Quotations from Johnson's Dictionary

Group 2

Fall 2020

Members:

Sean Datta
Jacob Hofstein
Chris Melton
Brent Pappas
Drew Schilling

Sponsors:

Dr. Amy Giroux
Connie Harper

Table of Contents

1 Executive Summary

It is the duty of students of higher education to acquire knowledge in their field of study while enriching that same field with their accomplishments. When Dr. Amy Giroux approached the University of Central Florida Senior Design I course of Fall 2020 with the *Johnson's Dictionary Online* project, she promised a way for prospective students to hone their software engineering skills while improving not just computer science, but English academia as well: a chance to go above and beyond in our duty in return for a powerful algorithm to parse quotes and source them using collected text repositories.

The *Johnson's Dictionary Online* project involves the digitization of Samuel Johnson's *A Dictionary of the English Language* (hereafter referred to as *Johnson's Dictionary),* an eighteenth-century dictionary publication that uniquely gave quoted examples of defined words to provide context to readers. These quotes were taken from various works of the time, such as from plays or popular editions of the Bible. This quotation usage was by few means structured: quotations are often paraphrased from their source to fit in the dictionary, and some quotes are simply not sourced at all. Paraphrased quotes may even have their source text's title paraphrased. These inconsistencies were of little relevance to readers in the 1800s, but now have become a hurdle in another aspect of the *Johnson's Dictionary Online* project: source attribution. The project is looking to give online users quick access to the most accurate source possible for each quote in *Johnson's Dictionary*. This effort was mostly manual up to Dr. Giroux's proposal, with researchers scouring multiple Internet text corpora (online repositories of written works) for the most true source they could find for each of Johnson's quotes.

Our *Quotations from Johnson's Dictionary Search Engine* aims to translate the painstaking manual labor of sourcing the quotes of *Johnson's Dictionary* into the ease and efficiency of modern-day computing. This document will detail the elements of the project's research, design, and implementation.

This project accomplishes two main goals. First, we design an algorithm that takes the defining quotations of *Johnson's Dictionary* and finds all possible source matches. To find these sources, our algorithm will work with numerous contemporary Internet corpora of varying formatting, such as Project Gutenberg or the Liberty Fund. This corpus data assimilation will be the first of its kind. Once source matches are found, each match will be given a rating to signify its algorithm-evaluated accuracy. Secondly, we will develop a web app tool that will allow users to parse the source matches returned by the search algorithm, manually evaluate quotes with multiple matches to indicate the best match and export the inspected matches in XML format.

The outcome of the *Quotations from Johnson's Dictionary Search Engine* project will be of utmost importance in the greater *Johnson's Dictionary Online* endeavor. When complete, the high quality of both our search results and parsing interface will provide invaluable utility to the work of historians, at UCF or elsewhere, in digitizing history.

2 Project Overview

2.1 Project Description

The goal of this project is to verify the quotations used in *Johnson's Dictionary*, which are approximately 300,000 in number. We need to ensure the actual content of the quote is correct, as well as the associated title and author. To facilitate this comparison, we need access to data stored across the Internet in various corpora. Once we have stored the text data locally in an accessible format, we will compare it with the quotes, generating a list of top five matches for each quote. Once we have these matches stored in the database, the user interface will allow a user to view all the quotes and matches and will also provide a method for users to select and save the most accurate match for each quote. At any point, the user will be able to export saved best matches in a ziped JSON.

Since one corpus will typically contain at least several gigabytes of data, it is not feasible to obtain the data manually; automation is necessary. However, automating a process to extract the data within online corpora is not a trivial task. We will utilize a web scraper to crawl across the corpora and extract any relevant information (the text content, as well as metadata, like URL, author, and title). In addition, we will need to readjust the scraper for each corpus we intend to scrape, since the format and storage of data will differ greatly across the corpora. The text content will be stored in the filesystem of our server as text documents (".txt" file extension), and the metadata will be stored in a relational database.

For the filesystem structure, a corpus will have its own folder. Each author within the corpus will have their own subfolder. Each work written by the author will also have its own subfolder in the author folder. The written works will be stored in text files, and it may be necessary to store each written work as a sequence of text files if a work does not fit in a single text file.

The fuzzy search algorithm will compare the original quotes with the content extracted from the corpora, generating the top five matches for each quote. This piece of software will run as a standalone command-line tool and take many hours to finish processing every quote. There will also be an API that enables the different components of the project to connect to the database and edit database records. Once all the matches are stored in the database, the frontend will allow a user to

select the best match for each quote. The resulting spreadsheet will contain the headword, original quote, title, and author (as they appear in the dictionary), as well as the matched quote, title, author, and URL.

Since the ultimate destination of the results is a database owned by the sponsor, we are providing an export functionality that can convert the Best Matches into JSON format.

2.2 Statements of Motivation

Sean Datta

Seeing this project piqued my interest for a few different reasons. I played Scrabble a lot in elementary school and sporadically onwards and used to also read through dictionaries for fun sometimes when I was a child. I even read some books on Scrabble strategy and learned many interesting words with very bizarre characteristics (i.e., Q-without-U words) and optimal letter combinations for word-building. Obscure words, solving anagrams, definitions, dictionaries, and the like have always intrigued me. Additionally, I have an interest in working with databases, back-end scripting, and writing algorithms and doing data analysis. I foresaw a great potential in this project for such applications of computer science and software engineering areas that I have been learning about in my spare time and would like to gain more experience in.

Jacob Hofstein

Reading and writing have been passions of mine since a very young age. Having the opportunity to contribute to the field of literature was something I was not willing to pass up, especially when it involves working with one of the most influential documents in the history of literature: *Johnson's Dictionary*. Being able to verify the correctness of the most highly regarded English dictionary was very appealing to me, and I am very excited to work on and complete this project. In addition, natural language processing has always been an interest of mine. The concept of designing a computer that can understand the actual meaning of human language is extraordinary, and its potential is limitless. So, when I saw this project, I knew that I would have the chance to learn more about natural language processing and apply it towards a real problem in academics.

Chris Melton

This project stood out to me from the others because I saw it as an opportunity to broaden my skill set while also contributing to a project that was interesting and that I felt I could make a meaningful impact on. Up to this point, my professional experience involves low-level coding for legacy systems that have an immense backend reliance as well as systems software development for microcontrollers. While being able to understand systems software and maintain legacy systems are valuable skills, I felt that contributing to the development of a new algorithm for a system interacting with a database and frontend UI would be a great experience for me. I currently have little database experience, and the experience I do have is only with setting them up as part of a LAMP stack. Learning new things related to my career interest is something that I value greatly and is my main motivator for this project. I am eager to begin working on this project and collaborating with a team of software engineers with similar interests and motivations as me.

