Web Search Engine

Search Engine Using Page Rank

Pramodh Acharya  
Computer Science  
University of Illinois at Chicago Chicago, IL, USA  
pachar7@uic.edu

ABSTRACT

This Report is the description of components, architecture, and the functionality of the final term project for the course CS-582. The project implements a web crawler to create a search engine for the “uic.edu” domain. The search incorporates the page rank algorithm along with a cosine similarity measure to rank search results for a query. The search provides a basic user-friendly UI to input their search query and obtains results.

KEYWORDS

Search Engine, Cosine Similarity, Page Rank, Information Retrieval

1 SOFTWARE

1.1 Introduction

This Search engine is composed of 4 components, namely crawler, preprocessor, retrieval system, and the website. The application uses Pytho3 and uses object-oriented style programming leaving room for a considerable extension of components or minor/major changes in the style of processing or implementing the algorithms.

1.1 Crawler

The crawler is implemented using the “scrapy” [1] library. A custom Spider is built on top of the vanilla Spider provided by scrapy. The spider is initiated to scrape the web pages via a Custom Crawler again built on top of the vanilla Crawler provided by scrapy. The crawling uses a Breadth-First Strategy for traversing the web pages. The crawler starts at the webpage “cs.uic.edu” and is forced to stay in the “uic.edu” domain throughout. Some of the primary points related to crawling are as follows:

* The fetched documents are maintained in a HashTable data structure. The key is the URL and the value being a Document object having attributes URL, outgoing URLs, and Id and the unprocessed html text.
* The visited URLs are stored in a HashSet. A site is visited as marked once scrapy spider can crawl it.
* Once a URL is scraped and the corresponding content is downloaded, the first check is whether it is of text/html content type. Only those ages are parsed and added to the HashTable.
* Then the downloaded HTML is parsed using Beautiful Soup[3].
* The text from the relevant tags is fetched and becomes the document’s text.
* Using Beautiful Soup all the links are obtained and filtered to fetch valid ones which will be traversed next. As the href tags contain relative URL’s we ignore the #url’s as they are fragments of the same page. The valid URLs are the ones that start with http and don’t end with extensions like xlsx, docx, gif, doc, xls, jpeg, mp3, png, jpg, pdf. I also used regex to match “uic.edu” here. If a href link starts with “/” then this url part was added to the parent url. Finally, the url is canonicalized and checked if it is already visited or not.
* In canonicalization of URLs, the trailing “/” is removed, and “[www.](http://www.)” Part of the url is removed and all “http” is changed to “https”.
* At this stage after all the valid URLs from a page are fetched, these make the outgoing URLs for the parent page which is necessary for calculating page rank.
* Only valid responses like 20 and 301(redirects) are considered valid and the rest are discarded.

Once the crawl limit which is an input indicating the number of pages to be crawled has been reached then the crawler stops and returns the HashTable of the Document objects to the Retrieval system. This HashTable is stored in a pickle file for easy storage and retrieval.

1.2 Retrieval System

The retrieval system is the orchestrator and links the Crawler, PreProcessor, and the UI. This module contains the Vector Model as well as the Page rank generator. On the initial run, if the crawled documents are available at the input file location, then those are read using pickle else the Crawler is run and the documents are fetched. Then this System generates the page rank for the set of documents using the out links available for each document. After this, the vector model containing an inverted index is generated for the set of documents. This Model also exposes a method to retrieve relevant documents from the corpus for any given query.

1.3 Text Processor

The same PreProcessor from the previous assignment has been used here. It is a well-maintained Processor enabling a stemming as well as lemmatizing strategy along with the basic processing like discarding white spaces, numbers, and special characters. As usual, the stopwords and characters lower than 2 digits or greater than 20 digits are discarded. To remove stopwords nltk library corpus is used.

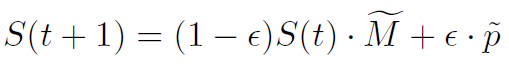
1.4 User Interface

The user interface is a simple user-friendly single page with a search bar and a navigation pane at the top. The top 10 results are shown initially with an option to see more. The interface is built using the python flask framework along with html, css and bootstrap for decorating.



