CPE202

Project Assignment 4

Document Search Engine

Objective

In this assignment, you are going to actually build something useful using the data structures you have learned in this course. Namely, you are going to build a search engine for documents, which finds documents relevant to query terms given by your user among documents stored in a directory on your computer.

Prerequisites

You are going to use one of the hash tables you have implemented in labs. Make sure they work. Fix them if there are some issues. We are also going to use the import_stopwords function from lab 8.
You also need to add keys() function, which returns a list of keys in the hash table in your class for the hash table.

Concepts and Terminologies

Inverted Index

An index storing a mapping from content, such as words, to its location in a document. In this assignment, we cut corners and just build an inverted index from terms to documents.

Term Frequency (TF)

The frequency of a term in a document.

Document Frequency (DF)

The frequency of document that contains a particular term or terms.

TF-IDF

TF * IDF, where IDF is inverted DF.

$$\mathrm{idf}(t, D) = \log \frac{N}{|\{d \in D : t \in d\}|}$$

The idea is to penalize the term frequency by the number of documents containing the term because common term across multiple documents is believed to have small discriminative power. In this assignment, we cut corners again and we use weighted frequencies, which can be computed more efficiently, instead.

Program Structure

Create project4.py

SearchEngine class

Builds and maintains an inverted index of documents stored in a specified directory and provides a functionality to search documents with query terms.

main function

The entry point of your program. It takes a directory name as its command line argument.
It will create an instance of SearchEngine class by passing the directory name.
Then, it will get into an infinite loop.
Inside the loop it will present a prompt to the user and waits for inputs (use input() function).
If the user types q, it will exit from the loop and end the session.
If the user types s: followed by some space separated terms (ex. s:computer science), it will search for relevant documents and print a list of file names in the descending order of relevancy. All search query terms need to be converted into lower cases in your search engine.
Do not forget if __name__ == '__main__' at the bottom of the project4.py file.

Attributes of SearchEngine class

- directory (str) : a directory name
- stopwords (HashMap) : a hash table containing stopwords
- doc_length (HashMap) : a hash table containing the total number of words in each document

- term_freqs (HashMap) : a hash table of hash tables for each term. Each hash table contains the frequency of the term in documents (document names are the keys and the frequencies are the values)

Constructor for SearchEngine class

Take a directory name and a hash table containing stopwords as arguments.
Initialize doc_length, term_freqs with empty hash tables. Assign the stopwords hash table to stopwords. Call self.index_files(directory) and index files contained in the directory.

```
def __init__(self, directory, stopwords):
    self.doc_length = HashMap() #Replace HashMap() with your hash table.
    self.term_freqs = HashMap()
    self.stopwords = stopwords
    self.index_files(directory)
```

Methods in SearchEngine class

Preprocessing

Create a function read_file() in SearchEngine class to read a text file. The function needs to read all words contained in the file except for stop words. (*from now on we are going to call a function defined in a class as a method*)
Use “with open() as infile:” so that the opened file will be automatically closed when you get out of the with block. Read more about the with here:
https://realpython.com/working-with-files-in-python/
And here:
https://realpython.com/read-write-files-python/

```
def read_file(self, infile):
    """A helper function to read a file
    Args:
       infile (str) : the path to a file
    Returns:
        list : a list of str read from a file
    """
```

Create a method parse_words() in SearchEngine class. It takes a list of strings as an argument. Each string contain multiple words. Split each string into words at spaces. Convert all words to lower cases and remove new line chars.

```
def parse_words(self, lines):
    """split strings into words
    Convert words to lower cases and remove new line chars.
    Exclude stopwords.
    Args:
      lines (list) : a list of strings
    Returns:
      list : a list of words
    """
```

It is recommended that you create a helper function for the parse_words function to exclude stop words.

```
def exclude_stopwords(self, terms):
    """exclude stopwords from the list of terms
    Args:
      terms (list) :
    Returns:
      list : a list of str with stopwords removed
    """
```

Create a method count_words() in SearchEngine class.

```
def count_words(self, filename, words):
    """count words in a file and store the frequency of each
    word in the term_freqs hash table. The keys of the term_freqs hash table shall be
    words. The values of the term_freqs hash table shall be hash tables (term_freqs
    is a hash table of hash tables). The keys of the hash tables (inner hash table) stored
    in the term_freqs shall be file names. The values of the inner hash tables shall be
    the frequencies of words. For example, self.term_freqs[word][filename] += 1;
    Words should not contain stopwords.
    Also store the total count of words contained in the file in the doc_length hash table.

    Args:
      filename (str) : the file name
      words (list) : a list of words
    """
```