Brent Pappas

I was initially drawn to this project because of its adjacency to several of my own interests. I devoted much of this preceding summer to reading several books across an array of genres and subjects. That alone would attract me to a project whose chief focus is on textual data; however, the essence of this project also aligns quite neatly with another recent endeavor of mine. While proceeding through all these books, I often found myself confronted with words whose definitions were unknown to me. To remediate my ignorance, I began compiling a list of these esoteric words in a Google spreadsheet. Over the course of these past few months, I have accreted a list of over 800 such words. Admittedly, this is only a modicum of words when compared to the quantity contained in *Johnson's Dictionary*; but the premise of cataloging words and their definitions is much the same. The discrepancy (and source of entertaining novelty) between my personal compendium and Johnson's is that he also provides a copious number of quotations to exemplify the words he has defined. In retrospect, I feel that I should have done something similar with my own list of words and recorded where I first encountered each of them. To go back and do that now, however, would be an arduous process. The search algorithm we will be devising as a team therefore presents a solution to a problem in my own life. Since I can see the practical application of this project beyond the course first-hand, I am that much more confident that individuals other than our sponsor may accrue some benefit from it

as well. The congregation of these factors compel me to begin working on this project as soon as possible.

Drew Schilling

When I think of how I want to apply my computer science expertise, my mind turns to practical applications that increase the world's wealth of knowledge. With this mindset, I could not deny the appeal of being among the first to document a historical text, especially one that current academics work with. Though historical dictionary work may seem dry, *Johnson's Dictionary* is a unique literary artifact, even today.

The opportunity to design a tool to enable such a unique undertaking is one that cannot be passed up. The frontend of the project is where this important academic work will be completed, and I am humbled to have the chance to add my personal touch to our implementation of it.

I am more than proud to work for the betterment of historical and literary knowledge. I am also eager to learn practical frontend design philosophies to create even more effective tools in the future.

2.3 Goals and Objectives

The chief objective of this project is to provide a tool to facilitate the process of determining the correct attributions of the quotes contained in *Johnson's Dictionary*. *Johnson's Dictionary* defines approximately 40,000 words and employs over 100,000 quotations to exemplify them. These quotations are not verbatim however, and often feature abbreviations and paraphrasing. Further complication arises from the fact that some of the quotes are misattributed or lack attribution altogether (being attributed to "anonymous"). It is also difficult to ascertain the veracity of correctly cited quotes, as it is not uncommon for an attribution to feature solely the author's last name, initials, or even a single letter (e.g., "M").

To rectify the reconditeness of what is supposed to be a source of reference, the project sponsor has tasked assistants with the responsibility of meticulously assessing the accuracy of each quote's attribution. The current method of achieving this goal is that for each quote, one must manually scour the Internet for its source, double-check the result, and finally cross-reference the source with its Library of Congress Control Number (LCCN). Our project will attempt to mitigate

the arduous nature of this task by automating the source and LCCN lookup process, so that assistants only must select the correct source from a list of likely candidates. The simplified process would be as follows: First, users select a quote for the system to search the Internet for; second, the user selects the correct attribution from a list of potential sources found and graded based on accuracy; finally, the selected attribution is recorded in a database and associated with the quote.

The user will interact with the system via the front-end interface of a web-based application. This interface will allow the user to select a quote to search for and display the results of the search back to the user after the search is complete. The results will be graded and sorted based on accuracy. The user may then select what they think is the correct attribution from the presented list to store it in the database and link it with the selected quote. The interface will also provide a means of exporting quote-attribution data to a common spreadsheet file format, such .csv or .xlsx. Export criteria will be offered so that users may avoid overwhelming the system with large sudden export commands, and instead export only portions of the data at a time (e.g., only exports quotes associated with words starting with the letter "J").

The source lookup process (and essentially the crux of the project) will be handled by a fuzzy search algorithm that our team will design and implement. Our team anticipates that this algorithm will require research into the construction of search engines due to the myriad sources available to search from. Furthermore, the algorithm will likely have to interface with pre-existing APIs to obtain raw text data, metadata, and LCCN of source corpora.

The quotes, attributions, and other data relevant to the project will be stored in a relational database. All the quotes in *Johnson's Dictionary* are currently tabulated in a series of alphabetically ordered Excel spreadsheets, and will need to be exported to the database after its creation. The user interface will provide a means of executing standard create, read, update, and delete (CRUD) operations on this database, such as updating a quote's recorded attribution or adding an attribution to a quote that lacks a confirmed source.

Finally, the project will require some sort of user authentication system since it is intended for the use of vetted project assistants only. User credentials will be stored in the project database, and passwords will be encrypted using a one-way password-hashing algorithm such as bcrypt to ensure account security. It is to be

determined whether the user interface will provide a means for creating users, or if this project will require all user accounts to be created manually by the project sponsor.

2.4 Design Criteria and Constraints

There are no mock-ups for the application interface, but the interface must be simple and effective at conveying information. We may decide to utilize some sort of frontend framework or library such as React or Bootstrap to lend greater aesthetic appeal to the interface; however due to the relatively narrow scope of this application we may decide to forego such tools and instead create the frontend using HTML, CSS, and JS.

A web hosting service is provided by the project sponsor; however, a data storage service will need to be determined by the team. The size of the sponsor-provided Excel file containing the quotes is approximately 15MB, if we are to assume that citations will require another 15MB then our database will require at least 30MB of space. The storage requirements could vastly increase, however, depending on the implementation of the search algorithm and whether the team decides to run it for all the quotes in the dictionary and store the candidate attributions in the database before releasing the application to production.

If the team decides not to run the search algorithm on all the quotes in the dictionary and not to store the candidate attributions in the database before releasing the application to production, then the database will require as much space as described above. If the team chooses to take the opposite approach, and to run the search algorithm on all the quotes in the dictionary and to store the candidate attributions in the database before releasing the application to production, then the database will require substantially more space.

This decision has implications for constraints on the runtime of the search algorithm as well. If the former approach is chosen, then the algorithm must be very fast to present results to the users of the application in a timely manner. If the team opts for the latter approach, then the runtime complexity of the algorithm will be much less important since users will only have to query the database to obtain the candidate attributions for a quote.

The team may decide to download all the potential quote sources beforehand to further reduce the runtime of the search algorithm, since it is quicker to search

through local files offline rather than to make a web request to obtain the data for a source each time it is accessed for searching. This has the potential to increase the storage space requirements of the application by tens of gigabytes; the size of all the English text on Project Gutenberg, for instance, is 40GB.

All code must be checked in, committed, and pushed to the project GitHub repository before being deployed to production. The sponsor must be granted full access to the repository by the end of next semester so that they may expand on the project later if they so choose. The GitHub repository must be private.

The project must be completed by the end of the senior design showcase at the end of the 2021 spring semester.

