EECS 767

Information Retrieval System

Group 2: Daniel Murga, Casey Sader, James Muoghalu

16 May 2018

Search Engine Location: https://people.eecs.ku.edu/~j286m692/eecs767_webfiles/

Git Repository: https://github.com/jmuoghalu/EECS767_CourseProject

Abstract

The goal of the project was to design and implement a complete information retrieval system with the following components: document processing and indexing, vector space model, niche crawler, term proximity, and relevance feedback. We implemented these components using the Python language. Before the system is used, a web crawler has scraped a web page and its links, indexing and dumping the followed URLs. These downloaded pages are processed and index and a vector space model is created. A user query is then compared to this vector space model and the most relevant links are displayed on the web page to the user. Two other extensions have also been added. The first involves term proximity which takes into account not only the words in a document, but also the proximity of the query terms to each other. The other extension involves relevance feedback which allows users to decide whether certain links provided are actually relevant or are not relevant. A new list of links is displayed to the user based on the new information.

Approach and Outline

For this system, it was designed and implemented in the Python language. Python provides many features to programmers that make implementation easier, especially for widely used algorithms in information retrieval systems. Libraries for string manipulation, HTML parsing, OS interaction, and various others. Each section of the information retrieval system is separated into unique classes, so each part is encapsulated and can be used separately from the main system. First, the search engine directory is populated with documents that were crawled from a specific domain using our implemented web crawler. The crawler returns a set amount of pages crawled from a domain and stores them for future processing. These pages are indexed and processed before the system is used, as to prevent this processing time from slowing down user searches.

Once the files have been processed into text documents, these text files are read in order to create an inverted index file that will be saved for running the search engine. The vector space model is later created from the crawled documents and the dictionary of words for the system. This vector space model is saved to memory at runtime so the calculations required are only performed once after the documents have been processed. This directory and model is the basis for the search engine. Through the web portal, a user can input a search query to the system. This query is processed like the documents, compared to the vector space model, and the most relevant results are displayed to the user. Once the most relevant links are displayed, there are options to give feedback to improve the user's query and return results more closely relevant to the feedback given. The list is refreshed with the newly modified query and new results are displayed.

Code for this project was written in the object-oriented paradigm. The project repository contains five main class files and eight drivers. The class files correspond to the functionalities of the web spider, the document processor, the inverted index, the vector space model, and the query similarity calculator. Four of the drivers are used to run the search engine programs via the terminal, and the other four are used to send search data to the webpage that we have set up as our search engine's user interface. Among these drivers are one for activating the web crawler; three for offline basic searching, term proximity searching, and relevance feedback searching; and four drivers for online basic searching, term proximity searching, and relevance feedback searching.

Web Crawler

Web crawlers are algorithms designed to read and download webpages, collect all the links from the pages, and visit and do the same on those pages. All of the links are stored in a queue which threads, called "spiders", pop from and then continue to add pages to the frontier unless a link has already been crawled. Crawlers when at their best are designed to be multi-threaded on multiple machines while constantly crawling over many domains in an attempt to crawl as much of the web as possible while keeping all pages up-to-date. The web is basically endless and ever changing so this is not fully possible, but the best search engines come as close as they can. Another important component of a web crawler is deciding which types of pages and links to actually follow. Many sites have dynamically created pages and typically these are avoided as they are difficult to handle.

When it comes to being polite to web servers, there are two main actions to take. The first is to limit how often you hit a web server. This can most effectively be done by having all of the threaded spiders be on seperate domains, and then limit how often they can choose a link from the queue that is from each domain. Having multiple seed sites that will have links in different areas can greatly help in diversifying the URL frontier of sites that the spiders can select from. The other way to be polite is to follow the rules set out in the robots.txt file that many sites include directly next to the domain URL (e.g. http://www.espn.com/robots.txt). These are pages that are created to inform a crawler what pages the site wishes to not have crawled. This can have sections that are directed at all crawlers, or sections that are directed at specific crawlers, generally based on the way they have crawled the site in the past. These are not pages that are required to be forced though and that's why it's called being polite. It is possible that not following them can lead to problems as sites could set-up infinite links that lead to nowhere if a crawler doesn't listen to the robots.txt rules.

