CSCI 6370: Information Retrieval Friday June 19, 2015

Search Engine Phase One

**Objective:** To build a web crawler that can visit a web page, analyze its links and recursively follow them with the intention of download up to 100 html/text documents to a folder called document\_corpus which will in turn be analyzed at a second phase and build an inverted index for fast document retrieval.

In addition in this initial phase we also present a prototype for the user interface (UI) for this search engine.

**Team and Search Engine name**: Boom

|  |  |  |
| --- | --- | --- |
| **Team member** | **Student ID** | **Role** |
| Alex Campos | 10234425 | Python Developer |
| Alvaro Leal |  | Code Reviewer, Testing and Design Improvements |
| Divya Vuppala | 20331466 | Technology research – Python Libraries and Framework |
| Akhil Choppadandi |  | Functional Testing of search engine |

**Description and Accomplishments**:

This initial phase of the search engine consists of a simple web crawler which is used to build a document corpus of 100 files and a basic front-end HTML web form that accepts user’s queries and displays the query back to the user without really searching for any of the terms in the document corpus yet.

As one can imagine, the core of this phase is our web crawler which was written in python due to its simplicity and the massive amount of networking/web libraries readily available. In our web crawler for instance, we are using the library “***Requests***: HTTP for Humans”[[1]](#endnote-1) which is used to generate HTTP requests to our web server, analyze the HTTP response status, inspect the headers and convert the request into a string object. Another interesting library that we are using is called “Beautiful Soup 4”[[2]](#endnote-2) which we are using to parse the HTML file and find all the hyper-links (<a href=<http://.....>>) . On top of these previously mentioned libraries, we are also using one called “urllib.parse”[[3]](#endnote-3) which takes care of parsing a URI and extract the different components like protocol, domain name, path, query string, etc.

All of these libraries are working in synchrony to create a very simple web crawler which follows the following logic (assumes the initial URL is accessible and can be downloaded):

1. Given a URL, try to go to this resource via an HTTP request.
2. If we are able to download this page, we remember that we visited it and proceed to write it to disk and parse it to find all of the relevant hyper-links on this page. If we fail to download/write to disk we mark it as a bad URL.
3. For every link found that we have not already visited or that is not a bad link go to step #1 until you reach 100 downloaded documents
4. If we are not able to download a page because it is not text or the web server returned an error message, we flag this page as bad and continue to the next hyper link as needed.

Other features available on this phase are as follow:

1. **Restricted domain crawling:** this web crawler also takes into account the concept of “allowed\_domain” which is quite useful to prevent the crawler to follow links outside of a controlled environment.
2. **Support for absolute and relative urls:** When parsing an HTML document, it is very common to find relative links to other documents served from the same web server. This web crawler keeps track of the current web page and builds an absolute URL if needed. This increases the success rate of documents fetched.
3. **Download files based on Content-Type header**: Since our document corpus will be based on text data, the crawler ignores pdf files, images and any other link that is not strictly text data.

So far this web crawler is able to download the required 100 pages with no problems however, since we are using the filename found in the URL if we visit 2 or more files with the same name one of them might be overwritten. We will fix this in the next phase by writing each file based on the document id and keeping a separate table that holds the document id, URL, text snippet and other useful metrics.

Here is a sample screenshot of the web crawler in action:

