OER Common Search Engine

IST 441 - Custom Search Engine Project Final Report

Yuya Jeremy Ong
Penn State University
yjo5006@psu.com

Nick Donaghy
Penn State University
njd5253@psu.edu

Dana Shalit
Penn State University
dis5526@psu.edu

Wyatt Naftulin
Penn State University
wjn5051@psu.edu

1 INTRODUCTION

In the 21st century, access to the World Wide Web has granted millions of people to a whole trove of information and content. This open an opportunity for a wealth of information, knowledge, and wisdom to be transfered and shared across many people, helping many people learn new skills and gain valuable information. Furthermore, with the recent efforts by volunteers, university and educational institutions, and eager students, opportunities to make this learning experience free and without any strings to for-profit publishing companies have made a tremendous progress in streamlining high quality content and educational material for those that do not have the full funding to complete the necessary educational career through MOOCs and open-source educational education.

One exemplar of this movement is organized by the OER (Open Educational Resource) Movement which aims to push the human rights for people to have the right to a proper set of educational resources through the use of technology as an engaging medium to enhance the overall quality of content provided to students.

The OER organization as a result founded the website OER Commons [4], a community driven site comprising of a database system which allows students, teachers, professors, faculty, and volunteer members to post, share, organize, and maintain a whole repository of websites, documents, multimedia, and educational materials which can be adopted to anyone's educational curriculum system - a very similar mission to Wikipedia based organization for educational materials.

For this project, we worked with Assistant Teaching Professor, David Fusco under the College of Information Sciences and Technology, who specializes in the development and design of enterprise architecture systems. Professor Fusco claims to be an active user of the OER Commons resources repository as he often make use of the content which has been provided there. He believes in the mission of the OER Commons and strongly hoped that through developing an improved search engine system tailored for the content of the website, we can allow better access to the material and content provided by the OER Commons - which in turn would improve the overall experience for students to find relevant material and content faster and much more efficiently. Thus, he requested for this project to develop a custom search engine which indexes the material and content provided by the open OER Commons.

In this report, we present our entire process for developing a custom search engine and all of its constituent parts including the seed crawler, content crawler, extraction pipeline, text-preprocessing engine, as well as the indexing pipeline and the front-end component which users would use to interface with our custom search engine. In this paper we first describe the original OER Commons' search engine system, then a version of the Google Custom Search Engine which indexed the OER Commons' website. Furthermore, we then introduce the system architecture of our custom search engine with each of the corresponding sections describing the development and design choices for each of the corresponding components of our custom information retrieval system. Prior to describing our system, we then look into the evaluation metrics and results of our custom search engine from both an empirical and quantitative metrics for evaluating our system. We then follow our development and evaluation descriptions with a discussion which includes feedback and comments from our customer David Fusco, followed by our own reflection on the process and improvement we could have made to the system.

The source code for this entire project can be found on our Github repository: https://github.com/yutarochan/IST441.

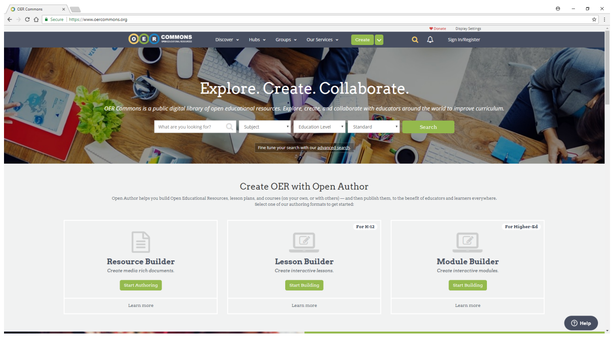
2 RELATED WORK

In this section we describe and evaluate two systems: the OER Commons Website's original search function and the Google Custom Search Engine indexed on the OER Website. We describe the underlying system and evaluate some of their corresponding features as well as some of the key problems and issues we have found in their search engine that we attempt to address in our custom search engine.

2.1 OER Commons Search Engine

The following section will describe and evaluate our existing platform based on the original website we are attempting to improve - the OER Commons website (http://www.oercommons.org/) [4]. Within the OER Commons website, the site already provides users with a search functionality which allows users to quickly navigate through their content as shown in **Figure 1**.

The search engine, unlike a traditional search engine system, provides users with a bit more control over the type of content they want the system to show based on a drop down filtration process notably by subject type, educational level, and educational standards. The design of this form of a search system provides us with a very strong suspicion that the system relies on a Relational-Databased based system for querying content from their repository and does not utilize a index-based information retrieval system. Looking further into their search function, we find that they also provide an Advanced Search feature which provides users with



OER Commons is a public digital library of open educational resources. Explore, create, and collaborate with educators around the world to improve curriculum.

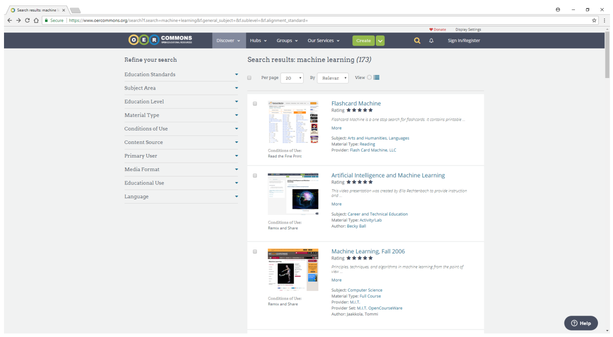
Fine tune your search with our advanced search.

Open Author helps you build Open Educational Resources, lesson plans, and courses (on your own, or with others) — and then publish them, to the benefit of educators and learners everywhere. Select one of our authoring formats to get started:

Figure 1: OER Commons Homepage

even finer controls of the type of content they want to search for. The functionalities of the advance search from a user experience stand-point forces users to provide effort in their end to manually refine and fine-tune their search criteria.

Providing the search engine with the query, as shown in **Figure 2**, the results page provides users with a typical search result interface, displaying a set number of results per page with each result providing a title of the content (most likely provided by the curator of the content), a short description of the content, the rating of the page, as well as other critical meta-data corresponding to the content. The options for controlling parameters for the resulting number of content to list as well as the criteria order for how the content is displayed. Interestingly, the content the default setting for how the content is displayed is based on the Title or the alphabetical ordering of the submission of the content title and not the relevance - which indicates that the relevance criteria to be very hard to define accordingly to the user or students of the page.



