University of Memphis

Memphis, Tennessee

Retrieval Engine for the University of Memphis

Information Retrieval (7130)

Tara Baniya

Due date: Dec 01,2015

Contents

1. Introduction

Information retrieval is the activity of obtaining relevant documents from the collection of documents. In this project, I have build a Retrieval Engine which has its own document collection. The crawler crawled about 12 thousand pages from the University of Memphis's website. Then the collected documents are preprocessed. In the preprocessing step: digits, punctuations, stop words, URLs and other html-like strings are removed. All the text are converted to lowercase and the morphological variation is removed by using Porter stemmer. Then an inverted index is created for the preprocessed documents. Preprocessing is done for the query as well. All the keywords of the query are converted to lower case and are stemmed. The Search Engine retrieves the document containing at least one of the keyword. The cosine similarity score between the query vector and the document vector is calculated to rank the retrieved documents.

Web Crawling:

Web crawling is the process of autonomously navigating through the URL to collect the content of web pages. The crawling is done by the software called crawler. A crawler starts from a seed URL, visit the pages and identifies the new URLs in the pages and visit the pages pointed by the new URLs. The process continues until the exhaustive navigation or satisfiable number of pages are visited. The following are the basic execution steps of a typical crawler.

- Initialize the queue of URLs (seed URLs)
- Repeat the following until no more URLs in the queue

  1. Get one URL from the queue

  2. If the page can be crawled, fetched the content of page

  3. Store representation of page

  4. Extract URLs from page and store them in queue

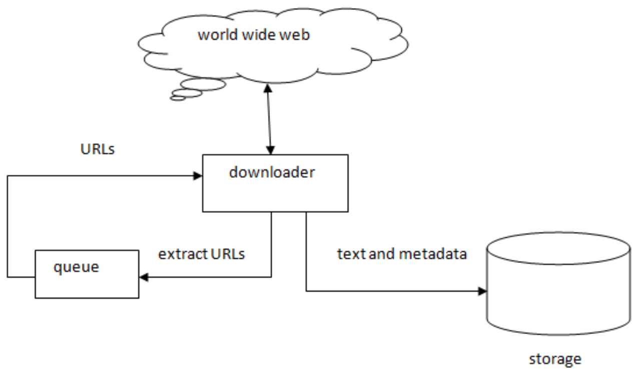


Fig. 1: web crawler architecture

Inverted Index:

The search engine incorporated an inverted index when evaluating a search query to efficiently locate the documents containing the words in a query and then rank these documents by relevance. The inverted index stores a list of documents containing each words , the search engine can use direct access to find the documents associated with each word in the query in order to retrieve the matching documents quickly.

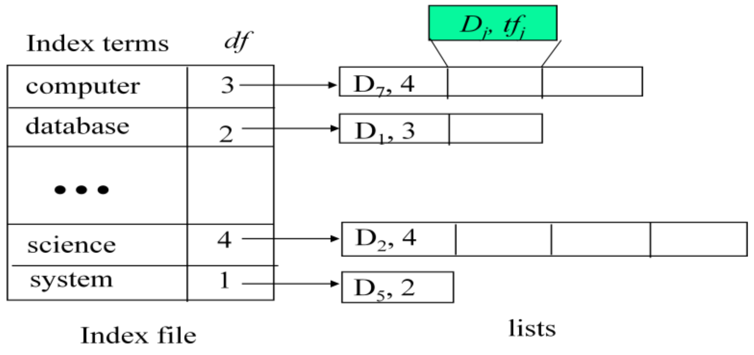


Fig. 2: web crawler architecture

2. Approach

Information Retrieval Model consists of document representation, query representation and a retrieval function. The main classes of retrieval models are : Boolean Models, Vector Space Models and Probabilistic Model. Boolean model is based on set theory and Boolean algebra. The documents are set of terms and queries are the Boolean expressions on terms. The probabilistic model works on the probability of relevance. Even though the Boolean Model is simple, it has certain limitation such as we can't rank the retrieved documents, it displayed all the relevant documents. For this project, I am using the Vector Space Model. The document and query are represented as vectors. The similarity between the query vector and the document vector is calculated to find the relevant documents from the large corpus of documents. This simple model provides partial matching and ranked result. It works well with large set of documents.

3. Design

The major components of a retrieval system are crawler, preprocessor, indexer, search interface and retrieval engine. The crawler crawls and downloads the web page, the indexer preprocesses and indexes the pages, search interface takes user query and displays the result and the retrieval engine receives query and retrieves and rank the pages. I will explain of them in detail below.

