Software Assignment: Search Engine

Final Project Documentation for Introduction to Python

Averie So

Created on : March, 2022

Contents

Chapter 1: Introduction

This is the documentation for the final project of Introduction to Python Programming. This search engine reads a corpus from a XML file and returns the most similar documents against a list of query words. The similarity is calculated based on the tf-idf weighting scheme. In this documentation,

- Chapter 1 explains the ways to use this program and how to install it.
- Chapters 2-6 explain all the functions in this module, split into 4 sections according to the order of how the search engine is built.
- Chapter 7 is a brief demo of how to use a search engine in python with various parameters.
- Chapter 8 is a class diagram of all class objects in this module
- Chapter 9 explains the issues faced in the process of implementing this search engine.

Chapter 2: Installation

Run the search engine from command line

If you would like to run the search engine directly from terminal, go to the directory that contains the folder you've downloaded, inside which you should find a file called `softwareAssignment.py`. You may run the search engine with the command:

```
python softwareAssignment.py
```

This program assumes that an index already exists, so make sure that the files containing the tf and idf indices are also within the same directory as the program. You will be prompted to enter the name of the index you would like it to read, eg. enter `nytsmall` if you would like to read `nytsmall.idf` / `nytsmall.tf`.

Import to python

If you would like to use this module in python, move `softwareAssignment.py` and `funcs.py` to the directory you would like to work in. This will also import all functions from `funcs.py` to your python file. You may import by this statement:

```
from softwareAssignment import *
```

There is a number of dependencies, make sure to have them installed in order to use this module:

- re
- string
- stemming.porter2
- math
- itertools
- collections
- operator
- numpy

Chapter 3: Parsing

`CorpusHandler(xml.sax.ContentHandler):`

A class which inherits from xml.sax.ContentHandler. It reads a .xml file, preserves only those that are tagged "HEADLINE", or "P" or TEXT".

`__init__(self):`

Constructs a CorpusHandler object.

`startElement(self, tag, attributes):`

Called when the parser encounters the opening element in a document, in this case the tag "DOC", and saves each document id by the attribute "id" every time a new document is identified.

Parameters

- `tag` (*str*) – An opening element in a xml document, in this case "DOC"
- `attributes` (*Attributes*) – The attributes of the opening element.

`endElement(self, name):`

Called when the parser encounters the closing tag, in this case the tag "DOC", and adds the content of the current document to the class object (a dictionary containing all the content that has been parsed) as value of this particular document. It then resets the content for the next document.

Parameters `name` (*str*) – A closing element in a xml document, in this case "DOC"

`characters(self, content):`

Identifies each chunk of character data by the tags "HEADLINE" or "P" or "TEXT" and adds them to the current content object, ignoring the others.

Parameters `content` (*str*) – chunk of character data identified by the SAX parser.

`get_parsed_data(self):`

Returns the class object which is a dictionary for the whole file, with document ids as keys and their content as values.

Chapter 4: Pre-processing

`remove_ws(token):`

Removes white space from a token

> **Parameters** `token` (*`str`*) – The token from which white space is to be removed.
> **Returns** The token with white space removed
> **Return type** str

`remove_puncs(token):`

Removes punctuation from a token

> **Parameters** `token` (*`str`*) – The token from which punctuations are to be removed.
> **Returns** The token with white space removed
> **Return type** str

`preprocess(text):`

Preprocesses a line of text into a list of tokens. Preprocessing includes: lowercase, splitting text into tokens at white space, removing extra white space, removing punctuations, removing empty tokens

> **Parameters** `text` (*`str`*) – The string which is to be preprocessed.
> **Returns** The preprocessed string.
> **Return type** list[str]

`token_to_numeric(preprocessed_dict):`

Returns a token-to-index association for a given corpus. Each token will be given a unique index for identification.

> **Parameters** `preprocessed_dict` (*`dict`*) – The corpus from which tokens are to be assigned an index value. It should be in the form of a dict with document id as key and the document's content as value (same format as the output from `parse()`)
> **Returns** A dict with tokens as keys and their index value as values.
> **Return type** dict

`parse(path_to_xml_filename, run_preprocess = True):`

Parses a xml file into text by turning the content into the ContentHandler class. It can optionally run preprocessing steps. This function automatically outputs the number of documents that were parsed.

> **Parameters**
> - `path_to_xml_filename` (*`bool`*) – The path to which the xml file is stored, without the ".xml" extension.
> - `run_preprocess` – An optional argument to decide whether preprocessing (from the preprocess function) is to be applied
>
> **Returns** A dictionary object with the document id as key and the content of that document as value.
> **Return type** dict

Chapter 5: Tf-Idf

`idf(preprocessed_dict, ind2word = True, save_idf = False):`

Returns the Inverse-Document-Frequency values for all words in a corpus.