Search results: machine learning *(173)*

Flashcard Machine
Rating ★★★★★
Flashcard Machine is a one stop search for flashcards. It contains printable ...
Subject: Arts and Humanities, Languages
Material Type: Reading
Provider: Flash Card Machine, LLC
Conditions of Use: Read the Fine Print

Artificial Intelligence and Machine Learning
Rating ★★★★★
This video presentation was created by Ella Rechtenbach to provide instruction and ...
Subject: Career and Technical Education
Material Type: Activity/Lab
Author: Becky Ball
Conditions of Use: Remix and Share

Machine Learning, Fall 2006
Rating ★★★★★
Principles, techniques, and algorithms in machine learning from the point of view ...
Subject: Computer Science
Material Type: Full Course
Provider: M.I.T.
Provider Set: M.I.T. OpenCourseWare
Author: Jaakkola, Tommi
Conditions of Use: Remix and Share

Figure 2: OER Commons Query Results Page

To then access the content directly, the user then can click the title of the corresponding page which will lead them to another page which provides a full disclosure of the meta-data submitted by the curator of the content. The page includes a large green button, "View Resources" which then redirects the users to the content which is wrapped on a page with the OER Commons header and the corresponding page wrapped in an iFrame window. As demonstrated here, one of the major caveat to the system from a user experience standpoint is the multiple steps and the intricate level of steps required for the user to be able to find. Furthermore, they would have to click through various levels of links in order to finally reach the actual content students want access to.

Speculating towards the backend of their system, we have mentioned previously that the site is most likely utilizing a databased driven query mechanism to mitigate their search and retrieval process for the content - mostly based on a keyword and manually annotated indexing mechanism to curate the indexing process for their search results. Looking further into their back-end system, we find that the OER Commons Organization may potentially be making use of a custom defined markup language which defines the metadata of each of the content corresponding to each of the material being submitted to their page. In particular candidate technologies that they may have been utilizing includes works from the Dublin Core Metadata Initiative (DCMI) [3], Learning Objective Metadata (LMO) - based on an XML data schema, and RSS (Rich Site Summary). However, these assumptions of the backend system they are utilizing are purely from a speculative point-of-view based on the interface and provided input interface of the system, which drew us to those conclusions.

Given the above assumptions and observations, we looked into the attributes and the empirical results of the search results provided by the search engine. For this evaluation process, we evaluated based on some typical search topics ranging from very general terms to domain specific topics such as "science", "biology", "civil war", "information retrieval". In these cases, we have empirically noticed that for much more general and high-level topics, the search results are often exhibit a qualitatively high recall and low precision - which is obviously easy to attain for a relational-databased query based system which relies on a keyword-based search criteria.

Given that such system is based on a keyword-driven system, we can then infer that the handling of semantics and pragmatics with regards to the search terms are very weak. To test this hypothesis, we came up with various terms which may have multiple meaning dependent on their context - such as the word "bat". The word may refer to the biological mammal or the sports equipment used in baseball which makes this a very interesting test to see how well it handles and ranks the pages accordingly when we evaluate the search engine. Although, the query results page provides both relevant topics for the biological and sports equipment, the corresponding relevant ordering of the search results appear to be very arbitrary in nature.

Furthermore, extending from the given assumption that it does not use a index-based retrieval system search engine for its platform, we find that the organization does not further crawl deeper into the content of the page. We can suspect that they only provide results to the content submitted by the community and nothing further than that. On one hand, this is a very good method for the OER Community have tighter controls over the overall quality controls over their content as they restrict the domain of their documents to pre-approved sites and materials by both experts and students - which should be a better strategy for improving the relevancy with respect to the precision of the search engine. However, on the other hand, this looses the opportunity for the OER Commons to provide a higher volume of potential content for students to have access to. In this respect, hand curating content manually from the Internet is not a very scalable solution with the ever increasing amount of material that is currently available today.

In short, although the OER Common provides a very adequate enough search tool for their content collection, we believe that this can be further improved with the index-based information retrieval system. The challenge here however will be to ensure that we balance the precision and recall of our search engine against the OER Commons site and to define a better sense of the relevancy for the content being queried by the user.

2.2 Google Custom Search Engine

As part of our project, we have also made use of the Google Custom Search Engine, which is based on Google's search infrastructure to help users easily build custom search engines tailored towards select content given a set of seed sites that Google would crawl and index over. Furthermore, this also provides us options to limit the type of files, content, and other specific schema depending on the nature of content we want the search engine to return.

In our Google custom search engine [1], the only relevant parameter we were able to provide was the OER Commons URL. Initially, ideas of having Google index the URLs from the OER Commons website, which we will further explain in the following sections, were considered. However, based on reports from other attempts and from other implementations of the Google Custom Search Engine, we found that there was an upper limit as to how many URLs you can provide to the Google Custom Search Engine. Therefore, the number of parameters we are able to consider is very limited - hence we also have suspected that the results that the Google Custom Search Engine to be very limiting.

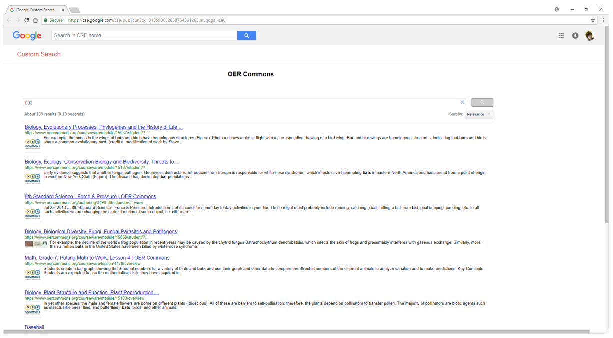
After initializing our Google Custom Search Engine, we performed various empirical evaluations against some similar search terms we provided to the OER Commons Search Engine in the previous section for a sample empirical basis for our comparison. Through our exploration, we have found that the Google Custom Search Engine handles pragmatics and semantics in a much more relevant fashion as opposed to the OER Commons' search engine. For instance, with the search results with respect to "bat" shows within the first few top results having a much higher concentration of the biological animal than the baseball equipment - which is well separated and pushed further down as shown in **Figure 3**. We speculate that this behavior occurs due to the overall distribution of the biological topic than more so the sports topic present within the OER Commons website.

