UIC Web Search Engine

Context Pseudo-Relevance Feedback and Page Rank approach

Mirko Mantovani
Computer Science department
University of Illinois at Chicago
Chicago, Illinois
mmanto2@uic.edu

ABSTRACT

This document is a report for the final project of CS 582 Information Retrieval at University of Illinois at Chicago. The project consisted in building a web search engine for the UIC domain from scratch. The software was built modularly, starting from web crawling, going through pages preprocessing, indexing and finally adding a Graphical User Interface. Moreover, a customized "smart" component invented by us was a requisite.

I decided to experiment in the search engine optimization by using query expansion based on a pseudo-relevance feedback approach that tries to get a broad context out of the user's query, I called it Context Pseudo-Relevance Feedback (**CPRF**). This should have added mainly two types of improvements to the search engine:

- Do not focus all the results only on the words in the query but including diverse and related content to the retrieved set of pages.
- Expanding the total number of results, including web pages that do not contain any of the words present in the query but can still be of interest to the user since treat the same topic.

A Page Rank implementation is also integrated and can be turned on or off from the application UI. More details on both the smart components can be found later on in this document.

The repository containing the software can be accessed through GitHub at:

https://github.com/mirkomantovani/web-search-engine-UIC

1 SOFTWARE DESCRIPTION

1.1 Introduction

The software is written in Python3 in an Object-Oriented programming fashion to make it easily extensible for the future. In order to make it easy to download and test the code without having to perform an extensive and time-consuming crawling and page preprocessing, the dataset containing 10000 pages crawled from the UIC domain: https://www.uic.edu/, and the preprocessed pages and needed files generated (Inverted Index, TF-IDF scores, Page Ranks, Document lengths, documents' tokens, URLs) are included in the repository. In this way by cloning the repository the *main.py* can be run and in less than a second the search engine is ready to get queries in input. However, the scripts to run the crawling and preprocessing are also included and explained in the following subsections with all the other components.

1.2 Crawling

The web crawling can be done by executing the script ***multithreaded_crawling.py***, the crawling happens in parallel by using the Queue module to access the resources in a synchronized way, the number of threads can be changed by modifying the global constant *THREAD_NUMBER* in the same script, which is 20 by default, however, the main bottleneck in crawling for me was the internet download speed and not the parsing time or anything else.

The crawling starts from the UIC CS subdomain: https://www.cs.uic.edu/ specified in the ***multithreaded_crawling.py*** script.
The crawling happens with a breadth-first strategy, every page is dequeued, downloaded and parsed using the *HTMLParser* library, its links are extracted and checked to belong to the UIC domain, then added to the FIFO queue if it is in an appropriate format. A blacklist of all bad formats was derived by me while seeing the results that I was getting in the crawling and it consists in this 18 formats: ".docx", ".doc", ".avi", ".mp4", ".jpg", ".jpeg", ".png", ".gif", ".pdf", ".gz", ".rar", ".tar", ".tgz", ".zip", ".exe", ".js", ".css", ".ppt". A timeout of 10 seconds is specified as the maximum time to download a page before closing the connection and passing to the next page.
The urls are also modified after they are extracted, every initial http becomes an https, all the slashed at the end are eliminated, the query strings are removed and also the intra-page links expressed by the hash symbol (#) are also removed in order not to let the crawler believe that more pages with a different inpage-links are different urls and so different pages. Moreover, the handling of links where an <a> tag is opened and another tag is opened inside before closing href were done by splitting the link on the opening of another tag "<" in the string.

During the crawling, two dictionaries, url from code and code from url are constantly saved to disk to be used later on, the code of a web page is the number of the downloaded page in chronological order.

1.3 Preprocessing

The preprocessing of the crawled pages can be executed from the *run_preprocessing.py* script, there you can also specify the number of pages to consider and the maximum number of iterations of page rank by changing the corresponding constants *PAGE_RANK_MAX_ITER* and *N_PAGES*.