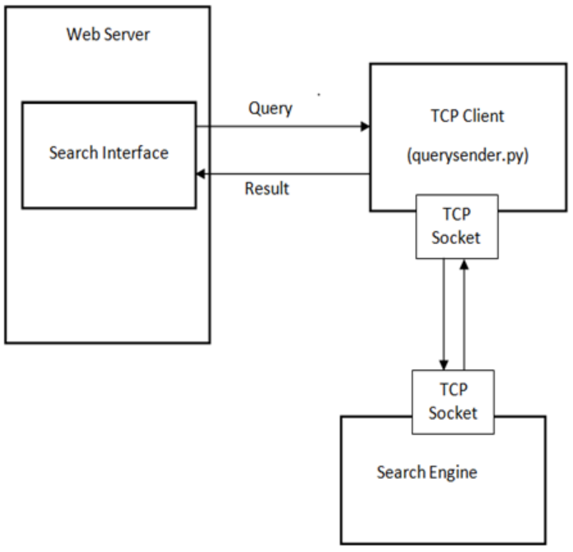


Fig 3: The architecture of the retrieval engine

Crawler:

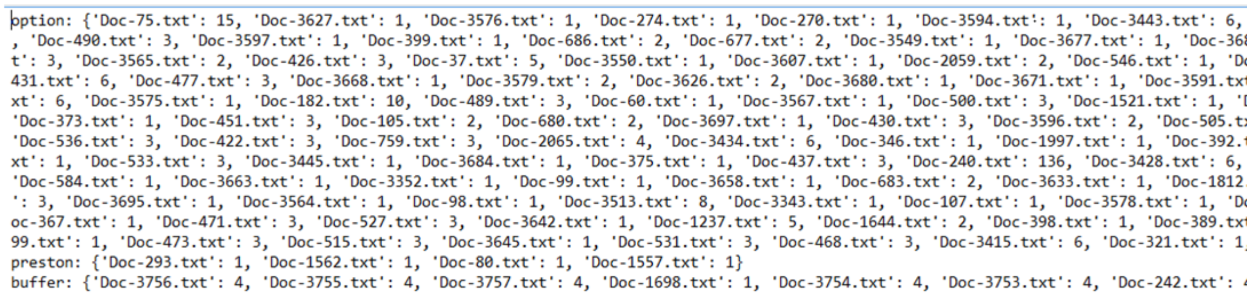
The crawler gets a seed URL visits the page, harvests more URL from the page and crawls through the new URLs in BFS order. To maintain the BFS order of visiting the URL, the newly harvested URLs are pushed in a queue and popped from the front of the queue to get next URL to visit. After visiting a page, the crawler downloads the page and save it in the local drive. In order to maintain the politeness, the crawler waits for few seconds after every visit. The duplicate crawling is avoided by maintaining a separate list of crawled URLs. The crawler looks up the crawled URL list before crawling any URL.

Preprocessor:

The raw documents collected by the crawler are needed to be preprocessed. The punctuations, digits, URLs and other HTML like strings are removed. The stop words are removed and all the text are converted to lower cases. Porter Stemmer is used to stem the words to remove the morphological variations. After preprocessing, the key words containing meaning are stored. The preprocessing helps to make the search efficient and gives the better result because it removes the stop words which occurs very frequently(about 80 %) but does not have their own meaning.

Indexer:

Indexing is one of the important component to build search engine. The purpose of indexing is to optimize speed and performance in finding relevant documents for a search query. Indexing constructs an inverted index of word to document pointers.



