UCI Information Retrieval Assignment 3

GOAL: To implement a search engine.

Introduction

This assignment is to be done in groups of 1, 2, 3 or 4. You can work on the same groups that were in place for the crawler Project if you wish to. Although this is presented as one single project here, it is internally it is organized in 3 separate milestones, each with a specific deadline, deliverables and score.

In doing milestones #1 and #2, make sure to consider the evaluation criteria not just of those milestones but also of milestone 3 — part of the milestones' evaluation will be delayed until the final meeting with the TAs.

You can use code that you or any classmate wrote for the previous projects. You cannot use code written for this project by non-group-member classmates. You are allowed to use any languages and libraries you want for text processing, including nltk. **However, you are not allowed to use text indexing libraries such as Lucene, PyLucene, or ElasticSearch**.

To accommodate the various skill levels of students in this course, this assignment comes in two flavors:

1 **Information Analyst**. In this flavor there is some programming involved, but not much more advanced than what you already did so far. It is a mixture of a Text Processing project and stitching things together. You will use a small subset of crawled pages. Groups where **ALL students are NOT CS nor SE/Informatics majors can choose this**.

2 **Algorithms and Data Structures Developer**. In this flavor there is more involved programming, and your code needs to perform well on the entire collection of crawled pages, under the required constraints. **This option is available to everyone, but groups that have at least one CS or SE/Informatics student are required to do this.**

Milestones overview

MS	Goal	Due date	Deliverable	Score
MS #1	Initial index	12th Feb	Short report + Code	2.5
MS #2	Boolean retrieval	19th Feb	Short report + Code	2.5
MS #3	Complete search	05th Mar	Report + Code + Demo	55.0

General specifications

You will develop two separate programs: an indexer and a search component.

Indexer

Create an inverted index for the corpus with data structures designed by you.

- **Tokens**: all alphanumeric sequences in the dataset.
- **Stop words**: do not use stopping while indexing, i.e. use all words, even the frequently occurring ones.
- **Stemming**: use stemming for better textual matches. Suggestion: Porter stemming, but it is up to you to choose.
- **Important text**: text in bold (b, strong), in headings (h1, h2, h3), and in titles should be treated as more important than the in other places. *Verify which are the relevant HTML tags to select the important words.*

Search

Your program should prompt the user for a query. **This doesn't need to be a Web interface, it can be a console prompt**. At the time of the query, your program will stem the query terms, look up your index, perform some calculations (see ranking below) and give out the ranked list of pages that are relevant for the query, with the most relevant on top. Pages should be identified **at least** by their URLs (but you can use more informative representations).

- **Ranking**: at the very least, your ranking formula should include tf-idf, and consider the important words, but you are free to add additional components to this formula if you think they improve the retrieval.

Extra Credit

Extra credit will be given for tasks that improve the retrieval and the user search experience - except for GUI, you must code from scratch. For example:

- Detect and eliminate duplicate pages. (1 point for exact, 2 points for near)
- Add HITS and Page Rank to ranking. (1.5 point HITS, 2.5 for PR)
- Implement 2-gram and/or 3-gram indexing and use it during retrieval. (1 point)
- Enhance the index with word positions and use that information for retrieval. (2 points)
- Index anchor words for the target pages (1 point).
- Implement a Web or GUI interface instead of using the console. (1 point for the local GUI, 2 points for a web GUI)

Additional Specifications for Information Analyst

Option available to all groups formed **only** by non-CS and non-SE students.

Programming skills required

Intro courses.

Main challenges

HTML and JSON parsing, read/write structured information from/to files or databases.

Corpus

A small portion of the ICS web pages (analyst.zip).

Indexer

You can use a database to store the index, or a simple file – whatever is simpler to you. If you store it in a file, the index is expected to be sufficiently small, so that it fits in memory all at once.

Search interface

The response to search queries should be less than 2 seconds.

Note

This project can be a great addition to your résumé and job interviews:

- **Tired**: "Wrote a Python script that finds words in Web pages."
- **Wired**: "Wrote a search engine from the ground up that is capable of handling two thousand documents or Web pages."

Additional Specifications for Algorithms and Data Structures Developer

Option available to all students. **Required for CS and SE students.**

Programming skills required

Advanced.

Main challenges

Design efficient data structures, devise efficient file access, balance memory usage and response time.

Corpus

A large collection of ICS web pages (developer.zip).

Indexer

Your index should be stored in one or more files in the file system (no databases!).

Search interface

The response to search queries should be $\leq$ 300ms. Ideally it would be $\lesssim$ 100ms, but you won't be penalized if it's higher (as long as it's kept $\leq$ 300ms).

Operational constraints

Typically, the cloud servers/VMs/containers that run search engines don't have a lot of memory. As such, you must design and implement your programs as if you are dealing with very large amounts of data, so large that you cannot hold the inverted index all in memory. Your indexer must off load the inverted index hash map from main memory to a partial index on disk at least 3 times during index construction; those partial indexes should be merged in the end. Optionally, after or during merging, they can also be split into separate index files with term ranges. Similarly, your search component must not load the entire inverted index in main memory. Instead, it must read the postings from the index(es) files on disk. **The TAs will verify that this is happening.**

Note

This project is a great addition to your résumé and job interviews:

- **Tired**: "Wrote a Web search engine using ElasticSearch."
- **Wired**: "Wrote a search engine from the ground up that is capable of handling tens of thousands of documents or Web pages, under harsh operational constriants and having a query response time under 300ms."