2.5 Broader Impacts

Though the scope of this project is limited, there are definitely a plethora of aspects in the final deliverable that could be used outside of quote searching. All of our data is coming from *Johnson's Dictionary*, but the techniques developed to search through a database of historical text could be applied to other older dictionaries, almanacs, or thesauruses. Additionally, the data export feature of the software could be streamlined and appropriated for use in other programs dealing with large internal data reports.

When completed, this project will be integrated into our sponsor's larger services, and so we can improve software even outside the scope of our own project. Our project is unique in this sense: though most of the work done will be entirely our own framework, our final product must still be applicable to a currently active framework. It is this challenge that will allow our group to bring our own contributions to our sponsor's literary software development.

However, the largest external contribution our project will provide has to do with enriching academia rather than solving a specific problem. As the years go by, more and more physical texts are lost to time, never to be recovered. Thanks to the work of projects like Quotations from *Johnson's Dictionary*, future students and scholars will have access to historical works that would otherwise fade away, making this project have relevance for years. Additionally, the main deliverable of the project will be its search engine, which is a technology that is never truly perfected. With that in mind, the information regarding the research, design, and implementation of our personal search engine could be used in the future by any

of the many talented computer scientists developing for text match searching.

A significant hurdle in our development of the project will be how we deal with extracting a large quantity of literary data in multiple forms from several different text collections, or corpora. All this data must be assimilated together in such a way that our search engine can parse it efficiently in its entirety. *How* exactly this data is collected is not specific to the works of *Johnson's Dictionary*, and the methods we develop could be implemented in other fields of literature and academia. These areas of study by nature deal with lots of historical text and could certainly use our design frameworks.

Besides data parsing, we must also develop text matching algorithms to track down quote sources as well as map out the veracity of Johnson's quotes themselves. It goes without saying that algorithms to match just how paraphrased a text is would be of utmost importance in fields that often have to deal with dubious sources. The text-to-source veracity checks will likely be easily portable and immediately useful to other implementations in academia.

2.6 Legal, Ethical, Privacy Issues

In designing and maintaining this project, there will be a variety of issues we will face that must be taken into consideration. This section will describe how we, as a team, plan to address these issues in our implementation of the solution, as well as how we will ensure that those maintaining this project in the future will not introduce new legal, ethical, or privacy issues. This is a critical aspect of any software development plan and failing to incorporate the described solutions to these issues within the design of our software would be detrimental to our credibility as developers.

2.6.1 Legal Issues

Throughout the development of this project, we will be referencing online resources to aid us in designing and implementing a solution to the problem. While the information in these resources may not necessarily be free use, we will not be copying the information provided by them in any way; these resources will be used strictly for the purpose of research to aid in the development of our own unique solution. A different legal concern involves the use of the data that this project will be centered on. This data comes from A Dictionary of the English Language by Samuel Johnson, which was published in 1755. The publish date is well over 70

years in the past, meaning that the copyright has expired for the publication and it is now considered public domain. This fact allows us to use this data without the need to worry about unintentionally violating copyright laws. Regarding the maintenance of this project after it is completed, as a team we recognize that the software we create will be delivered to the sponsor with the assumption that they will expand upon it. We will not retain any legal rights to the software after delivery, meaning there will be no issues with them modifying the original software. Overall, this project and all material created in support of it will be owned by the University of Central Florida, as it is a project that is being completed as part of a class.

This project will require the use of a web scraper to extract information from online libraries. Some websites specifically say they do not want their sites subject to this in their terms of service, so each of the online libraries' terms of service have been examined to determine whether they allow web scraping. HathiTrust and Liberty Fund do not mention this within their terms of service, so we have determined that it is safe to scrape their sites. The Library of Congress explains that they allow it, but they reserve the right to ban any IP addresses that are creating too much traffic. We will take this into account when developing the scraper for the Library of Congress, and ensure that we do not send too many requests at once. Project Gutenberg specifically states that they would not like their website to be scraped; this fact was another reason we had decided to download a local copy of the site's library to process.

2.6.2 Ethical Issues

The work done for this project will be done with no expectation of monetary compensation of any kind. In a professional setting this would raise many ethical concerns, but because we are doing this work for a college class, the team recognizes that there was no expectation of compensation in the first place and does not expect any to be offered. Another important ethical concern that is prevalent in the technology industry is the tracking of user actions while they are utilizing a web-based software system. The software developed for this project will be web-based, but we will not implement functionality that will track the user in an identifying way. The only tracking we will implement is the tracking of specific actions that the user performs, and this information will be recorded in a way that does not include any identifying information about the user. This information will be recorded to allow us – and future developers – to have a way of knowing what actions were performed leading up to an issue occurring in the system. Doing this makes debugging the issue much easier and leads to the issue being fixed as quickly as possible. In addition, this collected information, any other information

related to the project, and the software itself will not be monetized in any way by the developers. As a team we understand that this project will be used by the project sponsor however they would like, but we will be designing and implementing the solution in a way that does not lend to monetization.

2.6.3 Privacy Issues

An important privacy concern that applies to any web-based product that requires users to hold an account on the system is what user information will be required to create said account. Accounts that are created by those using this software will only require a username and password. No personal information related to the users will be collected, and the account credentials will be stored securely in a database so they cannot be accessed with malicious intent. Additionally, the account credentials will be encrypted and stored securely in the database used by the software. On another note, the code written for this project will need to be kept private from outside developers who may want to access our code with malicious intent. This is done by utilizing a GitHub repository that is set to Private, meaning it is only accessible if a developer is sent an invitation through GitHub to access it.

3 Specifications and Requirements

3.1 Database

3.1.1 Requirements

- Create a MySQL database with a table for each of the following:
    - Data regarding the original 299,588 quotations in *Johnson's Dictionary*.
    - Resulting top five matching quotations and their respective data as entries for each original quotation (should be just shy of 1.5 million entries in this table).
    - List of users and their authentication information.
- Store the MySQL database on the sponsor-provided development server.

3.2 Corpora Scraper

3.2.1 Requirements

- The corpora scraper shall take multiple designated corpora of literature and extract the desired contents from each.

    - Each extracted literature must have been published before or during the publication date of 1755. Corpora that do not provide this information shall have their texts verified after being scraped.

- The scraper shall start at a given corpus URL and scrape its contents from there.

    - The given starting URL shall be a page containing a list of published literature, and on that page, there must be a link to the next page of publications.

    - The starting URL shall have no more than 25 pieces of literature listed on it.

- The scraper shall maintain a Queue of pages to scrape, and only stop running when that Queue is depleted.

- When scraping each corpus, if a page URL fails to load on the first try, that URL shall be put into a secondary Queue (“failure Queue”).

    - The failure Queue shall be processed after the first Queue of pages has been depleted.

- Each piece of literature contained in the corpus shall have its entire written text extracted along with its author, title, URL, and LCCN (“metadata”).