Although the Google Custom Search Engine appears to be performing well from an empirical standpoint with respect to content ranking, the search engine has one interesting behavior that we found to be common within the results of all search queries. While the search engine does sometimes provide corresponding content and direct links to certain pages that it has crawled from the website, the search results also includes links to the search query results page of the OER Commons page. More concretely, the results of the URL provided to Google point to the corresponding search term (either exactly or semantically related) as part of the URL of the OER Commons' search engine.

For example with the search query "bat", one of the results that might be returned would be <URL>/browse?f.keyword=baseball,

[1]https:cse.google.comcsepublicurl?cx=015590652858754561265:mvqqgs_-zeu

Figure 3: Google Custom Search Engine Sample Query



Google Custom Search

Secure | https://cse.google.com/cse/publicurl?cx=015590652858754561265:mvqqgs_-zeu

Search in CSE home

Custom Search

OER Commons

bat

About 109 results (0.19 seconds)

Sort by: Relevance

Biology, Evolutionary Processes, Phylogenies and the History of Life ...
https://www.oercommons.org/courseware/module/15037/student/?...
For example, the bones in the wings of **bats** and birds have homologous structures (Figure). Photo a shows a bird in flight with a corresponding drawing of a bird wing. **Bat** and bird wings are homologous structures, indicating that **bats** and birds share a common evolutionary past. (credit a: modification of work by Steve ...

Biology, Ecology, Conservation Biology and Biodiversity, Threats to ...
https://www.oercommons.org/courseware/module/15187/student/?...
Early evidence suggests that another fungal pathogen, Geomyces destructans, introduced from Europe is responsible for white-nose syndrome , which infects cave-hibernating **bats** in eastern North America and has spread from a point of origin in western New York State (Figure). The disease has decimated **bat** populations ...

8th Standard Science - Force & Pressure | OER Commons
https://www.oercommons.org/authoring/3490-8th-standard.../view
Jul 23, 2013 ... 8th Standard Science - Force & Pressure. Introduction. Let us consider some day to day activities in your life. These might most probably include running, catching a ball, hitting a ball from **bat**, goal keeping, jumping, etc. In all such activities we are changing the state of motion of some object, i.e. either an ...

Biology, Biological Diversity, Fungi, Fungal Parasites and Pathogens
https://www.oercommons.org/courseware/module/15059/student/?...
For example, the decline of the world's frog population in recent years may be caused by the chytrid fungus Batrachochytrium dendrobatidis, which infects the skin of frogs and presumably interferes with gaseous exchange. Similarly, more than a million **bats** in the United States have been killed by white-nose syndrome, ...

Math, Grade 7, Putting Math to Work, Lesson 4 | OER Commons
https://www.oercommons.org/courseware/lesson/4478/overview
Students create a bar graph showing the Strouhal numbers for a variety of birds and **bats** and use their graph and other data to compare the Strouhal numbers of the different animals to analyze variation and to make predictions. Key Concepts. Students are expected to use the mathematical skills they have acquired in ...

Biology, Plant Structure and Function, Plant Reproduction ...
https://www.oercommons.org/courseware/module/15103/overview
In yet other species, the male and female flowers are borne on different plants (dioecious). All of these are barriers to self-pollination; therefore, the plants depend on pollinators to transfer pollen. The majority of pollinators are biotic agents such as insects (like bees, flies, and butterflies), **bats**, birds, and other animals.

Baseball

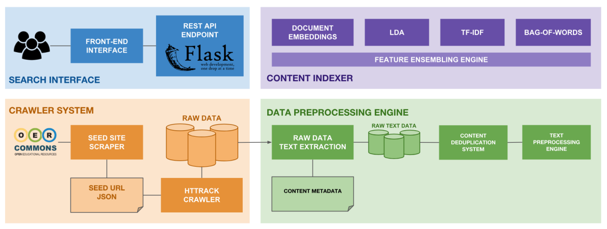
where the term "baseball" is passed into the f.keyword GET parameter of the URL. Interestingly we find that Google's search crawler does employ an interesting strategy to exploit this into further finding content in a efficient manner and is able to better improve the precision of the custom search engine. However, this still poses the same issue as the OER Commons' website as the user would still have to click even deeper in order to access the relevant content as they would have to look first into the result provided by the Google Custom Search Engine, then the OER Commons' search results, then click two more times to finally get to their desired content.

From this, we find that to some respects using the Google Custom Search Engine as a basis for a proper evaluation comparison against the OER Commons and the custom search engine we have developed may potentially not be adequate as the heuristic for determining whether we should include the pages from within the search query content is considered to be relevant would be up to debate depending on the customer.

3 SYSTEM ARCHITECTURE

In this section we present the system architecture we have developed for the OER Commons Search Engine and all of its corresponding components at a high level as shown in **Figure 4**. The following sections following this section will be based on describing the detailed implementation and design choices for which we have selected during the development process.

Figure 4: OER Commons Search Engine System Architecture



First, the entire system architecture is divided into four major components: the crawler system, the data preprocessing engine, the content indexer, and the front-end component. In our system, we have developed each system as a self-contained system which can be scaled across both vertically and horizontally and deployed based on whether more content needs to be crawled, indexed, and updated for future additions within the OER Commons website.

The data processing pipeline for system starts with the crawler extracting the base seed URLs from the OER Commons website, then subsequently crawling the seed URLs and persisting the corresponding raw data on disk. Following the crawling process, we then perform a raw text extraction process from the raw files to a raw text file using a custom content extraction system we have developed.

Concurrently as we extract the content from the pages, we also generate a content meta data generator which provides the necessary data we will be later using for rendering the front-end content such as the URL and page title. Additionally, we have also routines for content de-duplication to reduce the number of content that the indexer has to process over to reduce the number of documents to process within a reasonable amount of time.

For indexing the content of the pages, we have implemented various feature extraction algorithms for our system. We have designed our search engine to be able to leverage both individual indexing strategies and ensemble methods to combine the results of the various vectorized models we have produced.

