Wiki-Search: A Wikipedia Search Engine

Luna Szymanski

ABSTRACT

When I began this project I was planning to create a robust web app that incorporated traditional UI designs from other search engines to display wikipedia articles in a new and interesting way. What I wanted to accomplish was an easy to use system for accessing and reading multiple articles at a time. Throughout the course of executing this project my goal remained the same: I wanted a platform I could see myself using to quickly research a topic. However, as the project progressed I shifted my attention to creating an easy to use, lightning fast, and forgiving search engine. Instead of a robust UI I focused on always returning the right articles and doing so very quickly.

PROBLEM

I definitely remember the days when Wikipedia use was frowned upon in academia, but those days are long gone. Wikipedia has established itself as the leader in online encyclopedias and is widely accepted as a credible source of information. However the Wikipedia search bar is not the most useful for searching Wikipedia. I know at least for the circle of people around me I see that people unanimously use Google to search for the right Wikipedia articles.

One reason that the Wikipedia search function is not very useful is that it only returns one result. Instead of seeing a list of articles, it basically returns the single article with a title that most directly matches your query. There are a variety of reasons for why this is not always useful. First of all it is possible to have articles with subsections entirely dedicated to your query topic, and these results would never be prioritized over an article with a matching title. For example, if I search [anarchism] I get the article titled “Anarchism,” but I do not get the articles that discuss Russian history and the role of Anarchism in that history. Returning one result is also not particularly useful if you do not know the exact word of what you are searching, or perhaps you do not know how to spell it. The current Wikipedia search engine basically assumes you have some knowledge about the information you are looking for, which I imagine is not always the case for a user of an encyclopedia search engine.

Another aspect of the current Wikipedia search engine is that it is impossible is quickly access multiple articles at once. You currently need to make a separate HTTP request for every single article you want to read. Since most articles are almost purely text it would not be expensive to load a bunch of related articles at once and present them to the user, which is what I chose to do in my own implementation.

DATASET

The dataset I used is found on Kaggle, entitled “English Wikipedia Articles 2017-08-20 SQLite.” This database is 20gb and includes cleanly formatted articles with the following fields: ARTICLE_ID, TITLE, SECTION_TITLE, and SECTION_TEXT. The ARTICLE_ID field stores a unique ID for each article, and the TITLE field contains a String with the title of the article. The SECTION_TITLE and SECTION_TEXT are there for every section in the article and is how the article is structured.

One of the first challenges of this project for me was dealing with this database. I downloaded it and found a ".db" file that I had never encountered before. I quickly realized that I needed to convert this .db file into either a .csv or a .json to make it usable. I chose to stick with a CSV file since I am not familiar with JSON files or SQL. However when I converted the database into a CSV file I found that no program could open it because of how large it was, still ~20gb. It would also be very expensive to load my entire file into Solr, so I decided to split the CSV file into 10 parts of 2gb each.

After splitting the database I realized that the dataset was organized such that the first few GB had the majority of the robust text articles, and the later parts of the file had supplemental information like citations and less popular topics. This was very fortunate for me because even though my slice of the database is only about 10% of the total size, the slice I chose is enough to cover most topics. My results would be a lot less interesting if for example the database was organized alphabetically because then there would be a large portion of the alphabet for which my search engine cannot handle any queries.

In total my final database in Solr has 1,071,093 articles, which for the purpose of this search engine proved to be plenty. In the SECTION_TEXT field, which was my largest field, I found 2,077,085 unique words. In the TITLE field I found 86,157 unique words. On average each SECTION_TEXT field is 25 times larger than each TITLE field which makes sense for a database of articles. Each article is also composed of multiple sections so the actual length of each article is significantly more than 25 times the length of each title.