1.5 Intelligent Component (Page Rank)

The page rank algorithm is used as the intelligent component. Initially, a manual implementation was used to calculate the page rank scores for each of the pages. For this, an adjacency matrix was created and the following formulae were used



This module worked fine for up to 3000 documents but had performance issues for pages greater than 5000. So the module was replaced by a python module called network[2]. For this, a directed graph is created using all the edges whose information is stored in the HashTable of Document objects where the object contains a list of outgoing links. If any outgoing links are not a key in the HashTable (Dangling links) then it is discarded, hence ending up with a graph that has nodes that are present in the HashTable. Each edge is created in the graph by iterating through the HashTable of documents and an inner loop of the outgoing links. Then the pagerank() method exposed by the network is used to generate page rank scores for the graph. The eigenvector calculation is done by power iteration till convergence (set using a threshold). The damping factor used is 0.85 and the tolerance is 1.0e-6. The top 10-page rank scores and their links are as follows.

|  |  |
| --- | --- |
| **PageRank** | **Value** |
| https://today.uic.edu | 0.01131 |
| https://today.uic.edu/events | 0.01130 |
| https://maps.uic.edu | 0.00854 |
| https://uihealth.uic.edu | 0.00831 |
| https://catalog.uic.edu/ucat/academic-calendar | 0.00807 |
| https://uic.edu | 0.00793 |
| https://emergency.uic.edu | 0.00781 |
| https://uic.edu/about/job-opportunities | 0.00754 |
| https://library.uic.edu | 0.00710 |
| https://uic.edu/apps/departments-az/search | 0.00675 |

Table 1: PageRank value of the top-10 pages in the uic.edu domain. The document corpus contains 30,000 pages.

A manual inspection of the pages vaguely shows that there are a lot of incoming links to these pages and have outgoing links to some high authority pages which indicates the page rank scores are reasonable.

2 CHALLENGES

1. One of the issues faced in the scrapy crawler was that it does not obey allowed domains rule on page redirects. Many pages from uic.edu domain redirect to other domains. It was unavoidable to stop retrieving those pages, but once retrieved they could be discarded. Hence a check is provided once a url document is downloaded to check if it is of “uic.edu” and only then proceed else discard.
2. Another redirecting related issue was that of many pages redirected to login pages inside the “uic.edu” domain which were not useful and had bad repercussions in the inverted index. This was handled to some extent by catching the login urls like “login.uic.edu” and discarding them after downloading the pages. Handling redirects was a bit difficult with scrapy but can been handled well with a little more research barring some code issues in scrapy itself.
3. Crawling takes a long time and the decision on how and where to store the crawled documents in order to avoid multiple crawls was a difficult one. There were two ways to do this. One was to store each document as one pickle file in a directory. In this scenario the outgoing links for each page had to be extracted later to calculate page rank which was less efficient w.r.t time. If page rank module is not included this is a better approach. The files can be parsed for text later when created inverted index. As our system included page rank, the second approach was taken which involved creating a HashTable in memory that stores the extracted html text, all outgoing links from a document along with its url and after the crawling stops, this HashTable is stored in a file. This process is more memory intensive, but processing page rank becomes easier. This seemed reasonable as the pages crawled were small compared to enterprise level retrieval systems.

3 WEIGHTING AND MEASURES

The

4 EVALUATION

The

5 RESULTS

The

6 RELATED WORK

Code Snippets on examples of using scrapy were referenced while coding the crawler. Articles on how page rank is used in retrieval systems were read in the process and tried testing some of them.

7 FUTURE WORK

The algorithm can include pseudo relevance feedback using Rocchio’s Algorithm which might improve the query results. The search query can be expanded to use synonyms using some well-maintained corpus.

REFERENCES

[1] <https://scrapy.org/>

[2] <https://networkx.org/documentation/stable/reference/algorithms/generated/networkx.algorithms.link_analysis.pagerank_alg.pagerank.html>

[3] <https://www.crummy.com/software/BeautifulSoup/bs4/doc/>

[4] <https://github.com/scrapy/scrapy/issues/1042>

[5] <http://www.cis.drexel.edu/faculty/cyang/papers/yang2005c.pdf>

Conference Name:ACM Woodstock conference

Conference Short Name:WOODSTOCK’18

Conference Location:El Paso, Texas USA

ISBN:978-1-4503-0000-0/18/06

Year:2018

Date:June

Copyright Year:2018

Copyright Statement:rightsretained

DOI:10.1145/1234567890

RRH: F. Surname et al.

Price:$15.00