Title:
Assignment 4:
Build a simple Vector Space Information Retrieval System Corpus: Wikipedia movie corpus

Author:
Daniel Zhang

Date:
March 21, 2019

Description:
Implement a simple vector space information retrieval system supporting disjunctive ("OR") queries over terms and apply it to your wikipedia movie corpus. The system consist of (1) an indexing module that constructs an inverted index, with term dictionary and postings lists for a corpus and (2) a run-time module that implements a Web UI for searching the corpus, returning a ranked result list and presenting selected documents.

Dependencies:
This program was run on python3.7. And here are some packages used to run the profile:
Nltk: https://www.nltk.org
Shelve: https://docs.python.org/3/library/shelve.html
flask: http://flask.pocoo.org
json: https://docs.python.org/3/library/json.html
for nltk and flask, these packages need to be installed using the "pip install" command, while Collections, re, html, math, heapq: just import these packages, no need to install separately

Build Instructions:
In order to build the vector space information retrieval system, firstly the inverted index should be made. So for each term occurred in documents, a posting list is built and each element in the positing list consist of document id and if-idf score of the term.

Then a run-time module was built based on flask frame which contains 3 different UI pages, the query page can get user's input and pass it to the result page which will call the result function to calculate the cosine similarity between query and each document and show 10 results each page in a descending order according to the similarity score. And each result's title also works as a link to the doc page which will use the document id and extract detailed data and show on the page.

Run Instructions:

In order to run the information retrieval system, we need firstly run the Vs_index.py to make sure the inverted index and document length shelve are built. Then the Vs_query.py should be run to start the web UI and do query on the web. And the user could just input their query related to the text and title in the main query box click submit to do the query.

After got the result, user could also click the “More like this” button under each result to see more results similar with the one.

Modules:

a. **Modules in Vs_index:**

Create_word_frequency_list: This method call normalization method to do normalization for document data and create word frequency for each word in each document.

Word_normalization: This method will call all the normalization methods to remove stop words, lower the case, stem each word and remove punctuation etc.

In detail, it will firstly lower case all words in corpus, then remove stop words using nltk methods. Besides, since the database is all about movie, the words such as “movie”, “film”, “actor” etc. that occur highly frequently in almost every movie text data and help little to find what a user need. They are also added into the stop word list. Then all the punctuations are also removed since user rarely use that during query. After that, all words are stemmed using Porter’s stemmer because of simplicity and speed so that words in different inflectional forms can be matched.

Make_inverted_index: This method create term posting list for each term and the element in each term is tuple which contains the document id and occurrence frequency of the term.

Create_tfidf_inverted_idx: This method receives the inverted frequency index dictionary and convert frequency in each tuple into tf-idf score.

Make_document_length: This method use tf-idf vector of each document and calculate the length of each document used for cosine normalization and will be stored as a dictionary.

Create_shelve_file: This method store the inverted index into a shelve file so that user don't have to build index for query purpose each time they want to make a query

b. Modules in Vs_search:

Get_title_text(): This method receives the document id and return text and title of the document as a string.

Make_query_tfidf(): This method take query string and process it including removing stop words etc., then calculate the tf-idf and return it as a dictionary.

Get_cosine_similarity(): This method take the query's tf-idf vector, get posting list associated with each term and calculate the cosine similarity score for each document which contains at least one word in the query.

Get_k_largest_docid(): This method receive the dictionary of each documents cosines similariry score and use Heapq in python to get the k documents with largest cosine similarity score.

Get_movie_snippet: This method will return a title and a snippet of the movie text for the results page.

Modules in Boolean_query:

Query: For top level route ("/"), simply present a query page.

Result: This method takes the form data produced by submitting a query page request and returns a page displaying result if movies match all the query words can be found. Otherwise if words occur in query but not in index, error page will be returned

Movie_data: Given the doc_id for a film, present the title and text for the movie.

Testing:
In order to test the system, I built a test json file which only contain 10 movies in it and built the inverted index based on the test corpus data. One movie in my test corpus is "3 Gante 30 Dina 30 Second", I will get this movie in the first place if search the title of this movie. Also, the cosine similarity is much higher than other results. And the second result is still not bad which have matched words "30"and "Dina"

But if I put "Varanasi" to search movie, then it shows the unknown words "Varanasi" and total hits is 0 since this word does not occur in my testing corpus. Then if I click the "More like this"button, other similar movies are listed in a descending order based on the cosine similarity score.

Shelving time:
45 seconds

Discussion:
Sometimes user may use synonyms to do the query. For example, word “car” is in the index but “auto” not. When user search “auto”, those documents which have word “car” will not be selected as target document based this word which is not good. So, to optimize it I think one solution may use the wordnet in nltk, when user search a word, also use word net to search whether their synonyms in the index.

Another problem will be the wrong spelling problem. According to what we learned, based on what user input in the query, we may provide a list of candidates and calculate the probability of each candidate word. If the confidence is high, just correct the wrong spelling word, otherwise provide some candidate options for the user.

And the third problem is that the running time for searching is kind of long if use entire text and title to find similar documents when user click “more like this” to find more similar documents. The structure or algorithm may could be optimized in the subsequent optimization work.