Create a method index_files, which process all text files in a specified directory and build an inverted index. Use python builtin os.listdir(directory) function to get a list of files in a directory. Use os.path.join(directory, item) to construct the full path of each file. Use os.path.isfile(item) to check if the item is a file. If it is not a file, i.e. directory, skip it. Use os.path.splitext(item) to split a

path into a file extension and the rest. Check if parts[1] == '.txt'. Only process text files. For each text file, process it with read_file(), parse_words(), and count_words() functions. This function needs to be called in the constructor __init__.

```
def index_files(self, directory):
    """index all text files in a given directory
    Args:
      directory (str) : the path of a directory
    """
```

Searching

Create a method get_wf() in SearchEngine class.

```
def get_wf(self, tf):
    """comptes the weighted frequency
    Args:
      tf (float) : term frequency
    Returns:
      float : the weighted frequency
    """
    if tf > 0:
      wf = 1 + math.log(tf)
    else:
      wf = 0
    return wf
```

Create a method get_scores() in SearchEngine class. Use term_freqs and doc_length hash tables. Create a hash table called scores to store scores for each file. For each term in a query compute the score of each document using the formula: score = weighted frequency. Add it to the score stored in the scores hash table for the document. After the score has been accumulated, normalize the score for each document by dividing it by the total word count in the file (we exclude stopwords from the count).
Use keys() function of hash table to get keys in the table to extract all key value pairs stored in the hash table. Files with scores being 0 will be ignored.

```
def get_scores(self, terms):
    """creates a list of scores for each file in corpus
    The score = weighted frequency / the total word count in the file.
    Compute this score for each term in a query and sum all the scores.
    Args:
```

```
        terms (list) : a list of str
    Returns:
        list : a list of tuples, each containing the filename and its relevancy score
    """
    #The code listed below is pseudo code
    scores = HashMap()
    For each query term t
        Fetch a hash table of t from  self.term_freqs
        For each file in the hash table, add wf to scores[file]

    For each file in scores, do scores[file] /= self.doc_length[file]
    Return scores
```

Create a method rank() in SearchEngine class. You can use python builtin function sorted(). You need to sort the (filename, score) pairs in descending order of relevancy scores. An example usage of sorted, sorted(scores, key=lambda x:x[1], reverse=True), for using the second item in a tuple as a key for sorting. The argument, scores, shall be the return value from the get_scores function.

```
def rank(self, scores):
    """ranks files in the descending order of relevancy
    Args:
        scores(list) : a list of tuples: (filename, score)
    Returns:
        list : a list of tuples: (filename, score) sorted in descending order of relevancy
    """
```

Create a method search() in SearchEngine class. It takes a query string user typed as an argument called query. The query string needs to be parsed into words, being converted to lowercase with stop words removed with parse_words() method. Remove duplicate words using a hash table. In this function, the parsed query terms will be passed to get_scores() function, and the resulting list of tuples of filename and score will be passed to rank(), and its result will be displayed in descending order of the scores onto the screen.

```
def search(self, query):
    """ search for the query terms in files
    Args:
        query (str) : query input
    Returns:
        list :  list of files in descending order or relevancy
    """
```

Helper Functions

You are free to create additional functions or methods.

Test

Download a zip file containing test files to be indexed by your search engine.
For a query “ADT”, your search engine needs to output the following lines on a screen:
docs/data_structure.txt

The score of this document should be around 0.01.

For a query “Computer Science”, your search engine needs to output the following lines on a screen:
docs/test.txt
docs/information_retrieval.txt
docs/hash_table.txt

The score of test.txt should be 1.0.

Test methods and functions that return some values.

Further Development

The search engine presented here is very primitive. For example, the engine does not account for different forms of the same word, nor synonyms, misspelled words. Also, it can not do proximity search, in which the engine searches for two or more words that occur within a certain number of words from each other. You can extend the engine to do those things. Also, you can extend the engine so that it outputs the location where query terms occur in documents.

Submission and demo

Demo your work to your instructor or TA. Zip all your files including files containing your hash tables and import_stopwords function. Submit your zipped file to polylearn.