GUI

Throughout the course of working on this project, my plans for my GUI changed the most. I really wanted to have a GUI that was robust, and had multiple features. Some of the features I thought would be very useful for a Wikipedia search engine were term highlighting, proposing similar articles, an outline of the article, and pagination. I also wanted to put it online so that it was usable from any computer and not just my own. The majority of these features proved to be too difficult to implement because of my lack of familiarity with JavaScript and React. After realizing this challenge I audibled to using JavaFX because it would still let me accomplish the core of my project without having to learn anything about web development.

JavaFX is a java library that lets you create simple GUIs. It is very easy to begin using and intuitive, but also is surprisingly robust with features. Instead of trying to incorporate a variety of features I stuck to some of the more simple elements such as scenes and stages to build my GUI. See figure 1.1 for a sample of the GUI.

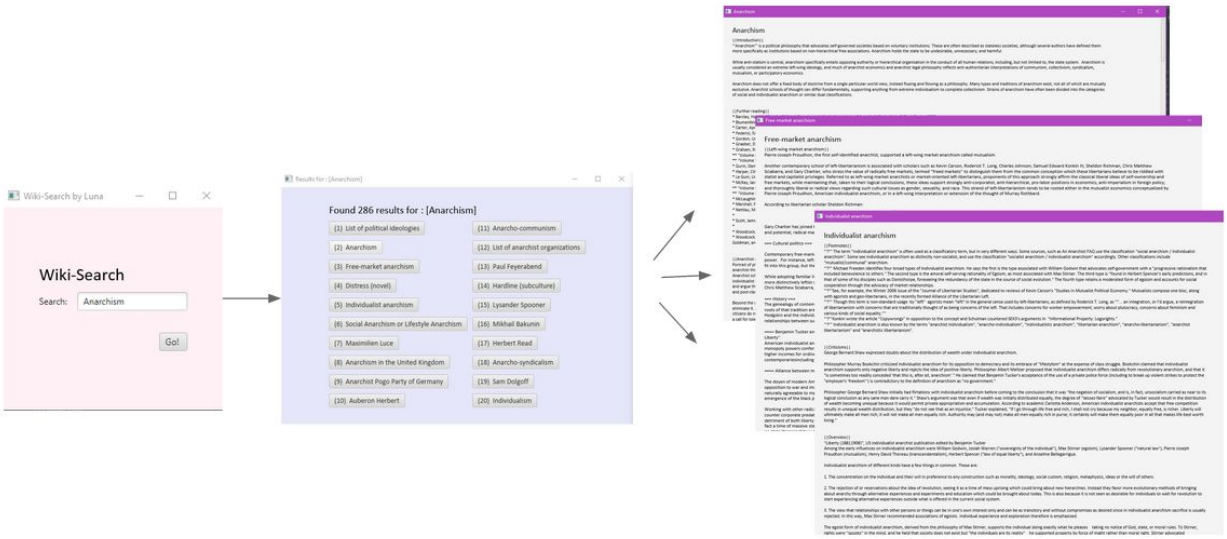

Figure 1.1: First window is the Search Box, second window is the Results List, and third window is the Article View window.

I decided I only needed three different windows to display my results. The first window is what you see when the program is booted up and is the pink colored home page with a single search bar. I chose to keep this simple because many successful search engines like Google do this and I think it makes for very intuitive navigation. Also it was simple to implement without any bugs and with my limited GUI-related experience.

After searching for a query using either the “Go!” button or the enter key, I chose to open a new window with the results. This makes it easy to search for additional queries without having to navigate back to the search bar, and avoids cluttering the results window. On the back end the articles are all built and ready to go as soon as the results list is loaded, and this happens really quickly. The results list contains a list of buttons with article titles and each button works as a link to open the article viewer window. I chose to also display the amount of results found on the top but other than that I kept it very simple to make sure I didn’t overextend with the UI tech and ended up with a working, bug free UI. Also from a users perspective, a simple UI like this is very intuitive and instinctive so everyone can use my search engine without me explaining it.