The general topology of a web crawler can be seen in Figure 1, found in the web slides from class. There are 3 domains: the crawled documents, the frontier of documents yet to be crawled, and the web which the crawler cannot reach and won't crawl. The domain of the crawled documents contains all downloaded and crawled documents which were parsed, which includes the seed pages. The URL frontier contains all pages that have yet to be parsed but will be and is where the queue of documents to be crawled is. Separating these two domains are the spiders. The last domain is all other pages on the web which will not be touched by these spiders. Figure 2, also taken from the class web slides, shows the general architecture of a web crawler. A URL is read and sent to a DNS server which then allows the site to be accessed and fetched. This page is then parsed and added to the list of downloaded and crawled pages. The links parsed from the page are then filtered by the cached robots.txt information and if allowed, this link is added to the set of URLs to be crawled. This process is then repeated continuously until all links in the frontier are crawled or the desired number of pages to crawl has been reached.

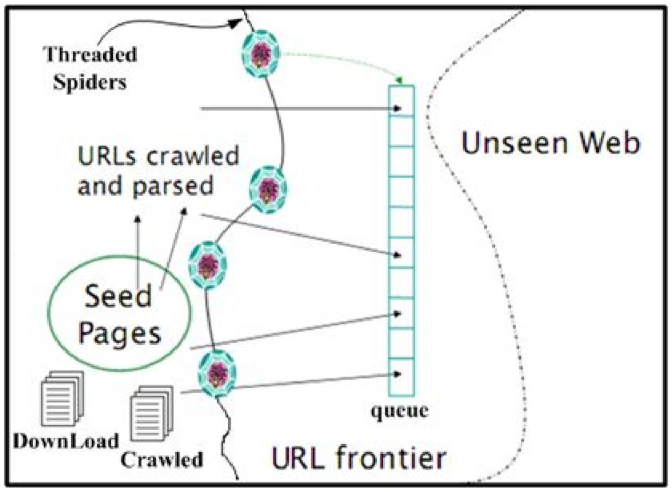


Figure 1

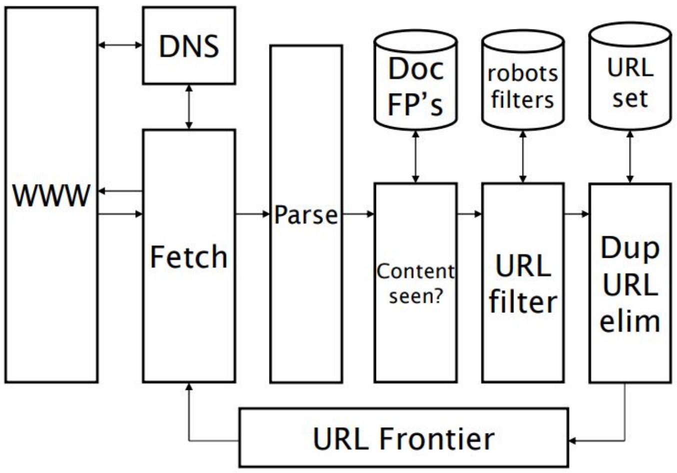


Figure 2

The final version of our search engine uses (https://en.wikipedia.org/wiki/United_States) as the initial page for web crawling. 500 Wikipedia files were downloaded in a process that took around 30 minutes.