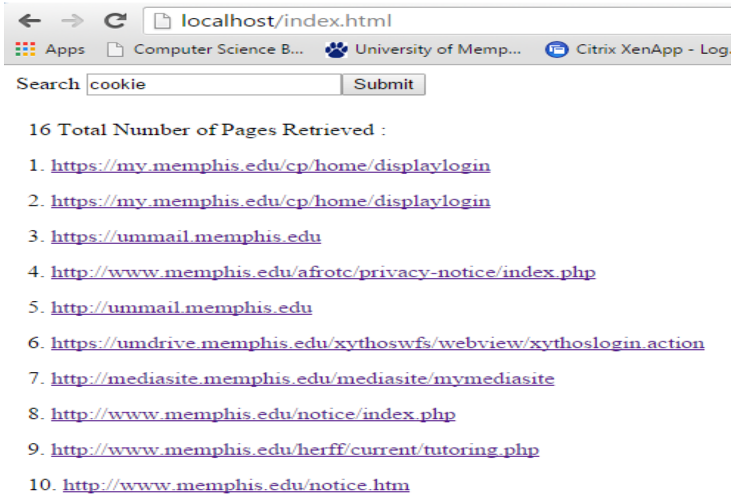
```
option: {'Doc-75.txt': 15, 'Doc-3627.txt': 1, 'Doc-3576.txt': 1, 'Doc-274.txt': 1, 'Doc-270.txt': 1, 'Doc-3594.txt': 1, 'Doc-3443.txt': 6,
, 'Doc-490.txt': 3, 'Doc-3597.txt': 1, 'Doc-399.txt': 1, 'Doc-686.txt': 2, 'Doc-677.txt': 2, 'Doc-3549.txt': 1, 'Doc-3677.txt': 1, 'Doc-36
t': 3, 'Doc-3565.txt': 2, 'Doc-426.txt': 3, 'Doc-37.txt': 5, 'Doc-3550.txt': 1, 'Doc-3607.txt': 1, 'Doc-2059.txt': 2, 'Doc-546.txt': 1, 'D
431.txt': 6, 'Doc-477.txt': 3, 'Doc-3668.txt': 1, 'Doc-3579.txt': 2, 'Doc-3626.txt': 2, 'Doc-3680.txt': 1, 'Doc-3671.txt': 1, 'Doc-3591.tx
xt': 6, 'Doc-3575.txt': 1, 'Doc-182.txt': 10, 'Doc-489.txt': 3, 'Doc-60.txt': 1, 'Doc-3567.txt': 1, 'Doc-500.txt': 3, 'Doc-1521.txt': 1, '
'Doc-373.txt': 1, 'Doc-451.txt': 3, 'Doc-105.txt': 2, 'Doc-680.txt': 2, 'Doc-3697.txt': 1, 'Doc-430.txt': 3, 'Doc-3596.txt': 2, 'Doc-505.t
'Doc-536.txt': 3, 'Doc-422.txt': 3, 'Doc-759.txt': 3, 'Doc-2065.txt': 4, 'Doc-3434.txt': 6, 'Doc-346.txt': 1, 'Doc-1997.txt': 1, 'Doc-392.
xt': 1, 'Doc-533.txt': 3, 'Doc-3445.txt': 1, 'Doc-3684.txt': 1, 'Doc-375.txt': 1, 'Doc-437.txt': 3, 'Doc-240.txt': 136, 'Doc-3428.txt': 6,
'Doc-584.txt': 1, 'Doc-3663.txt': 1, 'Doc-3352.txt': 1, 'Doc-99.txt': 1, 'Doc-3658.txt': 1, 'Doc-683.txt': 2, 'Doc-3633.txt': 1, 'Doc-1812
': 3, 'Doc-3695.txt': 1, 'Doc-3564.txt': 1, 'Doc-98.txt': 1, 'Doc-3513.txt': 8, 'Doc-3343.txt': 1, 'Doc-107.txt': 1, 'Doc-3578.txt': 1, 'D
oc-367.txt': 1, 'Doc-471.txt': 3, 'Doc-527.txt': 3, 'Doc-3642.txt': 1, 'Doc-1237.txt': 5, 'Doc-1644.txt': 2, 'Doc-398.txt': 1, 'Doc-389.tx
99.txt': 1, 'Doc-473.txt': 3, 'Doc-515.txt': 3, 'Doc-3645.txt': 1, 'Doc-531.txt': 3, 'Doc-468.txt': 3, 'Doc-3415.txt': 6, 'Doc-321.txt': 1
preston: {'Doc-293.txt': 1, 'Doc-1562.txt': 1, 'Doc-80.txt': 1, 'Doc-1557.txt': 1}
buffer: {'Doc-3756.txt': 4, 'Doc-3755.txt': 4, 'Doc-3757.txt': 4, 'Doc-1698.txt': 1, 'Doc-3754.txt': 4, 'Doc-3753.txt': 4, 'Doc-242.txt':
```

Fig. 4: Inverted Index

An inverted index is a hash of hash with each unique word as the key and its value is the another hash with key: document name and value: the count of word on that document.

Search Interface:

The search interface is implemented in HTML. It facilitates the user to input the query to retrieve the relevant pages. It displays total relevant documents and all other documents on the basis of cosine similarity of query and documents.