    - The metadata shall be stored in a database.

        - The name of the file containing the full text shall be stored in the same database entry.

    - The written text shall be stored in raw text on the local file system.

        - These files shall maintain a naming structure that adheres to the following pattern: <corpus_prefix><file_index><title_substring>.txt, where <corpus_prefix> is a prefix created from the name of the corpus the text is associated with (i.e. “loc” for Library of

Congress, “hat” for HathiTrust, “gut” for Project Gutenberg, and “lib” for Liberty Fund), <file_index> is a number maintained within the respective scraper that is guaranteed to be unique within that scraper’s set of text files, and <title_substring> is the first five characters of the literature’s title wherein all non-alphanumeric characters have been replaced with “-”.

    - If the literature is not scraped from the Library of Congress, that literature shall have its LCCN attribute set to “-1” in the database.

- Each different corpus shall have its output formatted in the same way to maintain continuity.

3.3 Fuzzy Search Algorithm

3.3.1 Requirements

- Implement a fuzzy search algorithm in any language to identify the top five matching quotes for every quotation in *Johnson’s Dictionary*, and add the results to the appropriate database tables prior to deployment (*not* at runtime).

    - The fuzzy search algorithm should utilize a scoring method based upon some metric for gauging string similarity and this “score” should also be added to the appropriate table in the database in a designated column.

    - The algorithm must also return the metadata (e.g., author name, title of work, etc.) for each matching quote.

- The algorithm must run quickly enough to finish searching for each quote in the *Johnson’s Dictionary* before the project is released to the sponsor at the end of the Spring 2021 semester.

- The algorithm must be implemented as a stand-alone command-line application.

    - This command line application must support parsing several command line arguments for configuring the algorithm's output. This application may also be able to read environment variables to obtain configuration settings which are unlikely to change frequently.

- For quotes which do not change across different editions of the dictionary, the algorithm must only be run for the quote in one edition of the dictionary. This result may be stored in the database and associated with the quote across all editions of the dictionary.

3.4 API for Interaction with the Database

Requirements

- Write a server in Python that includes all the necessary API methods to query the database to retrieve the requested information and return them to the client, particularly the front-end

- The following functionality should be provided by the API:

    - User authentication

    - Retrieving matches for each given quotation

    - Retrieving metadata information for each given work

    - Adding or overwriting matches for any quotation

    - Adding or updating the best match for any quotation

3.5 Frontend of Web Application

Requirements

- Develop a user interface in the front end of the website which allows users to interact with the fuzzy search algorithm.

    - Users need to be able to search by individual words, such as by a work title or headword

    - Obtain the original quotation as well as their top five attributions for a particular search

- Create a login form page which is presented whenever a new user visits the website, which will interact with the users table in the MySQL database and upon successful authentication, redirect to the main page with the user interface to search words

- The five attributions should be ranked by their displayed score, or if functionality exists for users to vote on the best attribution, by the number of votes each attribution has so far received

- The user must be able to export the list of quotes and their user-confirmed attributions to a common spreadsheet file format such as .csv or .xlsx so that the sponsor may programmatically parse the data and add it to their private database.

- The user must be able to set export data criteria to limit the size of the export file. This is to minimize Internet bandwidth and latency requirements.

- The frontend app must be designed as a responsive web page for maximum simplicity and usability.

- The app must enable users to query the results of our quotation source algorithm and display those results in a spreadsheet style.

    - Which results are returned may be specified by the user before the query begins?

- The number of quotes that will need to be displayed should be able to vary from less than ten to multiple thousands, so variable data display will be used.

- Quotation matches must be rated by the algorithm and displayed in the

spreadsheet.

- The web app will also allow users to go through the displayed results.
    - Certain matches for each quotation must be able to be marked as being the “Best Match.” Results marked as “Best Match” will have a flag added to their database entries in case the match is pulled in a future search.
- Once the user is done inspecting the results, they will be able to export the data to an XML file.
    - This file will be formatted in such a way as to assimilate smoothly into our sponsor’s greater *Johnson’s Dictionary* project.
- Implement a system for users to vote on the best attribution and keep track of this information in the database.
    - Whenever a quotation is queried, utilize vote counts for each attribution to rank the five attributions by votes
        - The algorithm score should still be displayed, and the attributions should still be sorted by this method in cases of a tie or no votes

4 Division of Labor

4.1 Sean

- Set up phpMyAdmin on the server and the database to store the data.
- API: Use Python Flask to implement an API to query data from the database and fulfill requests from the front-end.
- Work on front-end and back-end integration, to make sure all requests are being fulfilled as expected and the front-end can utilize the data obtained from these requests properly.

4.2 Jacob

- String metrics: Establish a working method for evaluating and scoring the similarity of strings
- Pseudocode: Write pseudocode to detail the implementation of the algorithm
- Test cases: Write unit tests for each of the pseudocode functions to aid in test-driven development
- Natural language processing: Design methods of sentence segmentation, string tokenization, and approximate string matching

4.3 Chris

- Researched different corpora to extract literature from.
- Corpora scraper: Developed and maintained the Python code that will scrape each corpus and extract the full written text and metadata of each piece of literature.

- Utilized corpora scraper to scrape Project Gutenberg, HathiTrust, the Library of Congress, and the Online Library of Liberty (Liberty Fund).

- Project Gutenberg: Downloaded the contents of the Project Gutenberg corpus and wrote a shell script to extract and organize each HTML file into a central directory.

4.4 Brent

- Project management: Organized weekly team meetings and bi-weekly sponsor meetings, set the agenda for each meeting, and recorded the notes for each meeting.

- Sponsor liaison: Handled nearly all email communications with the sponsor.

- Fuzzy search algorithm: Conducted research into various string similarity metrics, assessed the benefits and drawbacks of certain design decisions regarding the fuzzy search algorithm, and analyzed the time complexity of the final fuzzy search algorithm. Worked with Jacob to design and implement the algorithm.

- Excel to JSON conversion: Wrote the Python script to convert the sponsor-provided Excel file containing all the dictionary's quotes to a JSON file and created Python unit tests for this script. Communicated with the rest of the team to determine what quote metadata needed to be preserved during the conversion process and what could be omitted.

- Dependency management: Conducted research into methods of managing the dependency lists for components of this project that were written in Python and concluded that pipenv would be the best choice.

4.5 Drew

- Research into proper front-end web design workflow, design philosophies, and supporting software tools.

- Research into front-end implementation tools and software frameworks.

- Designed End to End testing protocol.

- Establishment of final front-end requirements with respect to sponsor-requested features and team-designed fuzzy search algorithm.

- Creation of initial and final site page designs utilizing sponsor and team feedback.

- Bootstrap: Conducted research into Bootstrap as our choice framework, as well as its implementation, its grid system, and how to translate established designs into a Bootstrap environment.

