# COMP4321 Final Project Report

Name: Ling Yan Therese

Student ID: 20775871

Link to Presentation Video: https://youtu.be/J425L3JEB3g

# Overall Design

This project was done in Python 3.8.2. The application consists of mainly 3 parts.

* The crawler and indexer, that recursively fetches webpages in a BFS manner, stores words, information and child links etc. and other information about the page into the indexes;
* The first part of the retrieval function that does the calculation for the term weights in the document indexes
* The second part of the retrieval function tokenizes and calculates query term weights, and calculates the cosine similarity between query term weights and the already calculated document term weights, then ranks them.
* The web server that receives input query, sends the query to the retrieval function and displays results from the retrieval function.

# File Structure

There are a total of 12+2 indexes. 12 are created when running the crawler, to store information and index words, links, and child parent link relationships from the webpages, 2 are created when pre-calculating each document’s normalization term in the denominator of the cosine similarity equation, for title words and content words respectively. All indexes and calculated terms are stored as a sqlitedict into a sqllite database.

#### **parentID\_childID**

> Relationship: parentID -> childID

> Format: {parentID:[childID, childID]}

It is a dictionary containing parentID to childID mapping. Given a parent link’s page ID, we may obtain a list of child link ID.

#### **childID\_parentID**

> Relationship: childtID -> parentID

> Format: {childID:[parentID, parentID]}

It is a dictionary containing childID to parentID mapping. Given a child link’s page ID, we may obtain a list of parent link IDs.

**pageID\_url**

> Relationship: pageID -> url

> Format: {pageID: url}

It is a dictionary containing pageID to url mapping. This is implemented so when we want to fetch the url given a pageID, we may reference this index.

**url\_pageID**

> Relationship: url -> pageID

> Format: {url: pageID}

It is a dictionary containing url to pageID mapping, it is the inverse of pageID\_url. Implemented so when given a url, we can obtain the pageID.

**forwardidx**

> Relationship: pageID -> [word, word frequency]

> Format: {pageID:[[word1, freq1], [word2, freq2]]}

It is a dictionary containing pageID to [word, word frequency] mapping. This is done so we may obtain a list of a page's words and their frequencies given a pageID.

**inverseidx**

> Relationship: wordID -> [pageID, word frequency, tfidf of word in page]

> Format: {wordID: [(pageID1, freq1, tfidf1), (pageID2, freq2, tfidf2)]}

It is a dictionary containing wordID to [pageID, word frequency, tfidf] mappping. It is the inverse of forwardidx. This is done so given a wordID, we may get get the pages that contain the word, and their word frequencies. Initially after running the crawler, the tfidf of each word for each document is not yet calculated. Hence the format of the index before calculating term weights for words in the documents is {wordID:[[pageID1, freq1], [pageID2, freq2]]}. After calculating the tfidf of the words in tfidf.py, the format of the index will be as the

{wordID: [(pageID1, freq1, tfidf1), (pageID2, freq2, tfidf2)]} format.

**wordID\_word**

> Relationship: wordID -> word

> Format: {wordID: word}

It is a dictionary containing wordID to word mapping. It stores the wordID to word relationship of words in the content section of the page. Implemented to fetch a word given a wordID.

**word\_wordID**

> Relationship: word -> wordID

> Format: {word: wordID}

It is a dictionary containing word to wordID mapping. It stores the word to wordID relationship of words in content section of the page. It is the inverse of wordID\_word. Implemented to fetch a wordID given a word.

**title\_titleID**

>Relationship: title word -> titlewordID

> Format: {titleword: titlewordID}

It is functionally similar to word\_wordID. It stores the titleword to titlewordID relationship of words in title section of the page only.

**titleID\_title**

>Relationship: titlewordID -> title word

> Format: { titlewordID : titleword}

It is functionally similar to wordID\_word. It stores the titlewordID to titleword relationship of words in title section of the page only.

**Inversetitleidx**

>Relationship: titlewordID -> [pageID, word frequency, tfidf]

> Format: {titlewordID: [(pageID1, freq1, tfidf1), (pageID2, freq2, tfidf2)]}

Functionally similar to inverseidx; stores title word frequencies. Initially after crawling, the format of the index does not have the tfidf for each word indexed. After calculating the tfidf score of title words in tfidf.py, the score appended to each entry.

**Forwardtitleidx**

>Relationship: pageID -> [titlewordID, word frequency]

> Format: {pageID: [titlewordID, word frequency]}

Functionally similar to forwardidx; given a pageID, stores titlewordID and title word frequencies.

**pageID\_elem**

> Relatinoship: pageID -> (title, modification date, size)

> Format: {pageID: [title, modification date, index date, index date, size]}

It is a dictionary containing pageID to [title, modification date, size, index date] mapping. This is so given a pageID, we may fetch the metadata(title, mod date, size) of a page.