Milestone 1

Goal: Build an index

Building the inverted index

Now that you have been provided the HTML files to index, you may build your inverted index off of them. The inverted index is simply a map with the token as a key and a list of its corresponding postings. A posting is the representation of the token's occurrence in a document. The posting typically (not limited to) contains the following info (you are encouraged to think of other attributes that you could add to the index):

- The document name/id the token was found in.
- Its tf-idf score for that document (for MS1, add only the term frequency).

Some tips:

- When designing your inverted index, you will think about the structure of your posting first.
- You would normally begin by implementing the code to calculate/fetch the elements which will constitute your posting.
- Modularize. Use scripts/classes that will perform a function or a set of closely related functions. This helps in keeping track of your progress, debugging, and also dividing work amongst teammates if you're in a group.
- We recommend you use GitHub as a mechanism to work with your team members on this project, but you are not required to do so.

Deliverables

Submit your code and a report (in pdf) to with the following content:

- a table with assorted numbers pertaining to your index. It should have, at least the number of documents, the number of [unique] tokens, and the total size (in KB) of your index on disk.

Note for the developer option: at this time, *you do not need* to have the optimized index, but you may save time if you do.

Evaluation criteria

- Did your report show up on time?
- Are the reported numbers plausible?

Milestone 2

Goal: Develop a search and retrieval component

At least the following queries should be used to test your retrieval:

1 cristina lopes

2 machine learning

3 ACM

4 master of software engineering

Developing the Search component

Once you have built the inverted index, you are ready to test document retrieval with queries. At the very least, the search should be able to deal with boolean queries: AND only.

If you wish, you can sort the retrieved documents based on tf-idf scoring (you are not required to do so now, but doing it now may save you time in the future). This can be done using the cosine similarity method. Feel free to use a library to compute cosine similarity once you have the term frequencies and inverse document frequencies (although it should be very easy for you to write your own implementation). You may also add other weighting/scoring mechanisms to help refine the search results.

Deliverables

Submit your code and a report (in pdf) to Canvas with the following content:

- the top 5 URLs for each of the queries above
- a picture of your search interface in action

Note for the developer option: at this time, *you do not need* to have the optimized index, but you may save time if you do.

Evaluation criteria

- Did your report show up on time?
- Are the reported URLs plausible?

Milestone 3

Goal: Develop a complete search engine

During this last stretch, you will improve and finalize your search engine.

Come up with a set of at least 20 queries that guide you in evaluating how well your search engine performs, both in terms of ranking performance (effectiveness) and in terms of runtime performance (efficiency). At least half of those queries should be chosen because they do poorly on one or both criteria; the other half should do well. Then change your code to make it work better for the queries that perform poorly, while preserving the good performance of the other ones, and while being as general as possible.

Deliverables

- Submit a zip file containing all the programs you wrote for this project, as well as a document with your test queries (no need to report the results). Comment on which ones started by doing poorly and explain what you did to make them perform better.
- A live demonstration of your search engine for the TAs. You are expected to share your screen and guide the TAs through the code that you wrote, and also to turn on your video.

Evaluation criteria

- Does your search engine work as expected of search engines?
- How general are the heuristics that you employed to improve the retrieval?
- Is the search response time under the expected limit?
- Do you demonstrate in-depth knowledge of how your search engine works? Are you able to answer detailed questions pertaining to any aspect of its implementation?

Note for the developer option: at the end of the project, you should have the optimized index that allows you to run both the indexer and the search with small memory footprint, smaller than the index size.

Understanding the dataset

In Assignment 2, your crawlers crawled the many web sites associated with ICS. To avoid you need to crawl again and save you some time, we collected a chunk of these pages and are providing them to you.

Data files

There are two zip files: **analyst.zip** and **developer.zip**. The names should be self-explanatory: the former is for the analyst flavor of the project; the latter is for the algorithms and data structures developer option. The only difference between them is the size: analyst.zip contains only 3 domains[1] and a little over 2,000 pages, while developer.zip contains 88 domains and little under 56,000 pages. The following is an explanation of how the data is organized.

Folders: There is one folder per domain. To simplify your work, each JSON file inside a folder corresponds to one web page (*note that you would not do this in an operational environemnt*).

Files: The files are stored in JSON format, with the following fields:

- **url**: contains the URL of the page. Ignore the fragment parts, if you see them around.
- **content**: contains the content of the page, as found during crawling
- **encoding**: an indication of the encoding of the webpage.

Broken or missing HTML

Real HTML pages found out there are full of bugs. Some of the pages in the dataset may not contain any HTML at all and, when they do, it may not be well formed. For example, there might be an open `<strong>` tag but the associated closing `</strong>` tag might be missing. While selecting the parser library for your project or writing your own, please *ensure* that it can handle broken HTML.

[1] *Domains*, for the purposes of this project.