Parameters

- `preprocessed_dict` (*dict*) – The corpus from which idf values are to be calculated. It should be in the form of a dict with document id as key and the document's content as value (same format as the output from `parse()`)
- `ind2word` (*bool*) – An argument to decide whether the words in the corpus is to be represented words instead of an index representing that word. If False, a np.array is returned; else, a dict is returned. The default is True.
- `save_idf` (*bool*) – An argument to decide whether to save the idf index into a file in the current directory. The user will be prompted to enter a desired name for the idf file to be saved as. The extension is '.idf'. Each line in the file will be a token and its idf value, separated by tabs. The tokens are sorted alphabetically. The default is False.

Returns A dictionary or np.array, depending on whether ind2word is True. If a dict is returned, it will have the tokens as keys and the idf values as values.

Return type dict or np.array

`tf(token, document):`

A function that computes tf value for a token from one single document.

Parameters

- `token` (*str*) – The token for which tf value is to be calculated
- `document` (*list[str]*) – The document that contains the token to have its tf value calculated. It should be a list that contains all the words from the document.

Returns tf value for the token given the document.

Return type float

n_token_in_doc = document.count(token) maxOccurrences = collections.Counter(document).most_common(1)[0][1] tf_value = n_token_in_doc / maxOccurrences return tf_value

`tf_corpus(preprocessed_dict, ind2word = True, save_tf = False):`

Returns the Term-Frequency values for values for every token and document in a corpus.

Parameters

- `preprocessed_dict` (*dict*) – The corpus from which tf values are to be calculated. It should be in the form of a dict with document id as key and the document's content as value (same format as the output from `parse()`)
- `ind2word` (*bool*) – An argument to decide whether the words in the corpus is to be represented words instead of an index representing that word. If False, a np.array is returned; else, a dict is returned. The default is True.
- `save_tf` (*bool*) – An argument to decide whether to save the tf index into a file in the current directory. The user will be prompted to enter a desired name for the tf file to be saved as. The extension is '.tf'. Each line in the file will be a document id followed by a token in the document and the corresponding tf value, separated by tabs. The tokens are sorted alphabetically. The default is False.

Returns A dictionary or np.array, depending on whether ind2word is True. If a dict is returned, it will be a nested dict with document id as keys, then token as keys and tf values as values.

Return type dict or np.array

`tfidf(token, doc_name, tf_dict, idf_dict):`

Returns the tf-idf value for a given token, a document, a tf_dict and idf_dict (output from tf() and idf()) of a corpus.

Parameters

- `token` (*`str`*) – the token for which td-idf value is to be calculated
- `doc_name` (*`str`*) – the document id for which td-idf value is to be calculated
- `tf_dict` (*`dict`*) – A nested dict with document id as keys, then tokens as keys and tf values as values. (Same format as output from `tf_corpus()`)
- `idf_dict` (*`dict`*) – A dict with tokens as keys and idf values as values. (Same format as output from `idf()`)

Returns A tf-idf value

Return type float

Chapter 6: Cosine similarity

`similarity(query, document, tf_dict, idf_dict):`

Given a query (a word or list of words), and a document, tf index and idf index from a corpus, return the cosine similarity between the query and the document.

Parameters

- `query` (*list [str]*) – list of words to be compared against each of the document inz the tf-idf index.
- `document` (*str*) – the document id for which similarity is to be computed
- `tf_dict` (*dict*) – A nested dict with document id as keys, then tokens as keys and tf values as values. (Same format as output from `tf_corpus()`)
- `idf_dict` (*dict*) – A dict with tokens as keys and idf values as values. (Same format as output from `idf()`)

Returns A cosine similarity value

Return type float

Chapter 7: Search engine

Note that all functions up till this class is in `funcs.py`, while the SearchEngine class is in `softwareAssignment.py`.

`SearchEngine:`

This class implements a search engine based on a given .xml file or .tf/.idf files. The user may enter a list of query terms and it will return the most similar documents. This process will continue until the user presses ENTER without any query.

`__init__(self, collectionName, create):`

Parameters

- `collectionName` (*str*) – The filename of the document collection (without the .xml at the end)
- `create` (*bool*) – If create=True, the search index should be created and written to files. If create=False, the search index should be read from the files, the index files are assumed to have the same filename as the document collection, the idf index should have a '.idf' extension and the tf index should have a '.tf' extension.

`executeQuery(self, queryTerms):`

Returns the 10 highest ranked documents together with their tf.idf-sum scores, sorted score.