Finally, for our front-end interface, we have implemented a Flask based web-application [12] which executes both our search queries and renders the results of the search results on a front end interface we have developed for users to be able to perform search queries on. The application presents user with a very similar Google-like interface with a simple search bar and a results page rendered to take the user directly to the website of interest.

In this section we have briefly described the high-level details and the data process flow for our custom search engine. In the subsequent sections, we will describe in greater depth each component of our design choices and implementation for the search engine with regards to the algorithms, frameworks, and libraries we have used to build this search engine.

4 WEB CRAWLER

In this section, we first describe the most primary component of our search engine, the web crawler system. The web crawler system is primarily divided into two key phases: the seed crawler and the content crawler. In order to first have content for our page, we must have a basis for the list of URLs we must crawl.

4.1 Seed URL Crawler

For the OER Commons search engine, we have decided to use the entire collection of websites that the OER Commons has in our crawler. Within the OER Commons page, the site has an already in-built search engine as we have described in a previous section. Within the empirical analysis of the search engine's capabilities, we found that we can expose the entire list of URLs contained in the collection of the OER Commons' site by not entering any query.

As a result, at the time of scraping the site, the OER Commons contained 50,092 websites. In our seed URL scraper, we scraped relevant metadata and unique identifiers to extract the necessary content from the OER Commons Resource content. As we will describe in a later section, this unique identifier is a critical component in which will help us to extract the necessary seed URLs since the structure of the OER Commons' website is somewhat nested under a secondary level of redundancy to protect against scrapers.

To manipulate the results displayed from the OER Commons' search engine we analyzed some of the key GET parameters from the URL to see what we can adjust accordingly to minimize the number of page calls and maximize the number of content we can extract for each scraping process.

For the scraper, the base URL was https://www.oercommons.org/. Following the is a backslash are parameters we can control to display the necessary content we want to scrape off from the OER Commons' search engine page. The parameter after the backslash is "browse" followed by a "?" (question mark), which follows for GET based parameters that the website will use to control the search engine's visual parameters such as the number of pages to show at once and the result index to start showing from.

OER Common's URL structure is convenient for this search engine because we could manipulate the batch size and batch start after these components to alter the number of results per result page and which number result will start off the list of results on the page. Given the number of pages provided by the search result count, we computed the maximum number of pages that we can scrape with the highest efficiently to reduce the number of site calls we made in general.

We implemented a scraping routine in Python which manipulates the URL in this way until the maximum number of results have been loaded and their links have been extracted from the pageâĂŹs source code. Requests [10] is the name of the Python module that allows for us to access the access of the webpages provided by the OER Commons. In the robots.txt file, we have found that the site did not block crawler access which allowed us to safely perform the operation to find the relevant content.

To ensure that the requests library successfully downloads the page, we also ensured to use a proper User-Agent header to identify the scraper as a "browser" that is accessing a page. For our scraper we identified ourselves as a Chrome Browser which was running under a Unix based operating system (Mac OSX). To also respect the bandwidth load of the OER Commons' site, we have also place a one minute buffer time for every 500 requests we made to the page. This was a safeguard we implemented to ensure that our scraper's IP would not be banned from the OER Commons' page.

For extracting the relevant URL and corresponding metadata from our site, we utilized BeautifulSoup [2], a Python module for web crawling and parsing. Using this library, we analyzed the structure of the site and identified the corresponding code block which seeks out links to useful resources from the results page. As the structure of the codebase was repetitive in nature, we were easily able to identify the necessary structure to identify the corresponding html tag class attributes and extract the text content directly.

We subsequently persisted all of the meta data as a JSON file. Each entry contained the uid, title, abstract (both the short and long version), and any other metadata info which was encoded as "item-info", which contained things such as usage rights, institution names, and other arbitrary details pertaining to the resource. We persisted this data in the event that we want to filter our crawling mechanism based on certain criteria in the future depending on what the customer seeks.

However, one caveat to this system was the actual seed URLs that we required for the actual content of our pages. The actual URL of the page was not found on the search query results under the browse directories, and instead nested within the "courses" subsection of the site - which actually was blocked by robots.txt. We found that the designers of the site purposely nested their URL as an iframe when you place "/view" at the end of the URL. We believe that the OER performed this process to attempt to avoid scrapers or crawlers from having direct access to the content of the site. However, we regardlessly scraped the content and extracted the corresponding URLs through another routine which would utilize the uids we have found from our first pass through of the OER Commons' site. We then also subsequently appended the URLs we have extracted to our JSON file through an additional consolidation script.

4.2 Seed URL Crawl Results

In this section, we describe some the statistics from the seed crawler process we have described in our previous section. This section will be used as a short analysis of some of the key trends and observations we have noticed in our crawl data to mitigate for common trends and patterns that emerge within the OER Commons' community and their aggregation of sites they collect as a whole.

As previously mentioned, we have been able to successfully extract 50,092 seed URL links. The following analysis is based on the following URLs we have analyzed. Out of the 50,092 sites we have collected, we found that 35,292 sites utilized the HTTP scheme, while the other 14800 websites used the HTTPS protocol.

As seen in **Table 1**, we have found that the majority of the domains collected within the OER Commons' resource pool are based on a reliable source - mostly consisting of organizations and educational sites. We found this to be very interesting as this shows the overall content quality that OER Commons has to offer for its students and how much the community regulates upholds the content quality in a very strict manner.

Table 1: Top 5 URL Domains

Domains	Freq.
org	26,903
edu	10032
com	4987
gov	4805
ie	703

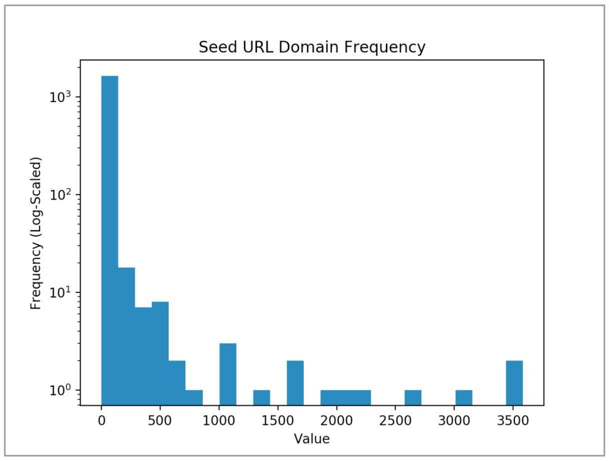
In **Table 2**, we present a table with the frequency count of pages, ranked by the top 10 domains scraped from the OER Commons' website. We can see that the top page, cnx.org, comes from another well known repository for open education, where they provide free textbook resources and other course materials which are all compiled by Rice University. Below, we also see other content such as that of GeogebraTube and Youtube and other media based content as an additional source of materials.

