CSI 4107 Search Engine

Vanilla System

Feb. 7th 2020

Group 15

Armin Zolfaghari
8715116

This project is already deployed on Heroku, so there is no need to compile the project. The project utilizes Django to create a web application where the scripts folder pregenerates processed data, index and dictionary that the project will utilize while live.

https://one-search.herokuapp.com

Note that Heroku puts the server to sleep after 30 minutes of inactivity, so first accessing the site may take up to 20 seconds.

User Interface (Module 1)

This project leverages Django's framework to create a simple user interface using html. These views are all controlled through *engine/views.py* with their respective HTML templates being located in the *templates* folder.

Data is passed in through the url with a GET request, then it is processed in the view and a fully generated template is returned to the browser. The individual urls for documents are specified in *onesearch/urls.py* and a custom template tag located in *engine/templatetags/correction_tags.py* handles the replacement of words in the request for spelling correction.

There are 3 views currently available: one main view for the search query and model selection, one to view the results of the query with a brief description (max 100 characters) and spelling correction if available, and a document view to view the document in full.

The UI uses Materialize CSS to create nicer templates.

Pre-Processing (Module 2)

This project uses an independant python script *scripts/preprocess.py* to filter the uOttawa Courses page (UofO_courses.html) and stores it in *preprocessed.json* in the root directory. This requires for the html file to be present in the root directory, and will not run if the generated *preprocessed.json* file is already made.

This does this using BeautifulSoup to parse through the html code, splits individual course by identifying div tags with class "courseblock", then grabs the title from class "courseblocktitle", and description from "courseblockdesc".

```
<div class="courseblock">
```

```
      <p class="courseblocktitle noindent"><strong>PSY 7190 Seminars in
Psychology II (3 units)</strong></p>
      <p class="courseblockdesc noindent">Selected topics on contemporary
psychology presented and discussed as graduate seminars.</p>
      <p class="courseblockextra noindent"><strong>Course Component:
</strong>Lecture</p>
</div>
```

It will then store its title, description, and auto generate a document id based on the order it is found in.

This automatically filters french courses by ignoring any classes where the second digit of the course number is greater than or equal to 5 (e.g. ADM 2703 is a french course as indicated by the 7)

- There are a few courses that are bilingual and therefore still included in the preprocessed data (e.g. PSY 5023, PSY 6002 etc.)

This script can be run by activating the virtual environment with all the dependencies installed and running *python scripts/preprocess.py*

The JSON file is created and an ID is generated based on its collection and the order it appeared in

```
{
      "UO_347": {
            "title": "PSY 7190 Seminars in Psychology II (3 units)",
            "description": "Selected topics on contemporary psychology
presented and discussed as graduate seminars."
      }
}
```

Initially, these were loaded into Django's MySQL database to be retrieved, but this caused issues with VSM as it would automatically sort the array of ID's requested thus taking away order of relevance. Because of this I migrated to just pulling the data straight from the json file but it caused issues with the merge-sort algorithms for boolean retrieval.

This is because the IDs were initially just numbers, but they were stored in json as strings. Though these appeared to be sorted by default, they were naturally sorted not sorted by python standards as python thinks "2" > "100" when they are strings. The merge-sort algorithms required pre-sorted ids to be given as parameters and the indexing kept the same order to remain sorted.

At first I attempted just sorting by ID, but python would convert them to numbers then sort which did not change the order. So the ids were changed to “UO_#” so that python would sort them as a string, then they would be presorted in the index, and therefore when they are given as parameters in the merge-sort function.

Building the Dictionary and Index (Module 3, 4, 8a)

These modules were joined into one script *scripts/dict+index.py* as both the dictionary and index already required going through *preprocessed.json*

The script takes arguments for not using stemming, normalization and stopword removal (*python process/build_dict.py --help* to see the arguments) but these are all enabled by default. Similar to module 2 this can be run in the same manner by activating the virtual environment and running *python scripts/build_dict.py*, and again will generate the files in the root directory.

- Case folding is already done by default with the NLTK stemming module
- One interesting observation is that the NLTK tokenization stores commas, brackets etc.. as an individual words which may have adverse results

It will output 4 different files:

First is the dictionary file *dictionary.json*, which stores all the processed/cleaned words in a list within json that is easily readable by python.

```
[
      "adm",
      ...
]
```

Next is *index.json* which is a simple json file with the inverted index. It stores which documents each word is located as well as its frequency, tf-idf, and idf. Though the order of words was sorted in the index in class, there was no use doing that as python treats dictionaries as hashmaps and therefore has an average retrieval time of O(1)

```
"adm": {
      "documents": [
            {
                  "doc_id": "UO_0",
                  "frequency": 1,
                  "tf-idf": 0.49633151327271724
            }
      ],
      "idf": 0.49633151327271724
```

```
}
```

It will also store *settings.json* which stores the settings used to create the dictionary so the engine can clean the query in the same way and *raw_dictionary.json* which stores all words and their frequency without any cleaning except case folding (for spelling correction module).

Corpus Access (Module 5)

Data gets accessed directly from the index to be displayed in the *search_results* function within *engine/views.py* then getting passed into the view. Since the json will be ingested by Python as a dictionary, the document retrieval should be on average in constant time (O(1)).

Boolean Retrieval Model (Module 6)

The boolean model is available within *engine/boolean.py*, and it processes the query by changing the string to an postfix stack, then processing each word individually by converting it to its list of documents.The function *boolean_search* is the main function which takes in the query string, and return the list of documents to the view.

Wildcard Management (Module 7)

Also located within *engine/boolean.py*, the *handle_wildcard* function splits a word into its elements, then uses their bigrams to find common documents. Any word with a wildcard will be passed into that function to return a list of ids for documents with those words.

The function also cleans the word regardless of it containing a wildcard, so if stemming is on, the word with a wildcard will also be stemmed.

For example co*te*r - > [co, te, r] which will then search each bigram (co -> $c, co | te -> te | r -> r$)

An independent script *scripts/bigrams.py* is used to generate a *bigrams.json* file with all the bigrams and their relationship to words in the dictionary.

Vector Space Model (Module 8)

As mentioned prior, the calculations for tf, idf, and tf-idf were already done and stored in the index. The retrieval model would be located in `engine/vsm.py`, where the top 15 documents are found and returned based on the query inputted.

Spelling Correction (Module 9)

This is located in *engine/spelling_correction.py*. Any words that had no corresponding documents would be passed into the function *correction* to be evaluated. If it is a stopword or less than 4 characters it would be ignored.

When searching for words to evaluate weighted edit distance, it only considers words in the raw dictionary that has a length of +/- 3 characters. It then calculates weighted edit distances, and only keeps words with a distance of less than 5, then sorts those words by raw frequency.

When calculating weighted edit distance, I thought of this article and put less weight on insertion and replacement of vowels (weight of 1), a weight of 2 for substitution of consonants, and a weight of 3 for everything else (insertion/deletion of consonants...)

Any word that has no value in the dictionary (after cleaning), will have the raw word passed into this function, and it will return the top 3 results if found. Since there is no cleaning done on the returned words, you may see correction suggestions for multiple words that will lead to the same list of documents. For example, the spell correction could suggest computer and computers, which after stemming would both be comput resulting in the same list of documents.

Example: If compoters is entered, it'll prompt to return for computer, computers, compilers