16 Total Number of Pages Retrieved :

1. https://my.memphis.edu/cp/home/displaylogin
2. https://my.memphis.edu/cp/home/displaylogin
3. https://ummail.memphis.edu
4. http://www.memphis.edu/afrotc/privacy-notice/index.php
5. http://ummail.memphis.edu
6. https://umdrive.memphis.edu/xythoswfs/webview/xythoslogin.action
7. http://mediasite.memphis.edu/mediasite/mymediasite
8. http://www.memphis.edu/notice/index.php
9. http://www.memphis.edu/herff/current/tutoring.php
10. http://www.memphis.edu/notice.htm

Fig. 5: Search Interface of the retrieval engine

Retrieval Engine:

The search engine uses the indexing information and retrieves the documents containing at least one of the key words. The cosine similarity between the retrieved documents and the query is calculated. The documents with higher cosine similarity score are displayed on top. i.e the pages are ranked on the basis of cosine similarity score. The cosine similarity is calculated as below.

$$\cos sim(d_j, d_k) = \frac{\vec{d}_j \cdot \vec{d}_k}{\left|\vec{d}_j\right|\left|\vec{d}_k\right|} = \frac{\sum_{i=1}^{n} w_{i,j} w_{i,k}}{\sqrt{\sum_{i=1}^{n} w_{i,j}^2}\sqrt{\sum_{i=1}^{n} w_{i,k}^2}}$$

Here the d_j is the document vector and d_k is the query vector.

As shown above in the Fig3, the Search Engine and Web Server are independent components. The Search Engine makes the index of the collected documents, open the socket and waits for the Client request. The indexing is the one time job. The Web Server has Search Interface and TCP client. When the user enter the query to Search Interface, the CGI sends the message to the TCP Client. The TCP Client sends the request to the Search Engine. The Search Engine retrieved the relevant documents and send the result back to the Client. The CGI send the result to the Search Interface to display to the user.

4. Implementation

I have implemented this project in python on window operating system. The search interface is build in HTML. The TCP client and the search engine are build in python. The CGI script is used to communicate between the search interface and the TCP client. I have hosted my Search Engine on Apache Server provided by XAMP. I developed several modules to crawl, preprocess, index rank and retrieve pages.

The crawler is implemented in *crawler.py* module. The crawler receives an URL as seed URL and visits the page and crawls newly discovered URLs. I restricted the crawler to crawl the University of Memphis' website and read the jsp, php, html pages, text and pdf files and stored all files in text format. Beautiful soup library is used to extract the URLs directly connected from the current page. The crawler used the Breadth First Search approach to crawl the pages. A hash of visited URLs is maintained to avoid the repeated crawling of the same page. The log file is created in which the message about successfully crawled or any error are written.

The URLs in the web pages can occur in different ways and forms. In order to minimize chance getting exception or 404 error from the server, the URLs should be normalized properly before crawling. There are several dynamic web pages parameters present in the URLs as GET parameters in the query string. We handled all these cases by removing the dynamic page parameters from URL's query string by throwing the substring of the URL from and after first occurrence of '?' character. We also threw the substrings from '#' character to remove page markers. It also handle the cases of invalid URLs (end with .zip, .ppt, .xls, .jpg .). Normalization of the URLs is also implemented which makes the complete link from the relative link obtained in crawling. For example, the crawler is reading the page: http://www.memphis.edu, it founds the connected page link as: /lambuth/courses /. If we do not normalize, the crawler will not be able to crawl the pages connected. The relative path /lambuth/courses/ is converted to http://www.memphis.edu/lambuth/courses/.

The module *preprocessor.py* does the preprocessing of the text files collected by the crawler. To remove the html like link I used the regular expression re.sub('**<.*?>'.** To remove the punctuations and digits and to convert all text to lower case, the regular expression re.findall('**[a-z])** is used. Porter Stemmer of NLTK library is used to stem the words. Only the text file with at least greater than 50 words are preprocessed. To remove stop words, the list of stop words is created and for each word check to see if it is on that list. If the word is not in stop wordlist then stored that word in text file. The URL of each page is also stored in the first line of preprocessed text files.

The module *indexer.py* creates an index of preprocessed documents. For each word, it makes the hash with word as key and the another hash as a value . The inner hash has document as key and no of words on that documents as value.