Figure 5, presents a frequency distribution of the number of sites each domain has. As shown it clearly displays properties of a zipf distribution with a majority of the URLs only provided to the OER Commons page having only very few pages per domain - however with the exception of the tail end values which have approximately 3000+ pages. Note, that the graph is scaled by a base-10 log along the frequency axis.

Table 2: Top 10 Domain Page Frequency

URL	Freq.
cnx	3582
geogebratube	3517
loc	2623
animaldiversity	2623
youtube	2216
carleton	2144
teachengineering	1630
ck12	1625
archive	1416

Figure 5: OER Commons Search Engine System Architecture



4.3 Content Crawler

To further increase the variety of the content apart from the content provided by the OER commons, we decided to utilize a web crawler to further explore the provided resources in order to further maximize the recall rate of the search engine. For this, we explored a variety of different crawler solutions including fscrawler (based on the Elasticsearch crawler) [5], Heritrix [6], Apache Nutch [1] and HTTrack[7].

However, out of all of the ones we have tried, we have found that HTTrack was the best crawler out of the different choices - primarily based on the ease of use, implementation and mostly the robustness and stability of the crawler utility itself. HTTrack is an open-sourced crawler that was developed by Xavier Roche. The software was originally intended as a utility for people to archive and mirror sites for archival purposes.

To facilitate the crawling process using HTTrack, we built the latest binary of HTTrack on a Linux based hardware cluster from source. HTTrack for the Linux operating system is provided as a command line interface containing many parameters that we are able to make use of to control the behavior of our crawler. Key parameters that we made use of for our crawler includes: preventing the crawler to crawl pages which contained a password prompt, the depth of the pages we crawled, number of simultaneous connections our crawler was able to have, saving only HTML or web based file types, downward traversal of the URL hierarchy, skipping of any mirroring prompts from the HTTrack daemon, and quite mode to cut off any logging and error reporting features.

However, one issue with the crawling process was being able to crawl all 50,092 seed URLs in a very scalable and efficient manner. To solve this, we needed to write our own asynchronous process to be able to orchestrate a whole crawl routine in a parallel manner. We implemented a crawl routine in Python which makes calls to the HTTrack binaries using a subroutine call and allocate them asynchronously through using Python's multiprocessing Pool function. By doing so, we can allocate 1 process per seed URL and crawl many pages simultaneously - which can certainly save a huge amount of time than performing this operation in a serial manner. To avoid overlapping of crawling the same domain space simultaneously, we ensured that the list of sites we crawled were initially randomized to avoid potential banning by the servers. This as a result has helped to finish crawling all seed URL pages within a time frame of approximately 8 hours.

5 CONTENT EXTRACTION

Content extraction for the specialized OER commons search engine was done by extracting raw text and generating a corresponding document metadata for each raw data crawled. We implemented a routine which performs document filetype analysis, metadata generation, and content extraction.

First, to ensure that the file type we are attempting to scrape is based on the correct type and to also filter any unwanted filetypes, we utilized the Magic library to help identify the file signatures based on the internal header data in the file. Given a file, we return a string based representation of the corresponding filetype based on the header.

Then for each of the corresponding approved document crawled, we generate a piece of associated metadata with the UUID, URL, the content title, and the filetype. The universally unique identifier (UUID) is a special document identifier which is used to normalize the naming convention of files. The URL is simply the original uniform resource locator of the scraped website. The title of the website is based on the title tag if one exists, if there is no title tag the URL is used as the title, or the filename of the document if it is not a website. All meta data is stored as a JSON file which will be used later in the final web server for quickly mapping the the results UUID to the corresponding content as a lookup index.

As for document preprocessing we base our extraction process based on the given filetype identified in the previous step. Depending on the identified file type, each of the document will go through a slightly different process. For the purpose of time constraint and demonstration, we have only focused on HTML based files - but we have implemented the rest of the features, but did not enable them. We defined the entire process as a single routine function which we then made use through also parallelizing the process as a multiprocessing step similar to that of the seed page crawler step using the multiprocessing library to speed up the process.

For raw text files such as .txt and .rtf files, we simply used the native Python I/O capabilities and did not do much as they were in the desired format. Our goal for this process is to extract the corresponding text and build an entire directory containing just text files named with the UUID of the document containing the extracted raw content.

For our system, we have also implemented an extraction process that handles any of the common Microsoft Office document types, which includes: Word (doc, docx), PowerPoint (ppt, pptx), and Excel (xls, xlsx). For this we utilized a Python library known as textract [11], which is a wrapper library which contains various other extraction libraries which helps to facilitate automated document text extraction very easily and abstracts the entire process as a single process function. For PDF files, we utilized the PDF extraction library GhostScript, which is an open-sourced interpreter for PostScript and PDF files.

One issue for raw text extraction from webpages is the amount of irrelevant text that is pulled from the site. A naive approach in extracting text may be to identify all of the corresponding texts and stripping off their corresponding HTML tags. However, across the many sites that we have seen, such as news sites, blogs, and most educational websites often include raw text coming from the header, navigational bar, and footer - which are all irrelevant to the actual content of interest which lies between those components.

To mitigate this issue, we utilized an open-sourced machine learning based library called Boilerpipe developed by KohlschÃijtter et. al [13] which utilizes two simple features, word counts and link density to build a stochastic model to predict whether a given content in the page is relevant or not. Their paper reports a very high accuracy based on empirical results and found it to be a very relevant and effective way to improve the precision of our model.