Document Processing and Indexing

The inverted index is what is used to hold all terms, term frequencies, term document frequencies, and frequency information for the dictionary containing all words in the processed documents. Once documents have been added to the system directory, either from the crawler or test files, they are fed to the python document processor. This uses available python libraries for HTML parsing, the stemmer, and stopwords to process the text within the HTML files. The document's HTML tags are stripped, and the text is normalized by the stop list and stemmer, two features that are provided by Python's Natural Language Toolkit, or NLTK. Various other pieces of information about the specific HTML pages are saved so that they can be retrieved easily at a later time within the system.

For the inverted index and posting list, a combination of python lists and dictionaries are used. Python dictionaries utilize key-value pair structures. Using each term as a dictionary key makes it easy to get all of a term's information. Dictionary values are created using a custom "IndexEntry" class. Objects created from this class act as useful containers for the term's posting list and other data that will be used to build the vector model. All the document's terms are added to the index if they are not already present. Posting lists, term TF's, and term DF's are then calculated and updatedat the appropriate times.   Documents which contain a specific term have their IDs added to that term's posting list, which updates the term frequency variable.

Once all documents have been read, the program will then write the index to a text file for later user queries. Before this occurs, however, the term IDF for each unique term is calculated and saved in that term's IndexEntry. These IDF values will be used to calculate the document lengths, which will save search engine time when building the vector model and calculating query similarities. The inverted index is written to a special file that holds both the index terms document list so that it can be read quickly when determining the vector space model and does not require recomputation. The index is then sorted by term to be easily retrieved.

Regarding the final cache of crawled documents used for our search engine, the program takes around 45 minutes to process all 500 HTML files and apply the stop list and stemmer. This will create a folder of shorter text files from which the program takes around 20 seconds to create and save the inverted index.

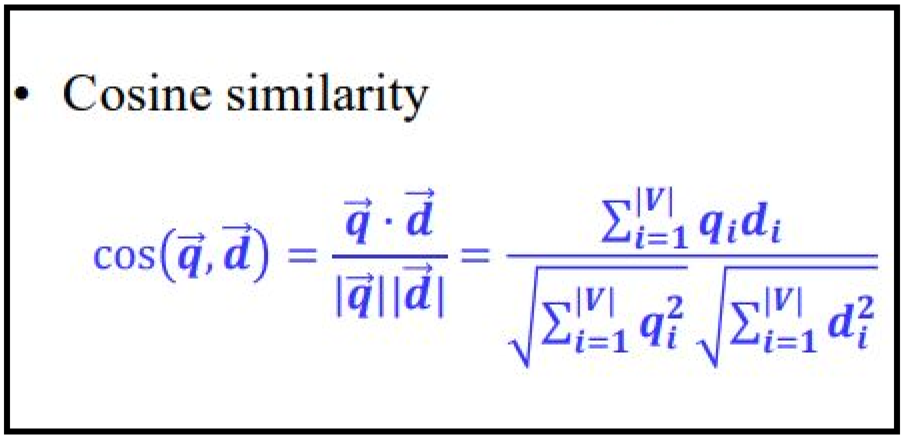
Vector Space Model and Query Similarity

A vector space model is a matrix that is used to represent documents and the terms within them. Values within the matrix are known as TF-IDF weights which represent products of the term's frequency within the document (i.e. the relevance between the term and the document) and the term's inverse document frequency (i.e. the rarity of the term appearing in the document cache and therefore the term's importance within the documents in which it does appear).

After processing all the documents and creating the inverted index, the vector space model for each document is created. Using the recently created index, the model first prepares each document vector to be calculated. It reads the IDF values from the index and calculates the tf-idf weights for each term, and updating it with the actual value if the term exists within the selected document.

The vector model is saved into a Python dictionary with the keys being the terms and the values being lists of tf-idf weights. Each index in the list corresponds to the tf-idf for a document. For example the weight value at index zero for a particular term corresponds to the first document in the document cache. Obviously, for sizable document sets with a large number of unique terms, filling the entire vector model each time the engine is run is inefficient. For similar reasons, saving the entire model into a text file is also not the most efficient way to gather term data.