The third window I implemented is the article viewer window that opens up when you click on a hit. In this window I reconstructed each article into an easy to read document. I used the “|” character to surround each section title because that character was not common in my collection and made it clear when the article was shifting to a new section.

Even though my GUI ended up very simple, it is very efficient for the specific task I designed it for: searching and returning wikipedia articles. It is designed to be easy to use and fast at the cost of additional functionality.

BACKEND

The code for this project is structured into two files: wiki.java and wikiFX.java. Wiki.java is responsible for communicating with solr, specifically for running queries. WikiFX.java is responsible for taking the information that Solr gives me and feeding it into JavaFX to create a GUI. Outside of these files I used bin/post commands to send my segmented CSV files to Solr. The reason I used bin/post is it was really easy to specify the seperator i used when building my CSV file: "^". This ensures that Solr read my data in the proper way without having to modify the schema.xml. I felt like because of my lack of experience with XML it was wise to do as much work as I can outside of the XML files.

Wiki.java

Wiki.java is primarily designed to query Solr. In this there are two main functions I use in combination to generate results: runQuery() and runQuerySP().

```java
/**
 * This function runs a query on the specified field using phrase querying
 *
 * @param solr solr Instance
 * @param q the query
 * @return the response from Solr
 * @throws SolrServerException
 * @throws IOException
 */
public static QueryResponse runQuery(HttpSolrClient solr, String q) throws SolrServerException, IOException {
    SolrQuery query = new SolrQuery();
    query.setRows(500);

    //tokenize query
    String tokenized = tokenizer(q);
    if (tokenized.length()>0){
        q=tokenized;
    }

    //this makes it a phrase and accepts phrases within 2 words
    q = "\"" + q + "\"";
    q = q + "~";
    //setup for Solr, searching
    q = "TITLE:" + q + " OR " + "SECTION_TEXT:" + q  + " OR " + "SECTION_TITLE:" + q;
    query.set("q", q);
    //apply weights to fields
    String qf="TITLE^1.1 SECTION_TEXT^0.3 SECTION_TITLE^0.6 ";
    query.set(qf);

    return solr.query(query);
}
```

Figure 1.2 The runQuery() function found in wiki.java

runQuery() does a weighted field search with query phrasing on top of Solr's BM25 functionality. This ensures that articles are displayed with relevant information in a section but not in the article title. I chose to weigh the title the most, then the section title, and then the section text. I used Solr's boosts feature to do this and applied the following boost values:

"TITLE^1.1 SECTION_TEXT^0.3 SECTION_TITLE^0.6."

I fiddled with these values and based on the results I adjusted the scores until I felt that the weights were roughly appropriate. If I had the ability to use machine learning, I would definitely do that here to figure out the best weights. This function also utilizes phrase querying, which is implemented by putting quotes around the query before sending it to Solr. This ensures that [current president of america] returns political results before returning anything related to electricity currents(see figure 1.3

in appendix). I also applied the concept of incorporating phrase slop by adding the " ~ " at the end of my query. This lets Solr know that I am also interested in situations where other words are between words in my query. For example, if the text says "current active president of america," then this would also count as a hit. The default value for the " ~ " feature is 2.

Another IR feature found in this function is my handling of stopwords. I call the tokenizer function that removes stopwords, and then if the resulting String is of 0 length, I know that the entire query was stopwords. This was brought to my attention during my presentation when I tried to query [the who] and got no results. This bug is now fixed and you can see figure 1.4 for the results for [the who].

The other function I use in wiki.java to query Solr is runQuerySP(). This function is only called if the original runQuery() returns less than 10 results and it returns results that account for spelling errors that are 2 Levenshtein distance away from the query. See figure 1.5 for the implementation of this function. In order to implement spell check I basically just parsed my query and added a " ~ " at the end of each term in the query.