Upon experimentally trying Boilerpipe on various websites, we have noticed few key observations. Boilerpipe works very well for text-heavy and dense pages, which effectively works well in extracting the relevant text. However, for some sites where the internal content is very sparse or barely present, we find that Boilerpipe essentially returns nothing because it classified almost all the text as irrelevant. Based on these empirical observations, we proposed to use a filtering mechanism to delete or remove any pages that have very few to no content on the page using the following heuristic as defined:

$$\alpha < \frac{Char\ count\ from\ Boilerpipe}{Total\ Document\ Character\ Count}$$

where this value computes the overall ratio of the text and compare it against an a threshold parameter (α) to filter out any document that is below that threshold.

6 PAGE DE-DUPLICATION

In this section we describe another key issue that we attempt to address, which is based on how to reduce the number of duplicate files crawled in our search engine. Based on the overall distribution as shown in **Figure 5** and our crawl process as described previously, we find that there may potentially be a potential for content duplicates found within the sites that we have crawled. This is due to some of the seed pages sharing the same URLs as each other and potentially overlapping the pages that they crawl.

To mitigate against this problem, we use a very simple and naive strategy to eliminate the majority of the content through a simple hash based deduplication process. We build a hashmap which maps the exact URL of the page as the key and the list of UUIDs as the corresponding to the document and simply remove any of the excess UUIDs from any URLs with more than one of the same pages. Although this is a very naive strategy, we have chose to implement our algorithm in this fashion as it is effectively very fast and efficient. However, in the future it would be best to implement a more content based deduplication process to evaluate the individual content along with the URLs in context as an alternative and hybrid approach.

As a result of performing the overall crawl process (crawling 397,848 pages), filetype filteration, content based filtering, and page deduplication (reduced to 307236 pages), we have a total of 109,650 pages. Performing this process will not only save disk space for further persistence of data, but also help to reduce the overall runtime of the indexing process for our search engine significantly.

7 DOCUMENT INDEXING

In this section, we describe our document vectorization and indexing process. We first describe the corresponding methods we utilized to preprocess our raw text data, then introduce the algorithm we used to extract relevant text-based features from our data,

7.1 Data Preprocessing

Prior to indexing our document, we perform a text preprocessing routine for each document. This pipeline is utilized during both the construction of the index as well as during the incoming querying process to normalize the inputs prior to generating the query index vector. In our implementation we have also made the effort to parallelize this process as well, making much use of the processing cores that were available to help improve the processing speed for the overall preprocessing step.

For much of our text preprocessing routine, we utilized the Natural Language Tool Kit (NLTK) library [15] under Python to perform much of the relevant NLP based task. NLTK offers many of the key functionalities for tokenizing and stopword removal. We also further lower cased all of our text to normalize between each of the different text entities found in our documents by calling the native lower function offered by Python string libraries.

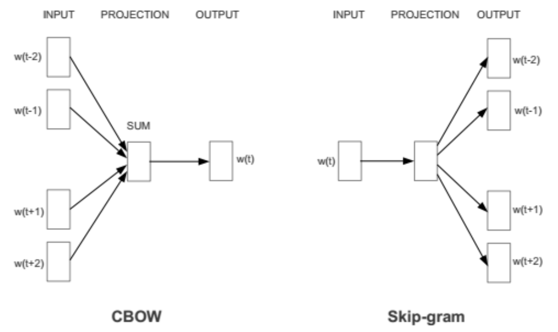
In our implementation we did not make use of the stemming as we empirically saw and noticed that much of the semantic distinction of the terms become rather coarse and reduces the overall empirical precision of the search engine. Therefore, we have not implemented the stemming function as part of our preprocessing routine.

7.2 Document Embeddings

To convert our documents to corresponding vector representations, we utilized a Deep Learning based method proposed by Le et. al [14] to use document embeddings - which are based on the original methods proposed by Mikolov et. al [16] for generating word embeddings for a distributed vector representation of words. We utilized Document Embeddings over regular word embeddings as they are capable of modeling texts of variable lengths and how these representations are much more appropriate for information retrieval type tasks.

While there are other methods for feature extraction of text based information such as Bag-of-Words, TF-IDF, and Latent Dirichlet Allocation (LDA), we find that Document Embeddings are much more powerful for their capacity to model word semantics effectively and how they are currently being used in many of the state-of-the-art natural language processing and machine learning task with great success.

Figure 6: Document Embedding Model Architecture



Document Embeddings are generated through training a two-layer neural network model (**Figure 6**) which aims to predict the corresponding words given a document ID (our UUID we have generated from the metadata) by maximizing the following average log probability:

$$\frac{1}{T}\sum_{t=k}^{T-k}\log p(w_t|w_{t-k},...,w_{t+k})$$

where T is the total number of tokens in a given data set, t being a positional index variable, k being the radius or the half the window size of the contextual frame being evaluated, and w being the individual token such that w_t would be the t-th word found in the document. The corresponding maximization is based on the modeling of a softmax classifier (multiclass classification algorithm) with the following probability function:

$$p(w_t|w_{t-k},...,w_{t+k}) = \frac{e^{y_{w_t}}}{\sum_i e^{y_i}}$$

where we optimize the parameters U and b through the average concatenation of the tokens w through function h given the set of words sequences W modeled as a linear combination between the parameter U, b and h defined as y.

In our implementation we, utilized the skip-gram model, where we set the document as our prior given the text found in the document as the conditional parameters in our model. Empirical results show that the skip-gram model outperforms the continuous bag-of-words methods and is often used in various embedding methodologies.

Correspondingly we train our model for 500 epochs over the entire dataset of crawled document, where we used 1024 dimensions for our embedding dimension, a window size of 8, a minimum word count of 5 in a document, and with a constant learning rate of 0.001 as our training hyperparameters.

To facilitate with the building, indexing, and querying of our document embeddings, we utilized a framework known as Gensim [17]. Gensim is an open-sourced library for various topic modeling and text feature extraction processing. It has been used across various industries and is a well maintained library with many other types of indexing models we can make use of. This library allows us to quickly prototype and iterate through the process of finding good parameters for our indexing engine, making it very versatile and powerful to use.