*Created Later:*

**Bodynorm**

>relationship: pageID -> normalization value in denominator for a page, content section only

>Format: {pageID: bodynorm}

Stores the normalization value in denominator for a page, calculated using content words only. It is pre-calculated to streamline the cosine similarity calculations done later.

**Titlenorm**

>relationship: pageID -> normalization value in denominator for a page, title section only

>Format: {pageID: titlenorm}

Stores the normalization value in denominator for a page, calculated using title words only. It is pre-calculated to streamline the cosine similarity calculations done later.

|  |  |  |
| --- | --- | --- |
| Name | Content | Created in crawler.py? |
| parentID\_childID | {parentID:childID} | yes |
| childID\_parentID | {childID: parentID} | yes |
| pageID\_url | {pageID: url} | yes |
| url\_pageID | {url: pageID} | yes |
| forwardidx | {pageID: [[wordID, freq], [wordID, freq]] } | yes |
| inverseidx | {wordID: [[pageID, freq, tfidf], [pageID, f, tfidf]] } | Yes(tfidf added in tfidf.py) |
| wordID\_word | {wordID: word} | yes |
| word\_wordID | {word: wordID} | yes |
| title\_titleID | {titleword: titleID} | yes |
| inversetitleidx | {titlewordID: [[pageID, freq, tfidf], [pageID, f, tfidf]]} | Yes(tfidf added in tfidf.py) |
| forwardtitleidx | {pageID: [[titlewordID, freq], [titlewordID, freq]] } | yes |
| pageID\_elem | {pageID: [title, moddate, index date, size]} | yes |
| bodynorm | {pageID: content norm value} | No (in tfidf.py) |
| titlenorm | {pageID: title norm value} | No (in tfidf.py) |

# In-depth Description and Features:

#### **Crawler**

Diagram

Description automatically generated

The indexing of pages is done by running crawler.py. It crawls the pages and their child links in a recursive, BFS way. This is done by storing child links into queue, but only if they haven’t already been indexed(check against forwardidx). When a page is indexed, we queue it’s child links into the back of the queue; and after indexing, we pass the top most link in the queue to the function, as well as the queue with the top most item removed.

After fetching a page, the crawler mainly obtains 3 kinds of information about the pages. The meta information of the page, the child links of the page, and the text of the page.

Meta Information:

The crawler obtains the size, last modification date, title from the page, then records the datetime the crawler crawled this page, and saves it as index date. All of these variables are then saved to pageID\_elem with the page’s pageID. We use index date and modification date to filter out pages we have already indexed before, and do not need to be indexed again(as they have not been modified).

Before each page is indexed, it is run through a filter to see if it is eligible for indexing. If the page has been indexed before, and if the saved index date is later than the modification date, then the page does not need to be indexed, and we may index the next item on the queue. But if the moficiation date is later than the index date, then the page needs to be reindexed. We clean up the page’s related indexes(mod\_cleanup()) before re-indexing the page.

Child Links:  
The crawler obtains the child links of the page, and indexes it into childID\_parentID, parentID\_childID, pageID\_url, url\_pageID. Then it queues the child links into the back of the queue, if the child link has not already been indexed.

Text information:

The crawler fetches the title, and the content text of the page separately. It then cleans, tokenizes, stems and crate bi, trigrams of those words, and indexes them to inverseidx, forwardidx, wordID\_word, word\_wordID, inversetitleidx, forwardidx.

Proccess of cleaning the words:

* First the text tokenized with nltk.tokenize’s Whitespace tokenizer, then it is converted to lower case, punctuation is removed. (pre-proccess\_text())
* Each token is then transformed into it’s stem by using the **Porter’s Algorithm**(/tools/porter.py).
* Then the stemmed tokens are saved as unigrams. The Unigrams are transformed into bigrams and trigrams(with the stopwords).
* The unigrams are then separately removed of stopwords.
* Unigrams and bigrams and trigrams are concatenated into one list and returned as the page’s tokens.

Bigram and Trigrams: (/tools/ngrams.py)

In this project, word position is not used, instead, to enable phrase search, bigrams and trigrams have been implemented. To not overpopulate the indexes with sparse words, Grammatical filtering has been used to only create birgram and trigrams of useful n-grams.

Word tokens are first POS tagged using nltk.tag, then Bigrams and trigrams are created from a sequence of words if:

*Bigram:*

First Word is any of: "NN","NNS", "NNP", "NNPS", "JJ", "JJR", "JJS"

and

Second Word is any of: "NN","NNS", "NNP", "NNPS"

*Trigram:*

First word is any of: "NN","NNS", "NNP", "NNPS", "JJ", "JJR", "JJS"

and

Second Word is any

And

Third Word is any of: "NN","NNS", "NNP", "NNPS"

***Retrieval Function: tfidf.py+cosinesim.py***

In this project, the retrieval function is separated into two parts, one to calculate the term weights of the documents; and another to calculate the term weights of the query+calculate the most similar documents to the query.