The default edit distance is 2, so that is what I used here. In order to decide what edit distance i should use I looked at the query [bebman sheephard]. I designed this query because it is a phrase and each term is 2 edit distance from the phrase "german shepherd." I compared how wikipedia currently handled this query to how Google handled it and realized that Google used an edit distance of two while wikipedia only allowed for an edit distance of 1. I decided to use the same edit distance found on Google because in my experience people actually used Google to search Wikipedia. See figures 1.6, 1.7, and 1.8 for the results for [bebman sheephard] from Wikipedia, Google, and my search engine.

The main difference between runQuerySP() and runQuery() other than the spell check is that runQuerySP() does not do phrase querying. This seems to be a limitation of Solr querying, because when you put quotes around the query you end up losing the " ~ " at the end of each term and it is considered part of the query which of course returns no results. This is why I split this functionality into a separate function that I only call when no results are returned by runQuery().

WikiFX.java

WikiFX.java is responsible for taking the results from Solr and putting them into the JavaFX GUI. There are two main functions that handle this task, doSolr() and wikiSearch(). The doSolr() function dispatches to my two querying functions: runQuery() and runQuerySP(), and uses them to make a combined result list. See figure 2.1 for the implementation of doSolr().

In order to achieve my goal of presenting multiple articles without incurring multiple HTTP requests I simply gather all of the articles in the top 20, and build them locally and save them into a HashMap. This allows constant time of retrieval when the user generates the article viewer window by clicking on a result button. Also it takes less than a second to build these articles because there are only 20 articles being built at once.

The bulk of the rest of the work happens in wikiSearch(). This function is very long because of how I nested my windows in Action Events tied to buttons or text boxes. See source code for the implementation but this is purely using JavaFX classes to present the UI in the three windows.

One problem that came out of nesting my windows in Action Events is that I was unable to implement pagination without basically completely duplicating my entire wikiSearch() function. I did not do this because it created a lot of opportunities for bugs to come up. If I restructured my GUI I would

definitely be able to do this within JavaFX but I felt like that solution was more verbose and difficult to understand.

FUTURE IMPROVEMENTS

The major improvement I would implement is to include my own ranking function. The top 3 results for most queries I tried seem to be very good but after that most of the results are not particularly relevant. Instead of displaying these irrelevant results I would implement an additional ranking on the results list from Solr to pick the best out of these articles or perhaps to expand my query and based off of the first few results to increase the precision of the remaining results.

The other major improvement I wish I was able to include was more UI features. I really think proposing similar articles, and categories, would make it easier to scan a variety of related articles quickly, which was my goal. I would love to have used React and JavaScript to make a more modern looking UI that included more information than just the article.

APPENDIX

Figure 1.3 Search results for [current president of america] showing the impact of phrase querying

Figure 1.4 Search results for two stopwords: [the who]. First result is the band

```java
/**
 * This function searches the TITLE: field for spellcheck results, with an edit distance of 2(default)
 * @param solr solr Instance
 * @param q query
 * @return the QueryResponse from Solr
 * @throws SolrServerException
 * @throws IOException
 */
public static QueryResponse runQuerySP(HttpSolrClient solr, String q) throws SolrServerException, IOException {
    SolrQuery query = new SolrQuery();
    query.setRows(500);

    //tokenize query
    q = tokenizer(q);
    String[] qArr = q.split( regex: " ");
    String querySP = "";
    for (String s: qArr){
        s = "TITLE:" + s + "~";
        querySP += s;
    }
    q = querySP;

    //setup for Solr, searching
    query.set("q", q);
    //apply weights to fields
    String qf = "TITLE^1.1 SECTION_TEXT^0.3 SECTION_TITLE^0.6 ";
    query.set(qf);

    return solr.query(query);
}
```

Figure 1.5 Implementation for runQuerySP() which supplements my search results using edit distance