To query new documents in our pipeline, we have developed a custom function where we perform the same preprocessing pipeline as describe previously, then use Gensim's "infer_vector" function, which aims to generate an approximate vector representation of the new document in context with the vectors found in our corpus through feeding the input in the linear combination vector we have modeled for our document embeddings. Given this vector, we can use the internal function provided by Gensim or another nearest-neighbor based algorithm such as KDTrees or Locality Sensitive Hashing to find the top-k results, where k is the number of results we want to return, which is ranked based on the cosine similarity metric to measure document similarity of two document embeddings. This function has been implemented as the primary query function used when a user enters a query into the search engine. In the following section, we will discuss the integration with the front-end interface and how our querying engine will render the content to the client end-user.

8 FRONT-END SYSTEM

In this section, we describe our front-end system which is based on a web-based interface that can be accessed through a web-browser. We subsequently describe our interface design, followed by a discussion of our REST-based API Query Engine, and finally some discussions on scaling and deployment of the system as a microservice oriented architecture for future work.

8.1 Interface

The front-end component of our website was developed in HTML, CSS, and JavaScript using various libraries such as JQuery and Materialize CSS [9] based on the Google Materialize UI for a clean and user friendly experience. We ensured that the front-end was plain and simple, so that our end-users are able to use the search engine with no issue or confusion. The rendering of the site is generated from a template based approach using a Python based library known as Flask [12] to handle content rendering on the front end using the Jinja [8] templating schema.

For coding our frond end, we simply built two static pages: the main query page, which has a very similar layout to the Google homepage, and the results page, which renders the list of the pages it has found. Due to time constraints and development effort, we did not focus too much on the front end. Hence, features such as pagination were not considered in the design of the page - only rendering the top 25 pages found from our querying engine.

8.2 Query Engine API

As previously mentioned, we implemented our entire backend for the front-end interface in Flask, which utilizes an MVC based structure to handle process flows between the business side logic and the rendering of the front-end client. Our views comprises of the two pages as described in the previous section.

When a user enters a query - either at the main query page or at the results page, the handler redirects the browser to the results page with the content accepted as a GET parameter from the request. We subsequently take this GET value from the URL and feed the text entry directly through the querying function we have implemented - which performs both the tokenization, vectorization, and nearest-neighbor search and returns a dictionary based data structure containing all of the relevant content which is shown to the user.

The querying function, internally from the nearest neighbor search returns a list of UUIDs ranked by their highest cosine similarity. We then take the corresponding UUIDs and map them against the JSON metadata lookup dictionary file we have generated earlier on through our content extraction pipeline and prepare a list of dictionary data structure back to the view logic that made the call to the query function. Here we provided only the URL and the corresponding web page title per document.

8.3 Microservice Scaling

To further scale the deployment of this search engine, we can easily convert the corresponding Flask architecture we currently have into a stripped down microservice architecture. Microservices allow for both horizontal and vertical scaling of the services across different platforms and easily allows for better integration through a REST based API service making it platform agnostic to the type of end users who wants to utilize our services.

To transform what we currently have and refactor it for a microservice architecture, we can simply remove the jinja template rendering and directly return the dictionary based results from the query engine API directly as the output for the corresponding REST API call response. When designing the query function we have had considerations of implementing this separately as a REST API more so than a monolithic MVC based application - however, we made sure the design of our platform was flexible enough based on the requirements of the assignment. To deploy the service within an enterprise architecture we can serve this on a Docker based container to ensure that we have no issues with library dependency and version control issues. This way, we can share the container and have this be deployed on any platform very easily.

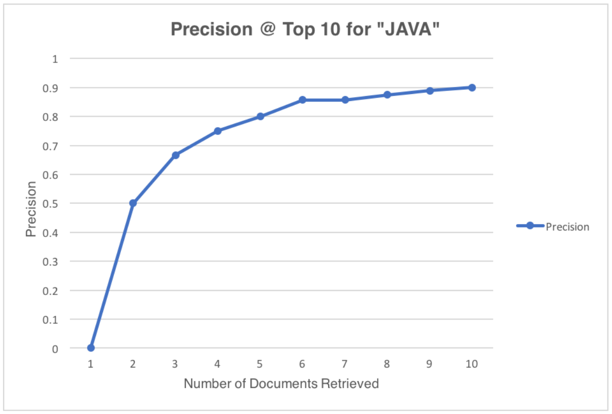
9 EVALUATION

In this section we discuss our evaluation methods for our custom search engine and the corresponding results for our empirical evaluation for the search engine's performance. In the previous section have presented both our Google Custom Search Engine and the Custom OER Search Engine. However, as noted in **Section 2**, our Google Custom Search Engine had issues with the results, primarily due to how it only returned a the results page of the OER Commons Browse page with the query as the "content" of the page - which functionally as a search engine is redundant and irrelevant to the eyes of a typical user. Therefore, we concluded that it would have been impossible to evaluate the Google Search Engine on a quantifiable basis, and decided that we would only evaluate the precision of the OER Commons Search Engine we have constructed. We only evaluated the precision as it is impossible to know how many pages total are out there based on the pages crawled.

In this evaluation, we queried the search term "Java" as we have found that in the OER commons site there are many content uploaded to the site pertaining to Java. The resulting query we had returned approximately 24163 sites, which we found to be too large - however for this evaluation we only looked at the top 10 results returned from the indexer. We also empirically found that the search engine queried and returned all of the results for this query in approximately 0.33208 seconds, demonstrating the speed at which our indexer is able to perform the query at.

We then empirically evaluated the performance of our search engine based on whether the returned page has anything to do with the "Java" Programming Language - which we sought as relevant through either tutorials, videos, lectures, or anything related to the specifics of the programming aspect of Java as **relevant**. As a result we found that our search engine performed significantly well, such that almost all of the returned result, except for the first page, was relevant - as shown in the precision plot in **Figure 1**. We see that even though we increase the number of pages retrieved in our collection, the returned results were all still relevant, thus proving our system to be very effective in indexing content.

Figure 7: Precision of "Java" @ Top 10



10 DISCUSSION

In this section, we provide a retrospective analysis of our overall implementation, some key observations on the search engine's behavior and any future work that we can consider to improve the overall quality of the search engine.