During the preprocessing, a ***CustomTokenizer*** object from the ***preprocess.py*** script is used. Each page is preprocessed by first getting the plain text in all the tags except <*script*> and <*style*>. For this step I decided to use a very fast library: *selectolax*, which is a Python API to use the Modest engine written in C. CPython would of course provide at least one order of magnitude less in computational time with respect to any HTML parses written in pure Python. The page is then tokenized, the tokens are stemmed using the *PorterStemmer* by *nltk*, the stopwords are eliminated using the list of stop-words provided in the file *stopwords.txt*, the digits are removed and the words shorter than 3 letters are not considered.

In the preprocessing the inverted index is built and the tf-idf of each word-doc pair is computed and stored in the inverted index. A directed web graph (***graph.py***) is also created and feed to the implementation of page rank in ***page_rank.py*.** All the files needed later to answer the query are then saved as binary files.

The total time taken for the preprocessing and page_rank convergence for 10000 pages was approximately 236 seconds, the page rank running time was only about 11 seconds.

1.4 Querying and Retrieving

The ***main.py*** script contains the access point to the search engine. When the program is started a CustomTokenizer object is created to tokenize the queries, a TfidfRanker object instead is instantiated in order to rank the documents based on the queries. When the documents are ranked based on a user's query, a maximum of 100 documents is considered, this constant can be change in ***main.py***: MAX_RESULTS_TO_CONSIDER, other customizable parameters are the number of results to display at a time RESULTS_PER_PAGE, the number of pages to use: N_PAGES.

1.5 User Interface

The user interface is graphical and implemented using the Python package: easygui. The GUI is an extensible module, ***CustomGUI.py*** which contains the main APIs for the basic functionalities of the program.

When the program starts the user is asked the basic settings, whether he wants to use page rank and the Context Pseudo-relevance Feedback or not. After that the main menu appears. The main menu shows the current settings of the search engine and some buttons for the main actions that can be performed. The settings can be dynamically changed at runtime by choosing the appropriate button from the main menu. The other two choices are: quitting the program and running a query. If the user presses the button to create a new query, a new window appears, and the user is prompted to insert a new query. When he clicks okay, the query is preprocessed, and the documents are ranked and the first 10 (variable in the program) documents are displayed together with the information on the preprocessed query and possibly the expanded tokens if the pseudo-relevance feedback is on. The user can now double click on a result to open the page on a new tab on the default browser of his system or can recursively press on "Show more results" at the end of the list to show 10 more results. Moreover, if the pseudo-relevance feedback is off, the user is also given the possibility to rerun the same query with the expansion.

After having run a query and decided to cancel or open a result, the user is prompted back to the main menu where he can change the setting and/or run a new query or the same one with different settings to compare the results with the previously obtained.

2 MAIN CHALLENGES

The main challenges that I have experienced during the development of this project are:

- Initially it was really difficult to choose what kind of smart component to design. I have no experience in this type of application whatsoever and thinking about improvements without having a demo to test on is just so difficult.
- During the implementation part, I spent a lot of time in learning about Python libraries and constructs that I did not know yet, like threads, how to implement or do web scraping to get the links out of an HTML page. Another thing that I had to learn from scratch was how to create a GUI in Python.
- An annoying thing was the fact that I had to crawl 10000 pages more than once, because I found in the pages various and wrong formats. In the end the entire list of formats that I had to blacklist had a size of 18 and it included ".docx", ".doc", ".avi", ".mp4", ".jpg", ".jpeg", ".png", ".gif", ".pdf", ".gz", ".rar", ".tar", ".tgz", ".zip", ".exe", ".js", ".css", ".ppt".
- A very challenging thing was the hyperparameters tuning and the integration of page rank to rank documents. The parameter "***e***" to decide how much importance to give to the tokens of the extended query is the most difficult to tune because you cannot know the average performances of your search engine if you don't have labeled data (relevant documents for each query). Also, query relevance is subjective, you never know what a person wants to retrieve, and you can only guess based on the query. I decided the *e* should be at most 0.5 and at the end I set it around 0.1-0.2, you don't want to bias the query to much by giving a lot of importance to words that are not typed by the user.