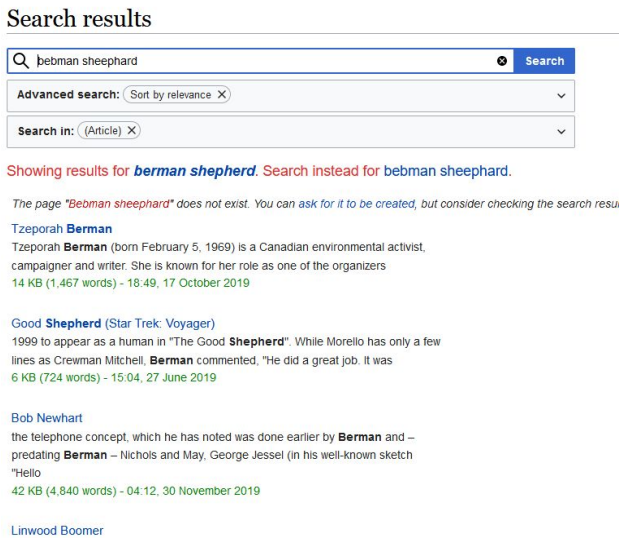

Figure 1.6 Wikipedia results that do not link the query [bebman sheephard] to results about german shepherds

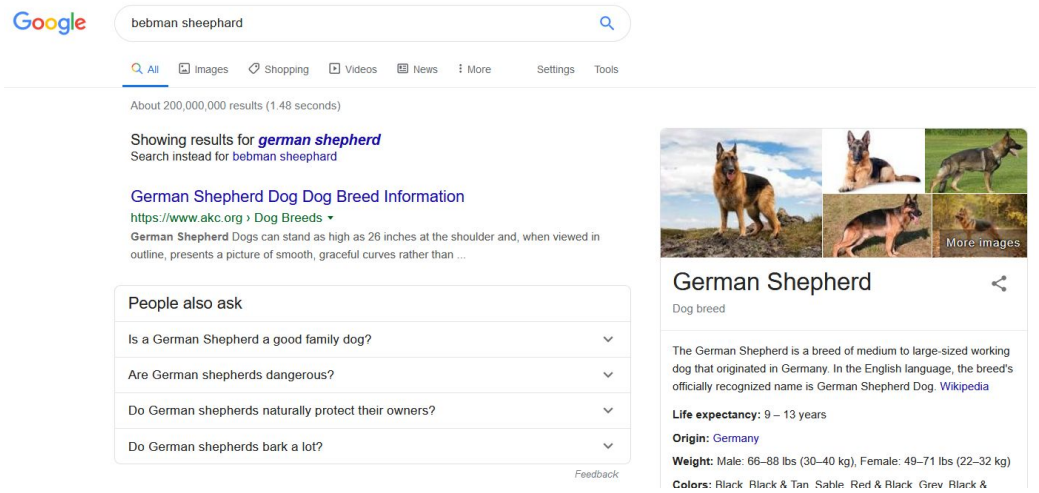

Figure 1.7 Google results for [bebman sheephard] that do link it to german shepherds

Figure 1.8 Wiki-Search results for [bebman sheephard] that align to Google's results

```java
/**
 * This function returns a newline seperated list of results from Solr
 * @param q String query
 * @return String results
 */
public String doSolr(String q){
    HttpSolrClient solr = wiki.startClient();
    String query = q;
    String output = "";
    Object prevTitle = "";
    try {
        QueryResponse response = wiki.runQuery(solr, query);
        SolrDocumentList results = response.getResults();
        //something is misspelled so run spellchecked version
        if (results.getNumFound()<10){
            QueryResponse responseSP = wiki.runQuerySP(solr,query);
            results.addAll(responseSP.getResults());
        }
        for (SolrDocument article: results){
            buildArticle(article);
            Object title = article.getFieldValue( name: "TITLE");
            if (title.toString().equals(prevTitle.toString())){
                continue;
            }
            output = output + "\n" + title;
            prevTitle = title;
        }
    } catch (SolrServerException | IOException ex) {
        ex.printStackTrace();
    }
    return output;
}
```

Figure 2.1 Implementation of doSolr()