- Development of PHP intermediary curl scripts.

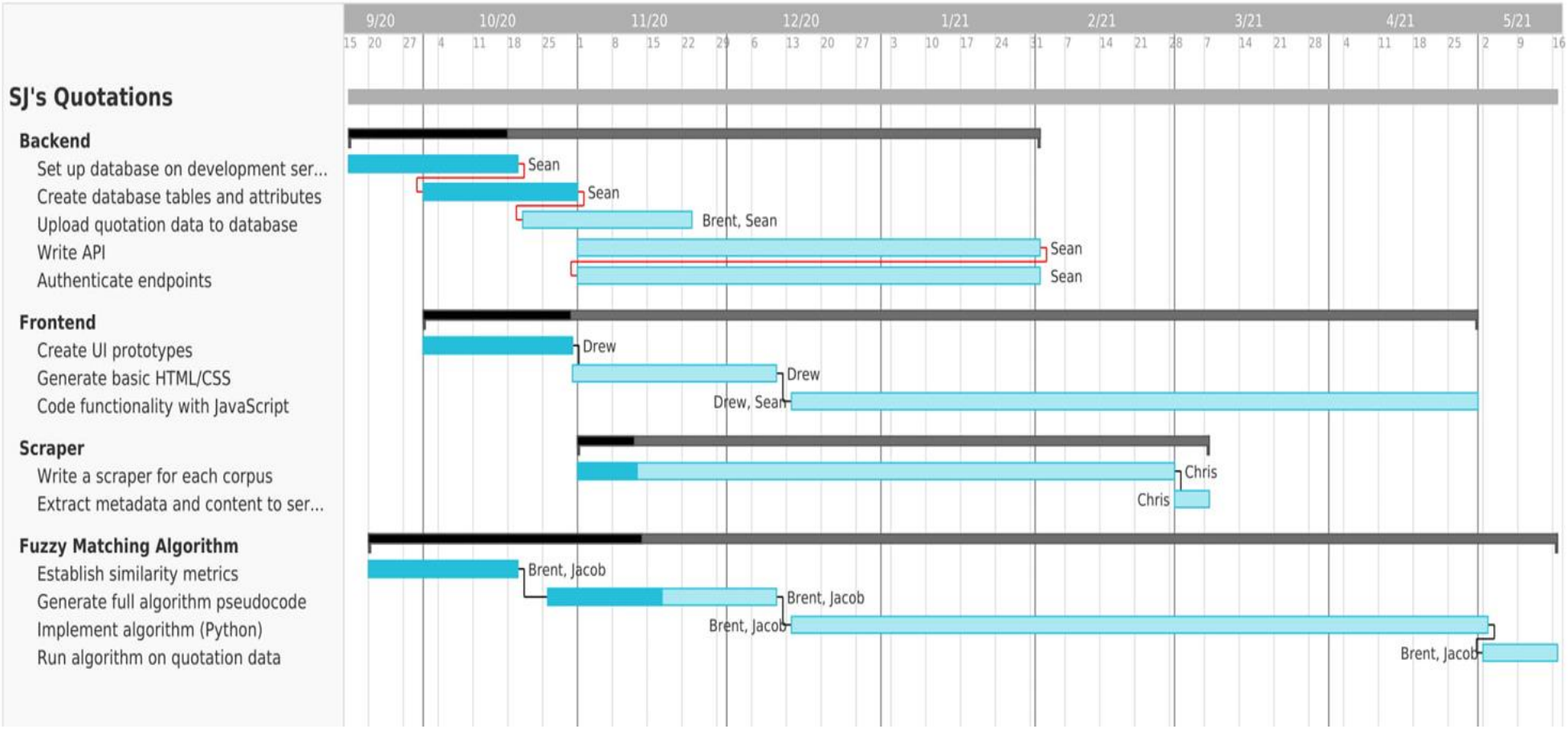


Figure 1: Project Gantt chart, created using TeamGantt.

5 Research

5.1 Front-end Web Application

5.1.1 Framework Language

There are a plethora of languages and frameworks available for designing web applications, but not all were appropriate for the scope of the project. Most client-side web app development in the current market is driven by a language trifecta of HTML, CSS, and JavaScript. These three languages packaged together underneath a general framework allow for the realization of most front-end needs. To avoid a ‘bandwagon’ scenario where using the most popular framework may cause otherwise avoidable hiccups in front-end development, we aimed to stay open to more niche frameworks as well. Purely single-language frameworks (like Vue for JavaScript, or the aptly named Pure for CSS) are also popular and cut the language learning requirement for front-end implementation by two-thirds when compared to other frameworks [1]. Unfortunately, utilization of these frameworks neglects the potential benefits of the languages not used. A JavaScript-only framework may require extensive code to align containers in a table, while a combination framework with CSS can take advantage of baked-in layout tools to easily create a grid. To create a spreadsheet-style most efficiently and responsive front-end, a more comprehensive framework in the style of Bootstrap, Angular, or React is necessary.

5.1.2 Complexity and Maintenance

Several project considerations went into the deliberations over the choice of framework. We had to relate it to the requirements laid out by the sponsor: simplistic frameworks like Pure are convenient for practical development but limiting the scope of our design to fit within those frameworks is undesirable. At the same time, a grand, complex front-end using a professional framework in the style of Foundation is mostly unnecessary for only the few client-side files that are needed for the project [2]. Even if the project stuck to a limited scope, the encompassing nature of these frameworks could drastically affect the speed of implementation.

Comparing frameworks by complexity is also useful when envisioning the code over the lifetime of the project. A good frontend should be limited enough to remain functional through multiple software updates over the project's lifespan and should be understandable enough to be easily maintained by those outside of our project group. Our sponsor's *Johnson's Dictionary Online* project will be continued for some time after completion of the *Search Engine*, and any interface hiccups that appear in that time need to be easily identifiable and fixable in our delivered code. A twisted, complicated mess of "spaghetti" code may get the job done, but it will doubtlessly impact user satisfaction of our final product. With this in mind, it is imperative to select a maintenance-friendly framework like Ulkit or Bootstrap to prevent program bloat before implementation even begins.

5.1.3 Technical Concerns

There are also requirements and characteristics of our project that necessitate deeper research into front-end frameworks' technical applications. Many front-end frameworks available in today's market are designed to be functional as an Internet web site [3]. Having a site be connected to the Internet introduces a dizzying array of potential security and display issues, which many popular frameworks such as React and Angular seek to alleviate. The *Quotations from Johnson's Dictionary Search Engine*'s front-end does not need these web-based tools: though the fuzzy search algorithm *does* interact with Internet corpora through its web scraper, this functionality exists completely outside of the final front-end's environment. Additionally, although all front-end frameworks are concerned with user interface development, different frameworks have different design paradigms on how that data is handled within the framework. A framework like Vue has a rigid, structured data flow that enables the front-end to manipulate received data as smoothly as possible. This is as opposed to a framework like SemanticUI or Bootstrap, which emphasize a consistent user interface with simple design tools over data manipulation [3]. These frameworks are useful when not much data has to be changed client-side, but a lot of data needs to be displayed in an orderly fashion: it is for this reason that we investigated these such frameworks, as the front-end of the project's only modification of data is in the form of the Best Match flag with the lion's share of the information retrieved from the database is read-only.