3 MEASURES AND WEIGHTING

3.1 Weighting scheme

The weighting scheme that I used was the simple **TF-IDF** of words in documents since it has been proved to be one of the most effective when it comes to web search engines and it accounts for the importance of words in each document in the correct way. I did not even think about trying another measure just because it was not the purpose of this project and this just seemed to work fine.

3.2 Similarity measure

The similarity measure used to rank documents was the **Cosine Similarity**. I implemented starting from the inner product similarity and switching to that would just be a matter of changing one line of code. I think cosine similarity is better to use because it takes into consideration the document length and the query length. It is more complex, and it usually works pretty good in practice in this type of applications so choosing the similarity measure was not really an issue, I just knew that Cosine similarity was the right one.

4 EVALUATION

I did an evaluation of the precision at 10 (only considering the first 10 results retrieved) for some random and diverse queries that I came up with. Here are the results:

- Query: *"advisor thesis"*, the first result was https://grad.uic.edu/electronic-thesisdissertation-faqs and I think that all the results were related to thesis and correlated things, I will give a precision: **p = 1.0** to this query.
- Query: *"career fair"*, the first result was https://ecc.uic.edu/career-fairs and I think that all the results were somewhat relevant to career fairs, career services, events and employment, I will give a precision: **p = 1.0** to this query.
- Query: *"research assistantship"*, the first result was http://grad.uic.edu/assistantships and I think all except the last one were related to assistantships, being it RA, TA or GA, just because the UIC domain does not have a specific page for RA, so I will give a precision: **p = 0.9** to this query.
- Query: *"internships and jobs"*, the first result was https://careerservices.uic.edu/students/internships and this time only 6 web pages were relevant, however, the ones which were not are still related to career and employment. I will give a precision: **p = 0.6** to this query.
- Query: *"student center east address"*, the first result was https://fimweb.fim.uic.edu/BuildingsData.aspx, this query is more specific and complex and in fact only from 4 results we are actually able to extract the address of the building, all the other results, however, were talking about the student center east, which is still not that bad. I will give a precision: **p = 0.4** to this query.

With an average precision of **0.78**, and the fact that every query returned at least one relevant result, I think the results are discrete.

5 INTELLIGENT COMPONENT

5.1 Page Rank

The first intelligent that I experimented with was plain and simple Page Rank. During the preprocessing of pages, the links are extracted and based on the link connections a world graph is created. The implementation of Page Rank was such that it created a strongly-connected-component with the entire graph, which means that from every node it is possible to get to another node of the graph with a non-null probability. With this interpretation there is no possibility that a random walker would get stuck in a page.

The page ranks were a bit difficult to integrate into the scoring of documents to rank them. At first, I tried to just do a linear combination of Cosine Similarity and Page Ranks and see how it worked and it was pretty bad because if the weight of page rank was too much than the homepage and other authoritative pages would always appear in the results. Instead if the weight was leaning more towards the cosine similarity then page rank would have no effect at all.

The second attempt produced pretty good results. I basically just took the first 100 results only using and discarded all they others and considered them non-relevant. Then I did the linear combination with page rank. This time it worked because it was just a different permutation of the already relevant documents, and for instance the homepage and other authoritative documents are already discarded at this point.

I think the purpose of page rank in this type of application was achieved, I was able to bias a normal search to consider more relevant more authoritative pages so that the user is more likely to find good and reliable sources of information in addition to the results that are very similar to what he types in his query.

5.2 Context Pseudo-Relevance Feedback

The Context Pseudo-Relevance Feedback (**CPRF**) I came up with this idea of my principle and customized smart component almost a month from the actual implementation. When I was implementing, I was not sure it would have worked as expected and I was really scared that I was doing all that for nothing.

When I was thinking about what kind of smart component a web search engine could have, I thought that it would be nice if we could guess the context of the user query and give him in addition to what he is specifically asking for, some results that are very related to what he searched for. A simple static query expansion based on synonyms would have been too simple and would not have been able to capture contents that are related but have a different semantic.