For our specialized search engine, we have implemented almost the entire architecture from scratch - which we have found to be a very good learning experience, however from a practical standpoint being very complicated and ambitious. If this system is going to be deployed as a usable service, there are several areas where we must improve and work upon to ensure that the system is robust and ready for production with minimal bugs and errors. The source code for our implementation can be found under https://github.com/yutarochan/IST441.

As for the behavior of the overall search engine, we find that the search engine quality works moderately well for what it was designed for with regards to how we defined "relevancy". For search terms, we found that it performed empirically well on general topic queries like "math", "science", "astronomy", and "economics". However, it is important to note that the content found within this search engine are only limited to the content based on the seed URLs we have scraped. In other words, if for example the OER commons site did not have any content related to "information retrieval" as a submitted resource, our search engine will not have found that particular type of resource within our indexed website because it was not there to begin with.

Another key issue we noticed is the types of pages that were returned to us often titled "Page Has Moved" or "Page Not Found" - which makes it very misleading for being able to find the right page in the results. We can either mitigate this issue using a topic modeling approach to identify key topics in the document using a machine learning based approach to recommend titles for a website that may not have corresponding or appropriate titles. However, for this we believe that the primary cause of this is due to the rendering as seen by HTTracker and shows the limitation of the type of tracker.

Finally, better error handling mechanisms should be enabled as we did notice a couple of queries leading to an error page generated by Flask - which indicates that something went wrong internally within our query engine. In short, to productionize the search engine we must ensure that all components are scalable and robust before being able to deploy it under such heavy use.

11 CUSTOMER EVALUATION

Our customer, Professor Fusco from the College of IST has a heavy background in Enterprise System Integration. It was raised to our attention that he expressed interest in acquiring an OER search engine. He believed that "this would be very powerful to implement within a classroom setting." It was important for us to deliver a well rounded custom search engine for him and other users such as students and faculty to retrieve educational resources efficiently. We presented the custom specialized OER commons search engine along with Google custom search engine to Dr. Fusco Friday morning, April 27th. Before diving right into the demo we briefly described a high level explanation of what we accomplished this semester touching back on the slides we presented during our last class.

After going over the system architecture Dr. Fusco requested us to create a more in depth system architecture model on Visio demonstrating the relationship between servers, microservices, and languages being utilized. The visual representation was designed with the intention of making it quick and painless to share the framework of the system with colleagues. Professor Fusco was satisfied with the simple, clean look of our search and results page. That being said, after examining the front end we proceeded to show him our Google Custom Search to showcase the type of results that can be expected from these custom search engines. The first query he had us type in was "json". Immediately, we received several links that led to informational sites covering JSON concepts that were quite relevant to the query searched. He was impressed by the engines ability to eliminate some of the unwanted clutter that a typical Google Search would display. Additionally, Professor Fusco was impressed by the variety of websites that were in the results and the amount of pages we crawled. Finally, most of the results we viewed were accurate to the professors query and there were few irrelevant results returned, adding to the overall experience. Our team is delighted that Dr. Fusco will use our specialized OER commons search engine and hopefully implement the engine to assist other users with validated data retrieval in the future.

Professor Fusco wishes to keep the specialized OER commons search engine live and we plan to provide him with our services aiding with the implementation and maintenance of the system in any such fashion. As we look to further deploy this system in a much larger scale, we will look into improving and finding areas where we can use alternative software to mitigate large asynchronous data processing platforms such as Apache Spark. With our strong background enterprise systems, microservices architecture, big data processing, and machine learning we are confident that we can communicate well with our customer and make a huge difference for Open Educational Resources Community.

12 CONCLUSION

In this paper, we have introduced our specialized search engine for the OER Commons website. We presented our entire IR system architecture and described each of the corresponding components and their key design choices for each of the components. Finally we provided some insights, comments, and evaluation for our system as well as some of the future direction of how we plan to evolve the system to help support the mission of the Open Educational Resources community.

REFERENCES

[1] [n. d.]. Apache Nutch. ([n. d.]). http://nutch.apache.org/
[2] [n. d.]. BeautifulSoup. ([n. d.]). https://www.crummy.com/software/BeautifulSoup/
[3] [n. d.]. DCMI: Home. ([n. d.]). http://dublincore.org/
[4] [n. d.]. Explore. Create. Collaborate. ([n. d.]). https://www.oercommons.org/
[5] [n. d.]. Github - fscrawler. ([n. d.]). https://github.com/dadoonet/fscrawler
[6] [n. d.]. Heritrix - IA Webteam Confluence. ([n. d.]). https://webarchive.jira.com/wiki/spaces/Heritrix/overview
[7] [n. d.]. HTTrack Website Copier. ([n. d.]). https://www.httrack.com/
[8] [n. d.]. Jinja. ([n. d.]). http://jinja.pocoo.org/
[9] [n. d.]. Materialize. ([n. d.]). http://materializecss.com/
[10] [n. d.]. Requests: HTTP for Humans. ([n. d.]). http://docs.python-requests.org/en/master/
[11] [n. d.]. Textract. ([n. d.]). https://textract.readthedocs.io/en/stable/
[12] [n. d.]. Welcome | Flask (A Python Microframework). ([n. d.]). http://flask.pocoo.org/
[13] Christian Kohlschütter, Peter Fankhauser, and Wolfgang Nejdl. 2010. Boilerplate detection using shallow text features. In *Proceedings of the third ACM international conference on Web search and data mining*. ACM, 441–450.
[14] Quoc V. Le and Tomas Mikolov. 2014. Distributed Representations of Sentences and Documents. *CoRR* abs/1405.4053 (2014). arXiv:1405.4053 http://arxiv.org/abs/1405.4053
[15] Edward Loper and Steven Bird. 2002. NLTK: The Natural Language Toolkit. In *Proceedings of the ACL-02 Workshop on Effective Tools and Methodologies for Teaching Natural Language Processing and Computational Linguistics - Volume 1 (ETMTNLP '02)*. Association for Computational Linguistics, Stroudsburg, PA, USA, 63–70. https://doi.org/10.3115/1118108.1118117
[16] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. 2013. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*. 3111–3119.
[17] Radim Řehůřek and Petr Sojka. 2010. Software Framework for Topic Modelling with Large Corpora. In *Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks*. ELRA, Valletta, Malta, 45–50. http://is.muni.cz/publication/884893/en.