Queries are passed in to the search engine as a list of strings. These strings are run through the stop list and stemmer so that they can be used to properly match query terms to terms inside the inverted index. The cosine similarity technique is used to match queries to documents. For every word in the query, the tf-idf values are calculated within the vector model. Document lengths and term IDF values are saved within the index and can be quickly loaded when the search engine. Once the query vector and its length have been calculated, a simple arithmetic equation is used to calculate the similarities between the current query and every document in the file cache. If the similarity value for a particular document is zero, then the document is ignored leaving only non-zero similarities that need to be sorted. Lastly, the ten most similar documents are returned to the driver programs so that users can see their search results.



- Cosine similarity

$$\cos(\vec{q},\vec{d}) = \frac{\vec{q}\cdot\vec{d}}{|\vec{q}||\vec{d}|} = \frac{\sum_{i=1}^{|V|} q_i d_i}{\sqrt{\sum_{i=1}^{|V|} q_i^2}\sqrt{\sum_{i=1}^{|V|} d_i^2}}$$

(cosine similarity equation used by the engine, where q is the query, and d is the current document)

Regarding the boolean model and the addition of keywords such as “AND” into the query, if the user were to search for “colorado AND mountain,” they would be seeking documents that contained both words. Since “and” is a stopword, it would be removed from the query before the document similarities are calculated. Therefore, the input for the search engine would be a list containing the word “colorado” and the word “mountain.” Intuitively, documents that contain both words would have a higher similarity value than documents that contain one or neither. Assuming there are documents in the file cache that contain both words, the user can safely assume that their information need has been satisfied by one of the documents that is displayed to them.

Term Proximity

An outline of our custom term proximity algorithm follows. When a user provides a query into the search engine, the document similarities will be calculated as they would for the basic searches. However, in this case, the engine would return up to 20 documents instead of the usual ten. This is so that the proximity ranking algorithm will have a greater body of files from which to determine new rankings. The term proximity algorithm is written as a method within our Query class, so the processed query vector, and the pertinent document vectors will be readily available. An array of size less than or equal to 20 will then be created for the scoring of documents. All values in this array will be initialized as zero.

The algorithm will then loop through each of the up to twenty most similar documents. For each document, a list every word inside of it will be created, and a helper variable will be set to track the location within each document. An additional dictionary will be used to save term locations. The keys are the terms in the query vector, and the values are initialized as empty lists. It should be noted that the files being read here are not the original HTML documents, but rather the text files that are created by the document processor.

For each term in the lists of words, if that term appears in the query vector, then the location is saved within the location dictionary. Note that this dictionary only tracks isolated occurrences of each search term. After the document has been read in full, this dictionary will be filtered to ignore query terms that do not appear in the document. This will ultimately affect the final score that this algorithm gives the document for reasons that will be addressed later. At this point, it is time to consider each term's proximity relative to the other terms. A new list of tuples will be created from the dictionary of term occurrences. The first value in this tuple will be the term itself, and the second value will be the location within the document. Observe that if one of the document is 100 words long, then these location values will lie between zero and 99, using the convention of zero-based addressing.

