Introduction to Python Programming Software Assignment: Search Engine

Eva Richter

April 2021

1 Introduction

This project is an implementation of a small search engine. The text corpus which is searched is given in an XML-like format and is pre-processed. The core of the search engine is an index that weights terms according to the tf.idf weighting scheme. The implementation includes a simple 'interface' that allows to execute queries via the console (python shell).

2 Workflow

The work flow for the project is given as:

1. Parsing the XML file
2. Saving the index for term frequency and inverse document frequency
3. Loading the indexes
4. Taking input query from the user
5. Pre-processing the query
6. Getting the vector tf.idf for query
7. Calculating the similarity of a query with each document's tf.idf
8. Getting the documents with the highest frequency

3 Methods

These are the methods used in the project that are not bounded to any classes, because of their functionality.

- **src.xmlParser.get_data**

  Extracts the XML file and returns the dictionary of the document and the content

  1. **Parameters:**

     XML file (str)

     Name of the XML file to be parsed

  2. **Returns:**

     dictionary

     dictionary with keys as document id and values, list of paragraphs and headlines

- **src.utils.removing_empty_tokens**

  Removes the token if it is an empty string

  1. **Parameters:**

     text (list)

     list of tokens

  2. **Returns:**

     list

     list of tokens filtered

- **src.utils.tokenize**

  Tokenizes the text passes

  1. **Parameters:**

     text (str)

     text to be tokenized

2. **Returns:**

   list

   list of tokens

- **src.utils.stemming**

  stems each word

  1. **Parameters:**

     text (list)

     list of tokens

  2. **Returns:**

     list

     stemmed list of tokens

- **src.utils.cleaning**

  Removes XML tags

  1. **Parameters:**

     text (str)

     text

  2. **Returns:**

     str

     cleaned text

- **src.utils.remove_punctuation**

  Removes punctuation from tokens

1. **Parameters:**

   text (list)

   list of token

2. **Returns:**

   list

   list of cleaned tokens

- **src.utils.pre-processing**

  pre-processes each text

  1. **Parameters:**

     text (list): list of tokens

  2. **Returns:**

     dictionary

     dictionary with keys as document id and values list of pre-processed tokens

- **src.utils.similarity**

  calculate the cosine similarity between two vectors

  1. **Parameters:**

     value1_n (list): list of float values representing a document as a vector
     value2 (list): list of float values representing a document as a vector
     value1_d (list): list of float values representing a document as a vector

  2. **Returns:**

     float: cosine similarity score

4 CLASSES USED AND THEIR ATTRIBUTES

- Class **src.xmlParser.xmlParser(xml.sax.handler.ContentHandler)**

  Class that is inherited from xml.sax.handler.ContenHandler which parses the XML file and returns the content as text for each document tag mentioned in the XML file.

  1. **Attributes**
     (a) **dictionary**
     dictionary to hold the document id as key and list of paragraph and headline as value
     (b) **_charBuffer**
     holds list of characters
     (c) **docid**
     document id
     (d) **paragraph**
     paragraph from each text tag
     (e) **headline**
     text from headline tag
  2. **Methods**
     (a) **_flushCharBuffer:**
     flushes character buffer attribute after end of every element
        i. **Returns**
        str: text for the given tag.
     (b) **startElement:**
     after reading off an element is started it calls this function and passes the name of element. This is the overriding function for ContentHandler
        i. **Parameter:** name (string): name of the tag attrs (string): attribute of the tag
     (c) **endElement:**
     after reading off an element is finished it calls this function and passes the name of element. This is the overriding function for ContentHandler
        i. **Parameter:** name (string): name of the tag
     (d) **characters** reads characters into buffer and calls the function with data. This is the overriding function for ContentHandler

i. **Parameter:** data (str)

(e) **clearFields**
clearing the custom buffer of current document id, paragraph as well as headline. This is the overriding function for ContentHandler

- Class **src.tfidf.TfIdf(collectionName)**

  Class that is inherited from xml.sax.handler.ContenHandler that parses the XML file and returns the content as text for each document tag mentioned in the XML file.