5.1.4 Bootstrap4

The web app will be primarily constructed with Bootstrap4. Bootstrap is a very popular frontend library that enables development with HTML, CSS, and JavaScript. Bootstrap is the ideal choice for the frontend for several reasons. Bootstrap has a famously simple and powerful data table display capability built-in, which is the primary use of the web app: taking advantage of this feature will greatly improve the simplicity of the implemented code. These Bootstrap tables are fast, easily modifiable, and can be searched, all features that synergize well with the high data volume of our project.

5.1.5 Web Stack Decision

To facilitate the creation of a user interface and a database to store relevant information, we decided to use a web stack as a framework for development. After some research and thinking, we concluded that a LAMP stack was perfect for our purposes.

For the database, a relational structure was by far the best choice. The quotation data we were given to work with suited a SQL database perfectly. We believed that a NoSQL, document-based database design would add needless complexity without offering any additional benefit over the power and simplicity of SQL. For the API, PHP had all the tools to allow us to connect to the database and send information between the frontend and backend. Since there is very little backend processing happening on the server besides the API, the PHP scripting language seemed like the perfect choice. To run PHP, we went with the Apache web server software.

The user interface has a simplistic design, and its only purpose is to display the information generated by the algorithm. What this meant for us is that a frontend framework would not be necessary since the actual functionality of the UI would be simple in nature. There was no intricate or complicated animations or actions that would need to rely on Angular or React. For our purposes, vanilla JavaScript, along with HTML and CSS, was the way to go. In addition, since the algorithm and the web scraper do not have to run on the server that will host the UI and database, they did not factor into our web stack decision. However, the server can easily run Python if necessary.

The final and greatest consideration for our LAMP stack decision was that the development server provided by the sponsor was already capable of running a LAMP stack without any additional installations or configuration. The ease and convenience of this is what ultimately led to the decision of a LAMP stack.

5.1.6 Front-end API Calls

There are a number of powerful JavaScript libraries that can be used by the *Search Engine*'s frontend to make calls to the API. Though basic functionality could easily be tested with any library, there are important distinctions between the libraries that are important to highlight to make the best choice for front-end development. This research delves into the strengths and weaknesses of these libraries as they relate to our project. Function of the chosen library will be supplemented by vigorous API testing as implementation progresses.

XMLHttpRequest

XMLHttpRequest is an older HTTP request library in JavaScript. It takes the form of a request object that specifies the request method (POST, DELETE, etc.) and the API endpoint to make a request. This library is rather simplistic, but it is still used today, both on its own for older browsers and modified as the groundwork for the other libraries discussed here [4]. Though XMLHttpRequest is tried and true, it is rather bulky to implement. Additionally, our project's front-end does not need to account for older browsers, so this library should be kept as a backup in case of buggy interactions with modern libraries.

Fetch

Fetch is the library that was built to replace XMLHttpRequest. In this sense, fetch could be considered its lightweight replacement. Basic fetch requests are facilitated by a global `fetch` method instead of an instantiated object, and the code required to implement is minimal. Fetch is not without its issues: `fetch` requests will not reject on HTTP error codes, and its status as a method makes debugging difficult [4]. Though Fetch would work for the project frontend, its code would be less intuitive to understand as compared to other libraries.

Axios

Axios is an open-source library more modern than XMLHttpRequest or Fetch. It is a promise-based client that can be implemented in JavaScript alone or as part of one of the more advanced frameworks researched in this document, such as React, Angular, Bootstrap, or its designed framework Node.js [4]. This means that we can incorporate Axios into the framework with an external CDN, making the final programmed frontend more concise, readable, and uniform. Axios is widely supported and is a modern library with better error handling than Fetch with more automated data manipulation to boot.

jQuery

jQuery is a library that simplifies event handling while implementing Ajax techniques. It is the most popular JavaScript library in the world [4]. Popularity aside, jQuery handles API calls from client-side users with a respect to JSON not found in other JavaScript libraries, even having a `getJSON` method to organize retrieved data for immediate parsing by the front-end [5]. jQuery also has the most error support and network event handling out of our prospective libraries. jQuery is readily supported by modern browsers, which is good for the front-end, and like Axios can be seamlessly integrated into the other frameworks that will comprise the bulk of the front-end. With these boons in mind, jQuery will likely be the best fit for our API calls from the front-end.

5.2 Algorithm

5.2.1 Methods of and Metrics for Measuring String Similarity

Many algorithms and metrics exist for measuring string similarity, and each is tailored to a specific task. Before designing our fuzzy search algorithm, it is essential that we research all the methods available for gauging the likeness of two strings so that we may choose the one best-suited to our project's requirements. What follows is a listing of the algorithms we have discovered during our research thus far, and our assessments on whether they would be suited to our project.

Levenshtein

Perhaps the most popular method of measuring string similarity is by computing the edit distance between two strings. The edit distance between two strings is the number of operations required to transform one string into another. There are different types of edit distances, and each type may include a different set of operations or weigh the operations differently. The most popular form of edit distance is the Levenshtein distance. This metric allows for character insertion, deletion, and substitution operations, and weighs them all equally [6]. An outline of an algorithm for computing the Levenshtein Distance between two strings in shown in figure 2.

In respect to other metrics for measuring string similarity, the Levenshtein distance is stringent in its comparisons between strings. For instance, metrics that rely on set comparison may not take character or word order into account, whereas the Levenshtein distance does. This strictness can be burdensome in some cases but can prove quite useful when leveraged properly. For our project, we decided that it may not be the best decision to find first-round matches for an entered quote using the Levenshtein distance since the original quote may contain additional characters or words or may be a very different string altogether if the entered quote is a paraphrase of the original. If we were to use the Levenshtein distance during our algorithm's first-round of searching, which is intended to find strings in the original text that are generally similar to the entered quote, then we run the risk of the algorithm immediately removing strings with such discrepancies from the list of potential matches.

With this in mind, we decided it may be in our best interest to instead employ the Levenshtein distance in the algorithm's second round of searching, after the pool of candidate matches has been narrowed down slightly to contain only those strings which generally match the entered quote. Here, the Levenshtein distance could be used to assess the similarity of candidate matches to the entered quote in more granular detail. Furthermore, each of these edit distances could be normalized by first subtracting it from the length of the larger of the two strings being compared in each case and dividing the difference by the same length [6]. This would provide an easy means of ranking the accuracy of the matches found.