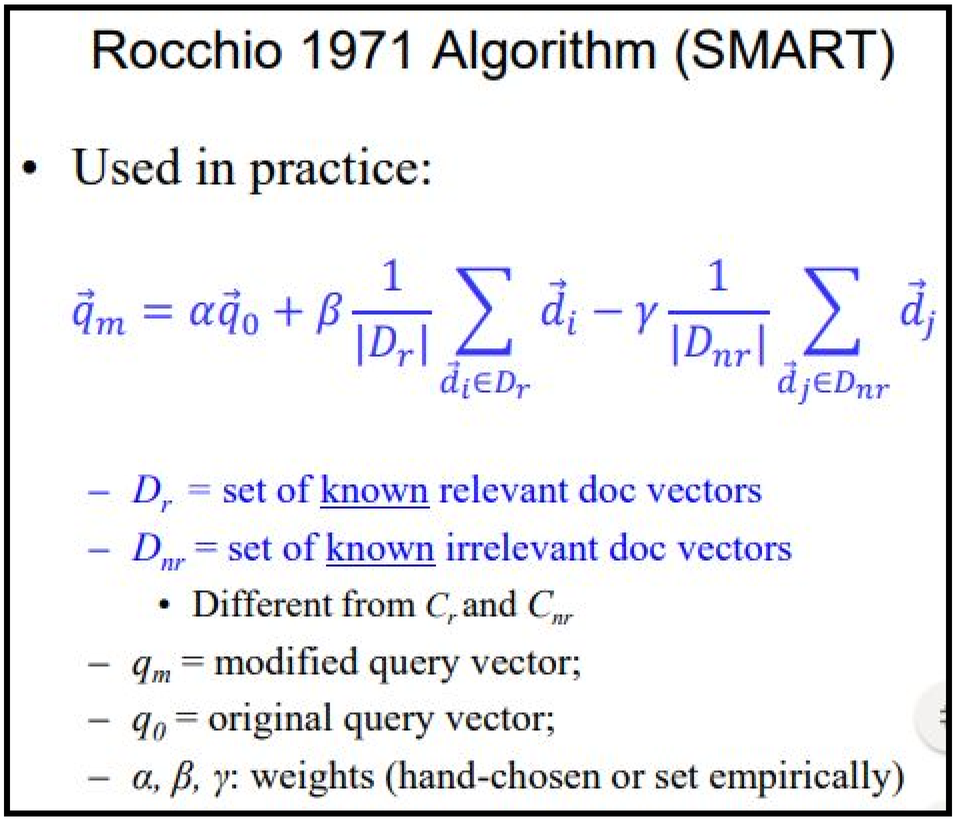
Now, the algorithm has populated a list of tuples containing terms and their locations in the document. It should be noted that this list is ordered based on the query vector. If a term appears first in the query, then all of the tuples related to its locations in the document will be placed towards the front of the list. It is likely that there are multiple tuples in the list correspond to the same query term. Our algorithm is only interested in term locations relative to other terms in the query. If a tuple in the list is only adjacent to another tuple referring to the same term, then that original tuple will be dropped. The algorithm will iterate through the list and drop tuples that match this condition.

The process of filtering the list of tuples will leave the algorithm with a relatively shorter list that will be used in the actual scoring of term proximity. For every pair of adjacent tuples that refer to different terms, the location distance between each term will be calculated. The minimum distances between a query term and another term in the query will be tracked during these calculations, and the inverse of this minimized value will serve as the document score. The inverse is taken to ensure that documents with smaller distances between query terms have a higher proximity ranking.

Once the initial loop of documents is complete, all document proximity scores will have been saved, and they will be used to update the document similarity values that were initially calculated. The new similarities will simply be a product of the old values and the document scores. As with the basic searches, the final list of similarities will be cleared of zero values and sorted in reverse order so that users can be given the best search results.

Relevance Feedback
Relevance Feedback is a mechanism by which users can provide feedback to the search engine on the "relevance" of the initial set of results that the engine provides after queries. Our engine's relevance feedback functionality was provided by applying the Rocchio Algorithm to our vector model in order to update the query vector using the values within the relevant documents' vectors and the irrelevant documents' vectors.

In order to weigh the relevant and irrelevant documents, our engine sets the values alpha, beta, and gamma as one, 0.75, and 0.25, respectively. Also, since the Rocchio Algorithm makes use of the full document vectors, searches that want to use the feedback option will create the entire vector model, an expensive operation that ultimately increases the time it takes to return the initial set of results.