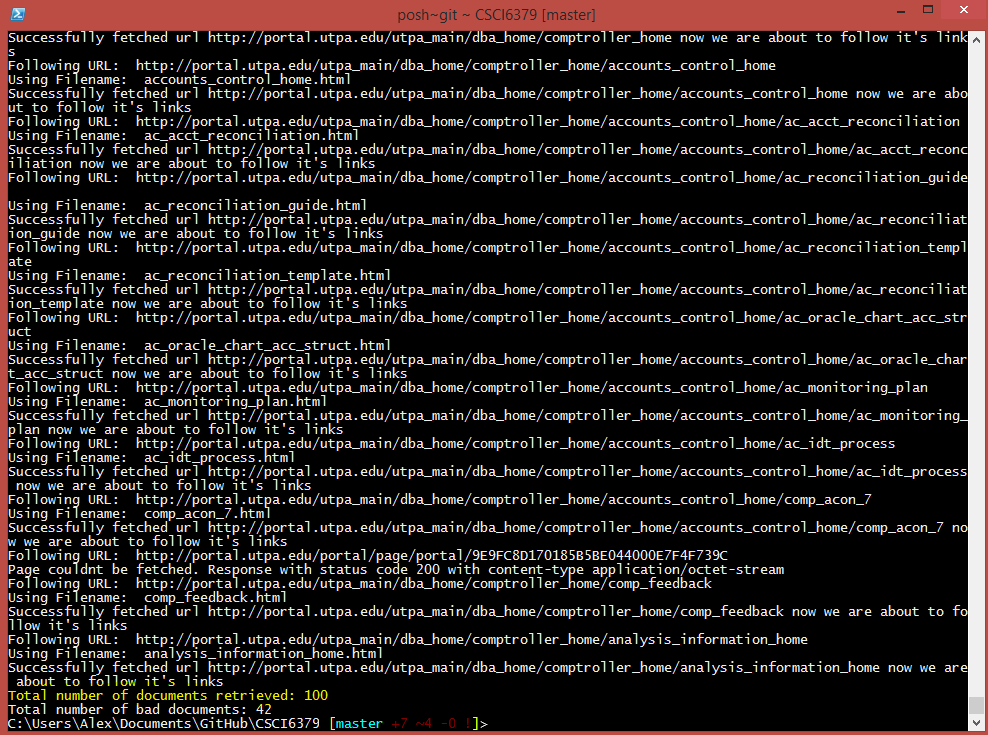


Figure : Sample run of the webcrawler. At the end of the run, it displays a summary of how many links it visited and how many URLS were flagged as bad (page not found or not a text/html file was found)

As per the search engine front end, as it was previously mentioned in the opening paragraph, it consists of a simple web site running in Django which presents an HTML form and allows the user to enter a query string which is then parsed and returned by the web server.

The way this works, is that one first has to run the Django web server (python3 manage.py runserver) which will in turn listen on port 8000 by default. Once it’s up and running, all that is left to do is open a web browser and go to <http://localhost:8000> (see screenshot)

The plan is that on phase two, we will take this input and following the vector space model convert it into a query vector that we can use to find keywords on our inverted index.

