned}
$$

$$
\begin{aligned}
Then\ for\ FileName_i, POS_i\ is \{&word0 : [pos_{00}, pos_{01}, \dots, pos_{0\alpha_0}],\\
&word1 : [pos_{10}, pos_{11}, \dots, pos_{1\alpha_1}],\\
&\dots,\\
&wordm : [pos_{m0}, pos_{m1}, \dots, pos_{m\alpha_m}]\}
\end{aligned}
$$

$$
Search\ relevance:\ \sigma_{FileName} = \min_{i_k, k\in[1,m]} Var[pos_{0i_0}, pos_{1i_1}, \dots, pos_{mi_m}]
$$

Finally, we will sort all the searched files according to their search relevance to obtain the top five search results.

```
## @brief Calculates the minimum variance among all combinations of data points.
#
#  This function generates all possible combinations of data points provided in a
dictionary, then calculates the variance for each combination. It identifies and
returns the minimum variance found among these combinations.
#
#  @param data A dictionary where keys are categories and values are lists of data
points.
#  @return The minimum variance found among all combinations of the data points.
#
def calculate_the_min_var(data):
    # generate all combinations of the data
    all_combinations = itertools.product(*(data[key] for key in sorted(data.keys())))

    min_variance = float('inf')
    min_combination = None

    # return the combination with the minimum variance
    for combination in all_combinations:
        variance = np.var(combination)
        if variance < min_variance:
            min_variance = variance
            # min_combination = combination
    return min_variance


## @brief Calculates the variance for each file in the intersection_files and updates
the global minimum variance and combination.
#
#  This function iterates through each file in the intersection_files, calculates the
minimum variance of the positions using `calculate_the_min_var`, and updates the
global minimum variance and corresponding combination.
#
#  @param intersection_files A dictionary of files where each file contains positions
of words.
#  @return A list of files with the minimum variances.
```

```python
#
def calculate_Var(intersection_files)
    for filename in intersection_files:
        var = calculate_the_min_var(intersection_files[filename])
        for i in range(5):
            if min_var[i] > var:
                min_var[i] = var
                min_comb[i] = filename
                break
    return min_comb
```

3 Run tests to show how the thresholds on query may affect the results

3.1.1 Common Cases

Sentences with multiple possibilities:

We list 4 more possible results. And we use `colorama` to make the output colorful.

```
/usr/bin/python3 /Users/anya/Documents/ADS-Projects/proj1/Proj1.py
[info] accessing noisy dic...
[info] building inverted file index...
[Input Info] Please enter the word or sentence you want to query, enter -1 to quit: All I know not what you call all
"All I know not what you call all" probably comes from Act 4, Scene 3 of two_gentlemen
More possible results are as follows (we list 4 more at most):
"All I know not what you call all" probably comes from Act 3, Scene 5 of timon
"All I know not what you call all" probably comes from Act 1, Scene 1 of timon
"All I know not what you call all" probably comes from Act 5, Scene 1 of 3henryvi
"All I know not what you call all" probably comes from Act 1, Scene 3 of troilus_cressida
```

Sentences with only one possibility:

```
[Input Info] Please enter the word or sentence you want to query, enter -1 to quit: it is the east and juliet is the sun
"it is the east and juliet is the sun" probably comes from Act 2, Scene 2 of romeo_juliet
[Input Info] Please enter the word or sentence you want to query, enter -1 to quit:
```

Sentence with noise:

> This case shows a little robustness of our search engine

```
[Input Info] Please enter the word or sentence you want to query, enter -1 to quit: aaaaaaaaaaa tell me flatly I am no proud Jack
"aaaaaaaaaaa tell me flatly I am no proud Jack " probably comes from Part 2, Act 2, Scene 4 of Henry IV
[Input Info] Please enter the word or sentence you want to query, enter -1 to quit:
```

Words:

```
[Input Info] Please enter the word or sentence you want to query, enter -1 to quit: praying
"praying" probably comes from Act 2, Scene 8 of merchant
More possible results are as follows (we list 4 more at most):
"praying" probably comes from Act 4, Scene 2 of cymbeline
"praying" probably comes from Act 4, Scene 1 of henryv
"praying" probably comes from Act 2, Scene 2 of richardii
"praying" probably comes from Act 5, Scene 5 of merry_wives
[Input Info] Please enter the word or sentence you want to query, enter -1 to quit:
```

Stop words:

We print an red ERROR message for stop words, yet the program is still running.

```
[Input Info] Please enter the word or sentence you want to query, enter -1 to quit: thee
[ERROR] nothing found,because your words is noisy_words!
[Input Info] Please enter the word or sentence you want to query, enter -1 to quit:
```

Words that never appeared

```
[Input Info] Please enter the word or sentence you want to query, enter -1 to quit: iwannascream
[ERROR] Nothing found,because the words never appeared!
[Input Info] Please enter the word or sentence you want to query, enter -1 to quit: aaaaaaaa bbbbbbb ccccc
[ERROR] Nothing found,because the words never appeared!
```

4 Bonus: What if you have 500 000 files and 400 000 000 distinct words? Will your program still work?

The program probably still works.

1. **Pre-sorting**: During the construction of temporary indexes, pre-sorting each block enables more efficient merging during subsequent stages. This approach minimizes memory and disk usage and ensures the final index is constructed in a sorted manner, enhancing retrieval efficiency.
2. Disk Storage and Block Processing: By **storing indexes and other data structures on disk** rather than entirely loading them into memory, we avoid issues related to insufficient memory and allow for processing larger datasets. Additionally, partitioning the data into manageable blocks and serializing them to disk when they reach a certain threshold helps in better management of memory and disk resources.
3. Advancements in Hardware: Although the standard PC's main memory typically ranges around 8GB, modern computers often come with larger main memory capacities, such as 16GB. These hardware advancements significantly alleviate memory constraints, making it easier to handle large-scale datasets.