1.Document Term weights (tfidf.py)

The term weights of the words in documents are calculated by running tfidf.py. The calculated term weights are stored in inverseidx and inversetitleidx: {pageID:[ [wordID, freq, tfidf] ]}.The formula used to calculate the tfidf of each word in each document is as follows.

Where is the term frequency of the most frequent term in the document. And N is the total number of documents. Term weights are calculated for title words, and content words separately, and stored in inverseidx, and inversetitleidx respectively.

As seen in the formula, tf of a word is normalized by , such that the document length would affect the calculation of the term weight less.

In (tfidf.py), the document body norm and title norm are also pre-calculated such that we may lookup the value when calculating the cosine similarity between query and documents later. The Document body/tite norm is used in:

Where DocNorm is the root of the sum of the term weight of all title words/content words in document squared. DocNorm is replaced with TitleNorm or BodyNorm, depending if we are calculating tfidf for title words, or content words.

Where is the term weight of the title word/content word in their document.

2. Calculate the query term weights of the query + Calculate and find similar Docs(cosinesim.py)

For the second part of the retrieval function, it is run after the user submits a query to the webpage. The webpage receives the input, and passes the query text to functions in cosinesim.py. After processing in cosinesim.py, the webpage then receives a dictionary output of the highest ranking pages and their information. It is then outputted to the webpage, ranked in descending order.

In cosinesim.py, we first tokenize, lower case, remove punctuation, stemmed using Porter’s algorithm, then removed of stopwords, these tokens are saved as unigrams. Only text that is quoted(‘this is quoted’) will be regarded as a phrase, and separately saved as bigrams, trigrams and also unigrams. Frequency of query is also counted. Then the tfidf of the query words are counted, with this formula:

Where is calculated using the content words’s index, using inverseidx. The the query words, their frequencies and the tfidf is scored in this format:

{word:[freq, tfidf], word:[freq, tfidf]}

We then calculate Cosine Similarity score between doc and query when using term weights of title words, and Cosine Similarity score between doc and query when using term weights of content words.

The formula used is:

Where is the or the of the page, obtained by looking up the bodynorm and titlenorm db.

After obtaining the CosineSimlarity title score and CosineSimlarity content score, we weight the title score heavier, and concatenate them into the final document score.

We weigh the title score heavier, as we want to prioritize and rank pages that contain title

The documents’ scores and page IDs are then sorted in descending order according to final scores, and saved to a list(rank\_list). The list is then passed into another function(fetch\_info()) to obtain the page info, and send back to the webpage in score descending order, in this format:

page1ID: ["page1 score", "page title1",'url1', 'lastmoddate1', 'size', "word1 freq1; word2 freq2;", ["childlink1", "c2"], ["parentlink1", "p2"]]

# Installation procedure

**1. Setup preferred Virtual env (optional)**

- Packages must be installed, loading them into a virtual env would make it easier to delete them later.

**1.5 Install pip (optional)**

- Only required if pip is not already installed.

- Mac/Linux: `python get-pip.py`

- Windows: `py get-pip.py`

**2. Pip install packages**

- Using pip, install packages (into virtual env).

- `pip install sqlitedict bs4 requests nltk python-dateutil django`

**3. Run crawler.py to crawl page**

- `python crawler.py`

**4. Run tfidf.py to calculate document term weights**

- ‘python tfidf.py’

**5. move into mysite dir**

- `cd mysite`

**6. Runserver with Django**

- `python manage.py runserver`

- Search page is at: http://127.0.0.1:8000/

Notable Packages Used:

Nltk, Django, sqlitedict, BeautifulSoup, bootstrap.

# Conclusion:

All in all, I believe I’ve Implemented a fairly basic Seach page. It has moderate indexing time and search time, the search time reduced due to pre-calculating the normalisation value in the Cosine Similarity formula needed during search. But the Search page lacks features, such as query reforming, page ranking etc. In addition, word position is not saved in the indexes, impacting phrase search ability beyond bigrams and trigrams.

The easiest feature I would love to implement would be a pagerank system. Given we already have child links and parent links, it would be relatively easy to implement pageRank.

If given even more time, I would love to attempt to personalize the webpage, to be able to suggest recommended webpages, and be able to reform queries after the user inputs a query. Although I am unfamiliar with the functionality of cookies, Chapter 13 did introduce the click based preference/user profile mining technique, and I imagine the trained user vector can then be used to re-rank the pages.

#### Testing Test of cosinesim.py outside of the webpage, by using query: “Movies for Kids”. The final output is in the format sent to the webpage to display results.

Text

Description automatically generated

*Testing of porter.py: tested manually implemented porter stemmer against nltk’s nltk.stem PorterStemmer.*

Text

Description automatically generated

*Testing ngrams.py: inputed phrase "The hong kong university of science and technology is reknowed for it's high quality students" to see if function returns bi-trigrammed tokens.*

Text

Description automatically generated