Finally, since the time complexity of computing the Levenshtein distance is $O(n \times m)$, where n and m are the respective lengths of the two strings being

compared, this additional computation would not negatively impact the runtime of the algorithm as its runtime is O(max(*n*, *m*)), which is guaranteed to be at most O(*n*×*m*) when *n* and *m* are positive integers (which they will always be, since strings of length 0 are not being compared).

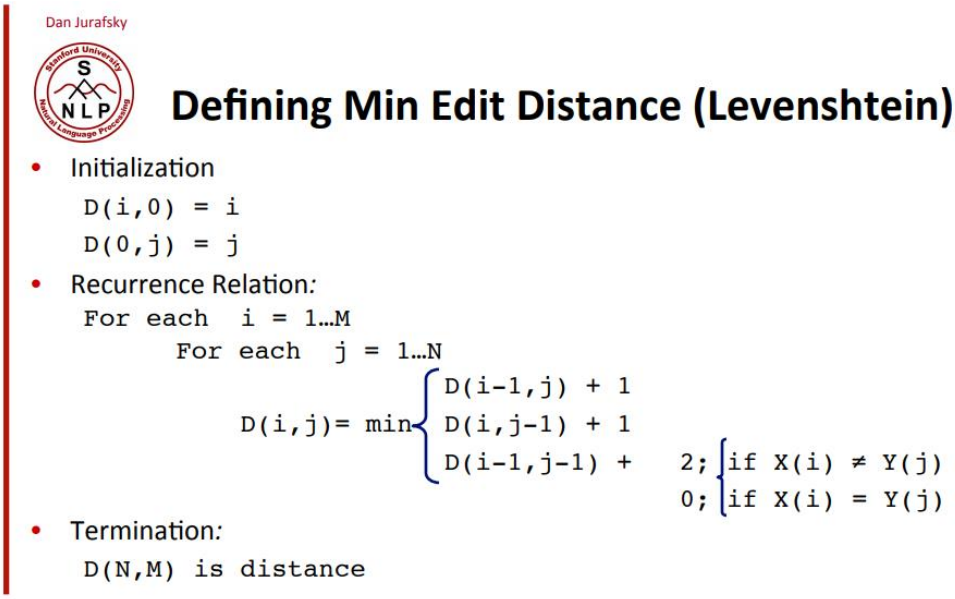


Figure 2: A succinct formal definition of the Levenshtein Distance. X and Y are the input strings, and D is a 2-dimensional matrix used for computing the edit distance between them [7].

Although an algorithm for computing the Levenshtein distance can be implemented in a few lines of code utilizing dynamic programming, creating our own implementation is not necessary since Python packages containing an optimized implementation already exist and are available on the Python Package Index. Two such packages are the python-Levenshtein [8] and rapidfuzz [66] packages, which both offer a fast implementation of the Levenshtein distance algorithm.

Modification: Damerau-Levenshtein Distance

There exists a popular modification to the Levenshtein distance, known as the Damerau-Levenshtein distance. This version of the metric adds the transposition (the interchanging of character positions [9]) of adjacent characters to the Levenshtein distance's list of valid operations [6]. For instance, the Levenshtein distance between the strings "search" and "saerch" is 2, but the Damerau-Levenshtein distance between them is 1. Python packages offering fast methods of computing this measure are also available (e.g., the pyxDamerauLevenshtein

[10]). The impact of this variation of the Levenshtein distance on search result accuracy needs to be evaluated before we can consider implementing it into our algorithm.

Jaro Similarity / Distance

The Jaro distance is another string similarity metric and is unique in that the only operation allowed is character transposition [11]. Formally, it can be defined as follows:

$$Sim_J(s_1, s_2) = \begin{cases} 0 & \text{if } m = 0, \\ \frac{1}{3}\left(\frac{m}{|s_1|} + \frac{m}{|s_2|} + \frac{m-t}{m}\right) & \text{otherwise.} \end{cases}$$

Formula 1: For computing the Jaro Similarity between two strings [11].

where:

- *s1* is the first string being compared
- *s2* is the second string being compared
- *m* is the number of matching characters between the two strings
- t is the number of minimum number of transpositions required to transform one of the strings into the other, divided by 2

Since the Jaro distance returns a normalized value and not the number of edits necessary to transform one string into another, it is not a true edit distance and can more accurately be described as a metric for measuring the similarity between two strings.

Given that the Jaro measure returns a normalized value, it enables an easy ranking of search results, and is thus an appealing option for our fuzzy search algorithm's second, more precise round of string comparison. If we ultimately decide to implement the Jaro distance into the fuzzy search algorithm, there exist a number of Python packages that offer fast functions for its computation. The jellyfish

package is one of the more notable of these packages as it also provides functions to compute other metrics of string similarity [12], some of which will be described later in this paper.

Jaro-Winkler Distance

As with other string similarity measures, a modification exists for the Jaro distance. This modification is known as the Jaro-Winkler Distance, and uses a scale factor to give a bonus ranking to strings that share common prefixes up to a certain length [13]. Though interesting, the applicability of this modification to our project is dubious, as there is no guarantee that the entered quote will share a common prefix with the original quote. If we decide to test this metric, the jellyfish library provides a function for its computation as well.

N-grams

An n-gram is essentially just a contiguous sequence of n elements from another, often larger, sequence of elements [14]. Though not inherently related to string matching, n-grams have many applications for string-matching algorithms. For instance, one could use an algorithm to generate all the possible n-grams for each of two strings, and then compare the number of n-grams that these two strings have in common to determine the similarity of the two strings. Figure 3 contains the pseudocode for an algorithm to generate all the n-grams of a string:

```
function ngrams(s: string, n: int):
    grams = new empty list
    m = length(s)
    for i = 0 → (m - (n - 1)):
        gram = s.substring(i, i+n)
        grams.add(gram)
    return grams
```

Figure 3: Pseudocode for an algorithm to generate all the n-grams of a string.

The time complexity of this algorithm is O($n \times m$), where n is the length of n-grams to create, and m is the length of the string from which the grams should come from. The size complexity is also O($n \times m$).

There are a few ways to compare two strings once their respective lists of n-grams have been generated. A simple way would be to count the number of elements that these two lists have in common; however, this result is of little use to use since it is not normalized. A slightly more sophisticated method that does return a normalized result would be to convert the two lists into sets and then compute the Jaccard index between them. The Jaccard index, and an example of its broader applications in string matching, is elaborated on below.