Parameters `queryTerms` (*list [str]*) – List of query terms

Returns the 10 most similar documents to the list of query and their tf.idf-sum scores, sorted by similarity. Returns less than 10 if there are less than 10 documents that contain words in the query list.

Return type list[(tuple)]

`executeQueryConsole(self):`

Starts the interactive console. It will ask the user to enter a list of query with each term separated by white space and prints the list of most similar documents. It will continue to prompt for query until the user simply hits ENTER without any query.

Chapter 8: Demo

First, import the `softwareAssignment` module.

```
>>> from softwareAssignment import *
```

Create an object of the `SearchEngine` class. Specify the relative file path from which the .xml file or the .tf / .idf indices are to be read. In this example, 'nytsmall.xml' exists in the same directory as the python file, so only the filename will be enough. The preprocessing steps are fixed in this program, so the only other argument to be specified is whether you would like to read from indices or create new ones.

If they are to be read from existing files:

```
>>> searchEngine = SearchEngine('nytsmall', create=False)
Reading index from file...
Done.
```

Otherwise, create new indices. You can choose to save the indices to the current directory, and you will be prompted to enter the desire name these files are to be saved as. They will have the extensions .tf and .idf.

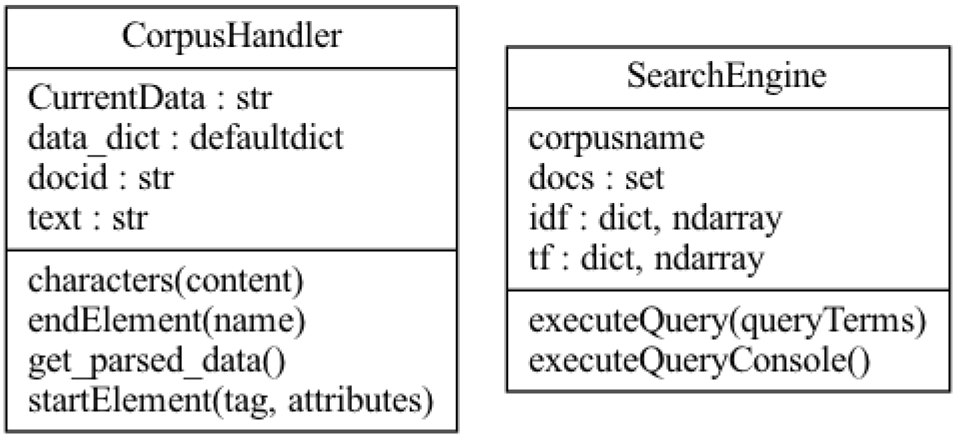
```
>>> searchEngine = SearchEngine('nytsmall', create=True)
Parsing xml file...
there are 102 documents.
Preprocessing data...
would you like to save the tf and idf files to your current directory? (y/n)y
saving files to current directory...
creating tf index...
there are  7739 unique words in this corpus.
Please input file name for this tf index: nytsmall
creating idf index...
there are  7739 unique words in this corpus.
Please input file name for this idf index: nytsmall
```

Call the `executeQueryConsole()` class method to start the search engine. You will be prompted to enter your query until you press ENTER without any query.

```
>>> searchEngine.executeQueryConsole()
Please enter query, terms separated by whitespace: america
I found the following documents:
NYT_ENG_19950101.0048 (0.06648198056830427)
NYT_ENG_19950101.0005 (0.05469997691711485)
NYT_ENG_19950101.0043 (0.05013670293332015)
NYT_ENG_19950101.0068 (0.04302969084462847)
NYT_ENG_19950101.0016 (0.04172672945245617)
NYT_ENG_19950101.0019 (0.04068984613046628)
NYT_ENG_19950101.0028 (0.03890415888131098)
NYT_ENG_19950101.0070 (0.032815744660355584)
NYT_ENG_19950101.0094 (0.02485507094109603)
NYT_ENG_19950101.0032 (0.021370882201949073)
Please enter query, terms separated by whitespace:
End of query.
```

Chapter 9: Class diagram

Here is the class diagram for all class objects from the modules:



Chapter 10: Issues

Problem remaining

The problem faced in implementing the search engine is that the similarity scores are not identical to the assignment guide. From the few samples provided in the guide, this search engine is able to retrieve the same order of documents, but the similarity scores are slightly different (eg. from the third decimal place). I have checked all preprocessing steps and the values from the tf and idf indices also align with the samples from the guide, but in the end I am still unable to solve this problem.

Problem solved

With my initial implementation it took a long time to process the large file (nyt199501). The solution was to transform every word into numbers and calculate the td/idf values with np arrays instead of dictionaries. This made it more efficient and the program should be able to return tf/ idf indices for the large file in 90 seconds.