Assignment 3: Boolean Search Engine

Carmi Rothberg

2/27/18

Description

This program searches Wikipedia film pages for Boolean queries. Users may enter a multi-term query into a search bar to receive a list of links to relevant pages.

Dependencies

The submitted code is written for Python 2.7. Python code should run identically under OSX and Windows.

Required Python modules are `json`, `shelve`, `nltk`, `flask`, and `timeit`.

Build Instructions

The inverted indices used for searching may be constructed using the `indexer.py` module. To build an index, create a new instance of the `NewIndex` class, as follows:

```
NewIndex('<corpus name>.json', <name to output>.dbm')
```

Four index models are available: two indices based on the shared class corpus `corpus.json` (one positional and one non-positional), and two based on my own corpus (`2017_movies.json`) created in the previous assignment. I have found the best results using the positional index from my own corpus — that is, the text is cleaner and line breaks and Unicode are preserved. However, for searches in specific fields, the shared class corpus may yield better results.

Each index is built using NLTK's tokenization and stemming tools. For tokenization, I use `word_tokenize`. For stemming, I settled on the Porter stemmer in `nltk.stem`.

Run Instructions

Setup

As mentioned above, four index shelves are available. The default index is `index_rothberg_positional.dbm`, which is loaded at the start of each session. To switch to a different index, simply edit line 14 of `boolean_search.py` and change `index = ShelvedIndex('index_rothberg_positional.dbm')` to `ShelvedIndex('<your index name>.dbm')`.

Searches

This program can be run in Python using the file `boolean_query.py`. The resulting search page can be found at the url http://127.0.0.1:5000/.

To search on the page, type a query into the search bar at the top of the page. This will direct you to a results page, from which you can either search new queries or click into document display pages.

By default, searches are conjunctive and exclude stopwords, which are defined as words whose stems occur in more than 1/3 of the documents.

For disjunctive searches, precede the query with the keyword `OR:`, as in `OR: castle palace`. This will produce results for all documents containing either the term castle or the term palace (as well as any variations, such as "castles" or "palaces"). Disjunctive searches also exclude stopwords.

For phrase searches, use the keyword `PHRASE:`, as in `PHRASE: beauty and the beast`. This will produce results for all documents containing the full phrase "beauty and the beast" (as well as word-variations with the same ordering, such as "beauties and the beastly"). Phrase searches include stopwords. This function still isn't working perfectly and doesn't display all documents with matching phrases.

Preceding a query with `AND:` is an alternative to the default conjunctive search but will yield the same results.

Results

If a positional index is used, results are ranked by term frequency. If a non-positional index is used, results are ranked alphabetically.

Each result shows a snippet consisting of the sentence in the document that contains the most instances of any of the query terms. Snippets are capped at 100 words.

Modules

`boolean_query.py`

The main module opens the search page and directs the user to the appropriate pages.

The `query` function opens the main query page.

The `results` function calls helper functions from the `boolean_search` module to process a query, extracting stopwords and unknown terms and directing to either an error page or a results page. This function also finds the number of hits for a search and calls a function to find result snippets.

The `movie_data` function renders the full data display page given a film's ID.

`boolean_search.py`

This file contains methods to open a shelved index, as well as to search it for query terms.

The `search` method finds movie IDs matching a given query. If possible, it sorts those IDs by each document's term frequency in order to provide ranked results.

The `get_unknown_terms` function returns query terms that are not present in the index.

The `parse_query` function looks at the beginning of a query for query-type keywords, and calls the `remove_stopwords` function to split a query into stopwords and non-stopwords.

`get_movie_data` uses the shelved corpus to look up title, text, and other fields given a document ID.

`get_hits` is a function to find term frequency in a given string.

`Movie_snippet` splits a document into sentences using NLTK's `sent_tokenize`, then finds the sentence in the document containing the most term uses using `get_hits`.

`boolean_index.py`

This is where the actual methods for creating and searching an index are.

The `Index` class contains methods to access the index, as well as AND, OR, and PHRASE methods that apply the `search_tools` module iteratively to multi-term queries.

The `NewIndex` class tokenizes each document using NLTK's `word_tokenize`, stems using the NLTK Porter stemmer, and then adds the document ID to the index for each stemmed term. Both positional and non-positional indices store document IDs as nested lists for consistency, though such storage is only necessary for positional indices.

After an index is created, stopwords are identified and added to the `Index` object as a set. Stopwords are defined as words that occur in more than 1/3 of the corpus documents.

Finally, the new index is shelved.

`search_tools.py`

This module contains methods for conjunction and disjunction.

The `intersect` method intersects two postings lists, in the form of term->[[docID, [position1, position2]], [docID, [position1]]], etc.. Intersected lists maintain each term's position lists. If two terms share a docID, the terms' position lists stay separate in the combined postings list. For instance, the intersection of

Term1 -> [[doc1, [position1, position3]], [doc2, [position1, position3]]]

and

Term2 -> [[doc1, [position2]], [doc3, [position1, position3]]]

is

Term1 AND Term2 -> [[doc1, [position1, position3], [position2]]

The `union` method takes two similarly structured postings lists and combines them. Again, if multiple terms occur in the same document, their position lists are maintained separately.

Testing:

A test corpus, along with the shelved test index, is included. It is comprised of 6 manually created documents.

In the test corpus, searching “man who produces films” should result in “The Disaster Artist” (which contains the words “man who is a film producer”), and searching “lady bird” should result in “Lady Bird” but not “Beauty and the Beast” or “Wonder Woman”, as the second two only contain one of the two search terms.