Rocchio 1971 Algorithm (SMART)

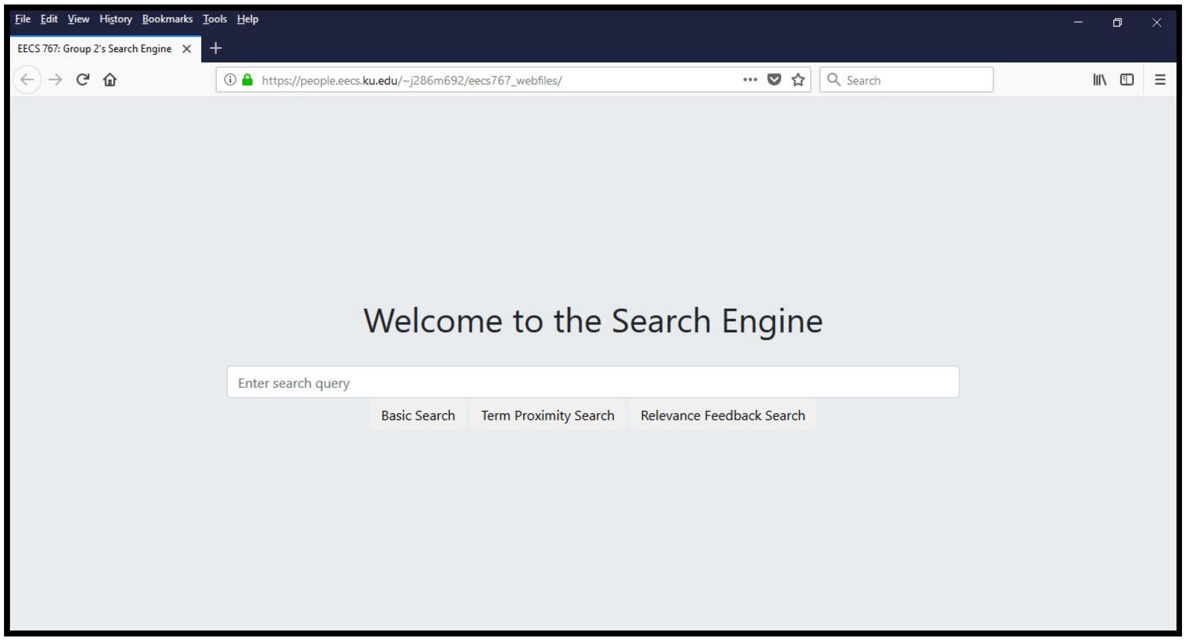
- Used in practice:

$$\vec{q}_m = \alpha\vec{q}_0 + \beta\frac{1}{|D_r|}\sum_{\vec{d}_i \in D_r}\vec{d}_i - \gamma\frac{1}{|D_{nr}|}\sum_{\vec{d}_j \in D_{nr}}\vec{d}_j$$

- D_r = set of known relevant doc vectors
- D_{nr} = set of known irrelevant doc vectors
  - Different from C_r and C_{nr}
- q_m = modified query vector;
- q_0 = original query vector;
- α, β, γ: weights (hand-chosen or set empirically)

(Rocchio Algorithm as used in our search engine)

Web Interface


File Edit View History Bookmarks Tools Help
EECS 767: Group 2's Search Engine
https://people.eecs.ku.edu/~j286m692/eecs767_webfiles/
Search
Welcome to the Search Engine
Enter search query
Basic Search
Term Proximity Search
Relevance Feedback Search

(*search engine homepage*)

Our search engine's web interface is hosted on the EECS servers via one of our group member's people.eecs.ku.edu page. Queries are provided through the search bar. The three buttons correspond to Steps Four, Five, and Six of the project, respectively. For Steps 4 and 5, the top results, document snapshots, and URLs will be presented in order from most similar result to least. The results page for Step 6 provides a list of checkboxes at the bottom of the page from which users can select the documents that are relevant. There is a numbered box for each result, and in order to mark a document as irrelevant, users should leave a box unchecked.