  1. **Attributes**
     (a) **collectionName**
     name of the XML file to be parsed
     (b) **unique_words**
     holds list of unique words

     (c) **docids**
     holds set of document ids

     (d) **tf**
     dictionary that holds term frequency per document per token

     (e) **idf**
     dictionary that holds inverse document frequency for each token

  2. **Methods**
     (a) **read_data:**
     reads the XML data and returns the dictionary containing document id, text and headline

     i. **Returns dict:** "docid":[headline, text]
     None

     (b) **create_idf** writes invert document frequency in a tab separated file
     (c) **doc_freq:**
     calculate the document frequency for each term. Document frequency is the count of documents the particular token is in

     i. **Parameter:**
     pre-processed (dict): keys are document id and values are list of tokens which were pre-processed

ii. **Returns dictionary:** keys are tokens and values are the document frequency of that token

(d) **term_freq**
creates the term frequency per document per token

i. **Parameter:**
docid (str): document id **tokens(list of string)**: list of tokens in a document **token (string)**: a single token whose term frequency is to be calculated

ii. **Returns**
dictionary: keys are tuples of documents and tokens. Values are the term frequency

(e) **create**
the main creates a function that calculates document frequency, inverse document frequency, term frequency and sends it to write it in a file

(f) **load**
Reads the term frequencies and inverse document frequencies and writes it into the attribute self.tf and self.idf

- Class **src.softwareAssignment.SearchEngine(collectionName, create)**

  This class calls the tf.idf class to read index or create it.

  Parameters:
  collectionName (str): name of the collection **create (bool)**: whether to create the index or load it.

  1. **Attributes**
     (a) **tf_obj**
     Object of **tfidf** class

  2. **Methods**
     (a) **executeQuery:**
     This functions process the query and searches for similar document and prints those with similarity score.

     i. **Parameters**
     queryTerms (list of string): query tokens to be searched

(b) **executeQueryConsole:**
This function is a console that asks for a user's query in loop unless they click ENTER

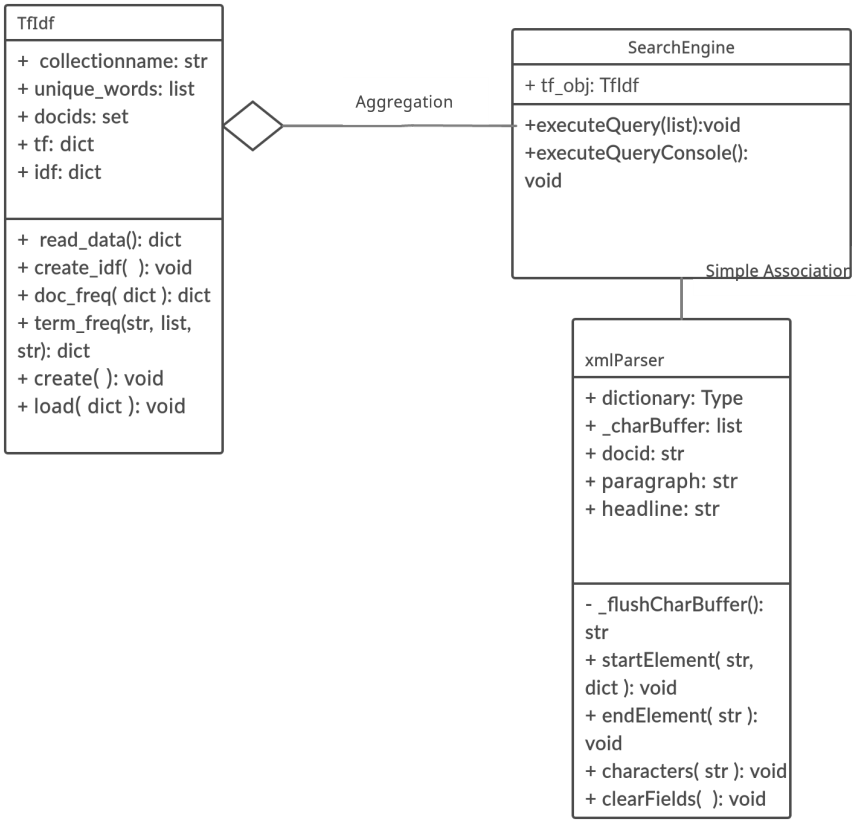

Figure 1: Class Diagram for the Project

5 HOW TO RUN

1. Go to src folder in the project

1. To run the the interface along with reading and re-creating index file. Run:

```
python3 softwareAssignment.py
```

2. To run the the interface along with reading xml and loading index file without creating. Run:
   The default value for collection name is nytsmall. And default value for create is True

```
python3 softwareAssignment.py -create False -collectionname
    ↪ nyt199501
```

6 PROBLEMS CAME ACROSS

1. First I tried to load all the tf.idf vectors for all documents in the search engine after creating the index. This was working fine in the small corpus. However, for the large one, I got some memory issue. This problem was solved by generating tf.idf for each document at once and calculating similarity with query document.

2. The second problem occured during pre-processing, all the documents were stemmed while saving the index. But I forgot to pre-process the query tokens. As a result, words like "hurricane" which was stemmed to "hurrican" in the document were not found during query td idf calculation. This problem was solved by pre-processing the query tokens as well.