When the *querysender.py* module which is inside cgi-bin of XAMP received the query input from the client browser (index.html), it opens the TCP socket with the Search Engine(*search.py* module) and send the request. The *search.py* module which is waiting for the client request, retrieves the documents by using the *retriever.py* module. The retriever also calculate the cosine similarity score between the keywords and the retrieved documents. The retrieved documents are sorted on the basis of similarity score and send the result back to TCP client which forward to the client Interface.

5. Result

I have calculated the Precision, Recall and F-measure for the provided 10 queries. I have used only top 50 ranked pages as the standard reference. At first, I annotated the relevant pages out of top 50 pages which is stored in "relevant-page.xls" file. For q1, there are 24 relevant documents in top 50 pages. I assume this as the total relevant documents in collection to calculate the recall. Then, I calculated precision and recall at those ranks at which relevant documents is retrieved. The calculation is stored in file "preRec-Tara.xlxs".

1) Recall Calculation

$$\mathrm{R} = \frac{\text{Total number of relevant documents retrieved}}{\text{Total number of relevant documents}}$$

2) Precision Calculation

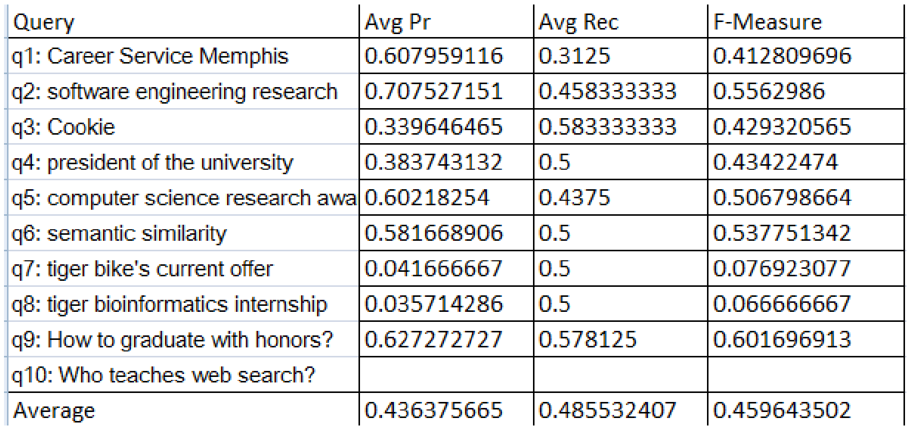
$$\mathrm{P} = \frac{\text{Total number of relevant documents retrieved}}{\text{Total number of documents retrieved}}$$

3) F - Measure Calculation

It is the Harmonic mean of Precision(P) and Recall(R) which is given by following formula.

$$F = \frac{2PR}{P+R} = \frac{2}{\frac{1}{R}+\frac{1}{P}}$$

The summaries of the result is shown below:



Query	Avg Pr	Avg Rec	F-Measure
q1: Career Service Memphis	0.607959116	0.3125	0.412809696
q2: software engineering research	0.707527151	0.458333333	0.5562986
q3: Cookie	0.339646465	0.583333333	0.429320565
q4: president of the university	0.383743132	0.5	0.43422474
q5: computer science research awa	0.60218254	0.4375	0.506798664
q6: semantic similarity	0.581668906	0.5	0.537751342
q7: tiger bike's current offer	0.041666667	0.5	0.076923077
q8: tiger bioinformatics internship	0.035714286	0.5	0.066666667
q9: How to graduate with honors?	0.627272727	0.578125	0.601696913
q10: Who teaches web search?			
Average	0.436375665	0.485532407	0.459643502

For the given queries, the engine retrieved good number of relevant pages for q2 and q. For q7 and q8, it retrieves very few number of relevant pages and for q10, it could not retrieved any relevant document up to top 50 pages.

6. Future Work

The crawler successfully crawled 12026 pages in about four hours. I believe it is not so efficient and I want to make it faster. Next point is that the crawler crawled only 12,026 pages and stops by itself. I believe there are more than 12 thousand pages in our university web site. When I checked the log file, the error message is "Exception: readContent http://www.memphis.edu/sph/people/faculty_profiles/skedia.php 'charmap' codec can't encode character '\u2010' in position 7506: character maps to <undefined>". It seems that the crawler was not able to encode some character. I need to handle those exceptions so that it will able to crawled more pages.

7. References

1) http://web0.cs.memphis.edu/~vrus//teaching/ir-websearch/

2) http://www.tutorialspoint.com/python/python_cgi_programming.html

3) https://en.wikipedia.org