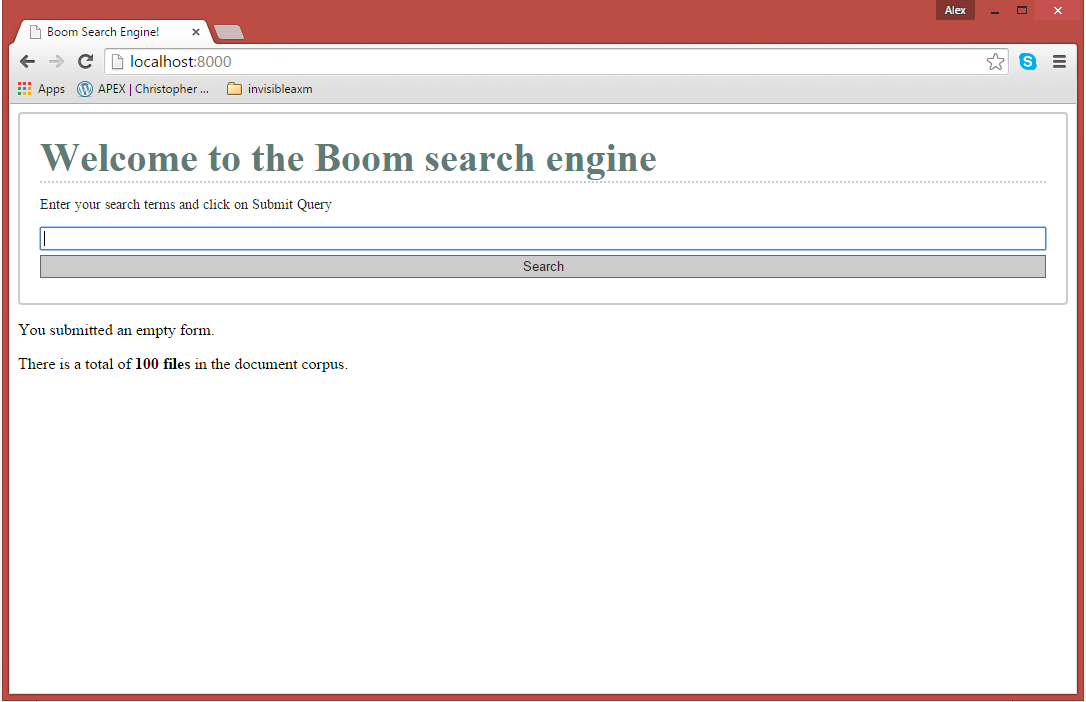


Figure : Web page displayed when user goes to http://localhost:80000

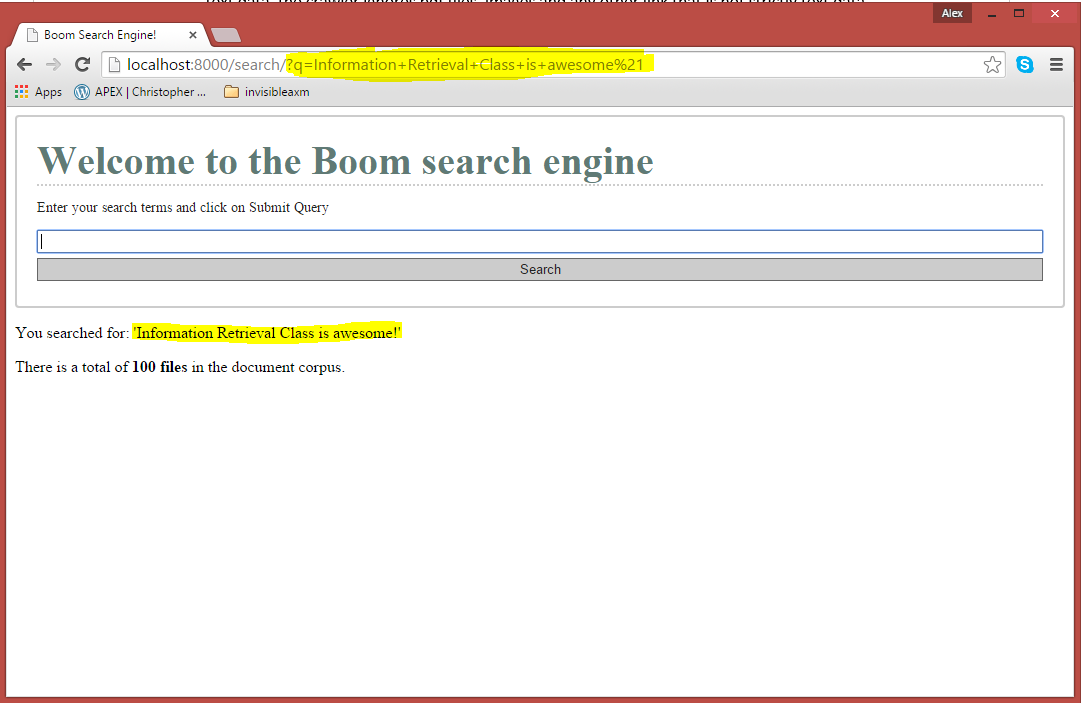


Figure : After the user enters "Information Retrieval Class is awesome!" the web server processes the query and it outputs it right back to the client

1. http://docs.python-requests.org/en/latest/ [↑](#endnote-ref-1)
2. http://www.crummy.com/software/BeautifulSoup/bs4/doc/ [↑](#endnote-ref-2)
3. https://docs.python.org/3/library/urllib.parse.html#module-urllib.parse [↑](#endnote-ref-3)