The web drivers used to produce results for the online search are called by the webpage's PHP code using simply terminal commands. The Python programs will then print search results via JSON encoding, which will be detected by the PHP and then displayed.

A link to the engine homepage is provided on page one of this report.

Results and Analysis

The Step Four version of this search engine determines search results by simply computing query-document similarity and sorting the values. The time it takes to sort the documents is made larger by the size of the file cache. As was mentioned earlier, in order to save the search engine time in creating the vector model, the entire document vectors are not created for this search. A set of tf-idf weights is only computed for search terms that appear in the query and in the index. Users can expect searches here to take five to ten seconds.

Step Five adds an additional step to the searching process. After up to twenty of the most similar documents are collected, this set is reduced using the term proximity algorithm. If users run the same query through the Step Four basic searches and the Step Five term proximity searches, they can typically find that the computation times are the same as well as the top two or three search results that display on the webpage. The remaining results, if they appear, are typically in different orders or contain different documents.

Searches utilizing Step Six's relevance feedback take a noticeably longer amount of time. Our implementation of Rocchio Algorithm utilizes the entire vector model and populates every document vector before calculating similarities. Also, the fact that updating searches requires user feedback complicates how our search engine must be run. Since the PHP is calling the Python code via the terminal, two parallel searches must be made. The first will call the source code and pass in the user's query in order to collect the initial result set, and the second will call the source code, pass in the user's query, and pass in the information concerning relevancy. As a result, our search engine is performing redundant computations of both the vector model and the initial similarities before then applying Rocchio Algorithm. The initial searches typically take 10 to 15 seconds, and the display of updated results takes close to 20 seconds.

Project Log (Previous Reports)

Updated Project Report 1- (3/16/18)

Goal:

To design and implement a complete information retrieval system with the following components: document processing and indexing, vector space model, niche crawler, term proximity, and relevance feedback.

Approach:

The information retrieval system will be implemented using Python, for its ability to support functional and object-oriented code. Python has a large list of internal libraries that make performing complicated data changes easier.

For document processing, the directory with unprocessed files is fed to an HTML parser that strips the HTML tags and makes the text lowercase line by line. To implement the stemmer and stopper, Python provides easily available stemmers and stoppers libraries that can be imported, created by the Natural Language Toolkit (NLTK). Each processed file is output to a new corresponding processed file.

Once the file has been processed, each word is added to the dictionary array as a tuple containing the word and its document frequency, with a pointer to the posting list object that contains the document number, frequency, and a pointer to the next object, creating a linked list. After document preprocessing is done, the dictionary data is used to calculate the TF-IDF based ranking for each document. We are currently working on comparing the query and document vectors efficiently, and determining which data structures would be most beneficial to hold the vectors and inverted index.

Results:

At the time of this report's submission, only the document processing and indexing was complete. Testing of this document processor was done using the "docsnew" file cache that was provided by the course webpage. The indexer was tested by using a set of four text documents based off one of the class quizzes and printing the formatted index to the command line.

Updated Project Report 2- (Progress Made by 4/23/18)

Goal:

To continue the work done on our Python search engine to allow for users to run the program via the terminal and locate documents.

Approach:

For the vector model, our code was based on the vector space model quiz that we completed in class. Two dictionaries were used, one for storing a term's IDF values and one for storing the list of a term's IDF weights. Initially, we would calculate the document weights

For our web crawler, we realized that we would need to integrate an HTML parser in order to properly fill the URL frontier with pages to download.

Queries would rely on a previously-computed vector model in order to fill the vector of search terms and calculate similarities.

Results:

Our vector model code was complete, but it needed to be updated in the future since it defaulted to creating the entire vector model every time the search engine was run. Our group knew that this would slow down computations once we added the crawled documents to our project.

At this point, we had made some progress on our web crawler and spider, but the documents collected in this process had not been integrated into our search engine.

Query code was incomplete by the due date of this report, but the inverted index, the vector space model, and all the necessary document data was ready to be used.