However, the starting point always has to be the user's query, as there is nothing else except that initially. I really liked the idea of how pseudo-relevance feedback was able to let you reformulate your query in an autonomous way. Of course an explicit relevance feedback when the user is asked which other words he would want to include in his query would probably be better, but at the same time it could be annoying for him to having to respond to some questions while searching. A pseudo-relevance feedback seems a more appropriate and easy way to run in background some more complex query that the user is not even aware of.

To extract some words that could express the context of the formulated query I decided to use some intrinsic information that the initially retrieved documents have. In particular, I think that the words with highest TF-IDF in a document could represent the topic of that document, and what it differentiates from the others, because TF-IDF is high when the word is not contained in many documents but it's recurrent in that document.

The process to extract the context words is the following: from the ranked documents I take NUMBER_DOCS_EXPANSION documents, which I set to 30 but could be changed in ***pseudo_relevance_feedback.py.***, for each document I take the tokens with highest TF-IDF (constant NUMBER_TOP_TOKENS) and for each distinct token in them I sum all the TF-IDF of this word of each document if the word is present. At the end I rank the tokens and the top ranked words are the common words of each retrieved document that represent the context of the query. I then return the n expanded words (constant NUMBER_EXPANSION_TOKENS) set, to which the original's query tokens are subtracted in order not to have repetitions and give too much importance to those words.

The expanded tokens will be given a difference weight, less than the original words in the query, this E_CONST can be changed in the ***statistics.py*** script.

5.3 Comparison with plain search engine

Based on the queries that I tried think the intelligent component really gives some improvements in some cases.

The first big results that works always is the ability to expand the sets of retrieved documents, in fact, if only a few documents contain any of the word in the query, the plain search engine would only retrieve those few documents. Instead, with the **CPRF** smart component the sets of results will be way bigger and could possibly always lead to a retrieved set whose size is more than 100 documents.

Another positive result that I noticed in some queries is that it actually finds what I was looking for as first result whereas the simple search engine does not even find it in the top ten results. An example of this can be observed by searching *"computer science courses"*, what I wanted to find is a list of all the main courses offered by the computer science department at UIC. This is the first result retrieved by the engine when the intelligent component is active. Instead, without activating it, that page is not in the first results.

Lastly, I tried to search for *"information retrieval"*, the search engine without **CPRF** was very bad, the first result is a search page for the departments of UIC, this is also probably because there is no page for Information Retrieval in the UIC domain or it's just not included in the domain. However, with the smart component on, many web pages related to this were found, 2 of the extended words were *"Cornelia Caragea"*, the professor who is teaching this course, so many pages were indeed related to her publications and her work in Information Retrieval, this is also a very good property of the search engine. In case very relevant results cannot be found, it is still able to find the best possible results about related things.

6 RESULTS

As explained in 5.3 when I did the comparison between the plain search engine and using the smart component, I think the produced results were pretty good and I got what expected when I was thinking about this. The summarized good things that I noticed by testing it were:

- The ability to find many more results even when the original query outputs only a bunch of websites.
- The ability of finding what I was actually looking for as first results, for instance in the queries *"computer science courses"*, like I explained in the paragraph 5.3.
- The very nice property of retrieving discrete content even where the corpus does not contain pages which are very relevant to the query and possibly what the user is searching for, I noticed this for the query *"information retrieval"* as explained in paragraph 5.3.

One way that showed me that this was producing the correct results was the fact that in the top 10 expanding words representing the context, some words belonging to the user's query were present. This shows how the method is effectively able to capture the topic and there is no other way to describe the other words in that set if not as words belonging to the same context as the ones in the original query.

Talking about page rank instead. I think it does its job, but it does not change the rankings too much in the way I implemented, it's just a way to bias the results to prefer more authoritative pages if there are any.

7 FUTURE WORK

For sure one of the things that has to be addressed from here is the hyperparameters tuning. It is the most difficult thing to do mainly because of the lack of labeled data. I cannot know which value of a parameter works better if I cannot evaluate the precision of the queries, and in order to tune it automatically, there should be a lot of already labeled data.