N-grams may also be utilized to enhance pre-existing metrics for measuring string similarity, such as that of the longest common sequence (LCS). Normally, the LCS approach quantifies the similarity of two strings by returning the length of their LCS of characters. The drawback to this is that it lacks context, i.e., two words that may be quite dissimilar may also happen to be spelled in a way such that the LCS between them is quite high. For example, the LCS of the words "natural" and "contrary", is "ntra", resulting in a relatively high similarity value of 4. This issue can be overcome by instead finding the LCS of the two strings' respective lists of n-grams. Using this approach, "nat-atu-tur-ura-ral" would be compared to "con-ont-ntr-tra-rar-ary" (assuming n=3), and the LCS of the two lists would become the empty string (yielding a match value of zero). This result can easily be normalized by dividing the LCS length by the length of the larger string (or list when using n-grams) [15]. The only remaining downside to using n-grams is the impact it would have on the time- and space-complexity of our algorithm. Given that we have decided to preprocess the algorithm's results and have ample storage space at our disposal, however, n-grams may still be a viable option.

If we were to use this approach, then we would most likely forego the use of pre-existing packages and instead implement these algorithms ourselves. This is because the code to generate the n-grams of a given string is relatively straightforward (it can be condensed into a single line in Python using list comprehensions), and because the algorithm to determine the length of the LCS between two lists is available online and not very difficult to write in Python.

The Jaccard index

The Jaccard index, also known as the Jaccard measure, is a function that operates on two sets and is formally defined as follows:

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|} = \frac{|A \cap B|}{|A| + |B| - |A \cap B|}.$$

Formula 2: For obtaining the Jaccard Index between two sets [16].

It used to measure the similarity between any two sets and can be applied to find the similarity between any two strings as well. For instance, one could convert two strings each into a set of n-grams, then compute the Jaccard index of these two sets to obtain their similarity. Furthermore, since the cardinality of the intersection of two sets cannot exceed the cardinality of the union of two sets, the function is guaranteed to return a normalized result. This fact makes the Jaccard index an appealing metric for our fuzzy search algorithm, as the inherently normalized nature of its results allows us to easily grade them. One potential pitfall of this measure, however, is that since its operands are sets it does not account for duplicates nor does it account for element order. For instance, one of our initial algorithm designs employs the Jaccard measure to compare the overall similarity of two sentences. First, the sentences would be converted into sets consisting of the words in the sentences. Then, the Jaccard measure between these two sets would be computed. Using such an algorithm, however, the two sentences "to be or not to be" and "be not be not to or" would be considered identical. The impact of this shortcoming of the Jaccard index on the accuracy of search results needs to be measured before it can be decided if the Jaccard index should be incorporated into the final fuzzy search algorithm.

Given that relative simplicity of the Jaccard index and the fact that sets and set operations are included in Python's list of built-in types and functions [17], we may decide to implement a function to calculate the Jaccard index ourselves without the assistance of an external Python package.

Prefix Matching with the Trie Data Structure

The trie, or prefix tree, data structure is one specially tailored for working with strings. It is a tree data structure in which each node contains one child for each letter of the alphabet being operated upon. The root node represents the empty string, and each of its non-null children represents the first letter of a string stored in the trie. Each of the non-null children of the root node's children represents the second letter, and so on. There are a few different methods of representing complete strings; one common method is to add a flag to each node which is to be

set to 1 if that node corresponds to the last letter of a string stored in the trie; 0 otherwise. Another method is to only store prefixes common to more than one string in the trie and add a suffix string to each node at a point of divergence. The nodes themselves do not contain the data about which character they refer to; rather this information is inferred for each node via its index in its parent's list of child nodes. Below are diagrams of the two described methods for storing strings in tries. The * represents the root node in the left diagram.

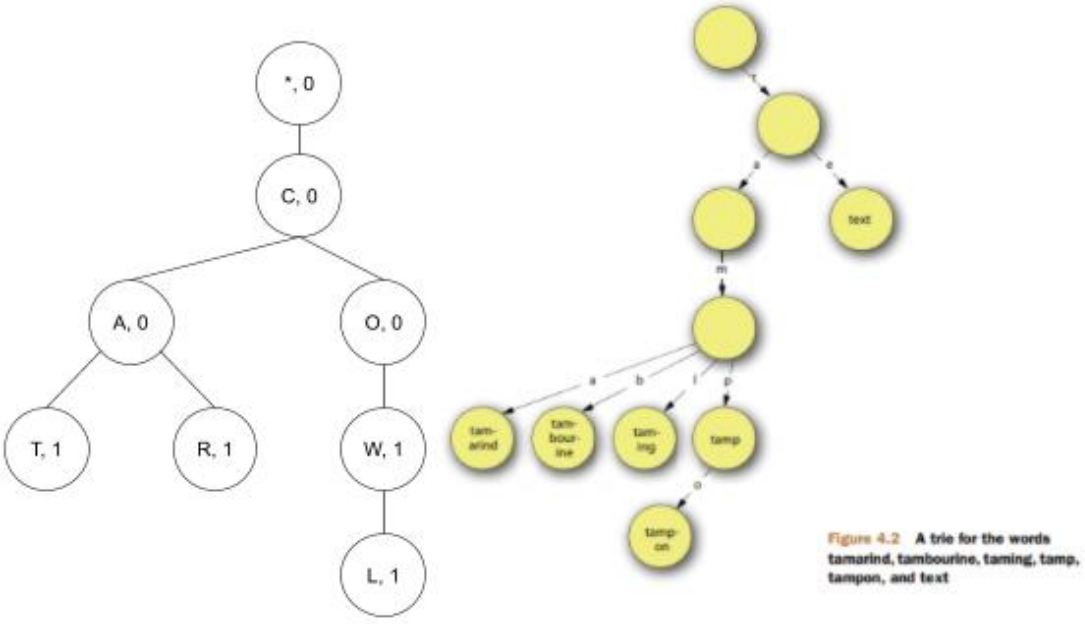


Figure 4: Flag method (left) and suffix method (right) of storing strings in tries. Right is taken from Taming Text [6].

Tries are most suited for prefix matching. To search for a range of strings with a common prefix, one can query the trie for that prefix. For instance, one could query the trie in the above-left diagram for all the range of strings with the common prefix "ca" to obtain the strings "cat" and "car". This sort of query could be modified to return only a subsection of the strings found with a given prefix, e.g., in the above-right diagram, one could search for all strings that start with "tam" followed by either the "b" or "l" to obtain the strings "tambourine" and "taming".

Though sophisticated and fast, prefix-matching by way of tries may not be the best candidate for our fuzzy-searching algorithm. Prefix-matching is an effective means of implementing predictive functionalities such as autocomplete, since the string

that the user is typing can be passed as the prefix to search for. Our project, however, does not involve searching for dynamically entered content; rather users will be selecting a static quote from a static list of quotes to search for over a static list of corpora. Furthermore, the quotes themselves could contain multiple typos at any position within the string, making it much more difficult to search for them via prefix-matching. Finally, some of the quotes may actually be paraphrases of original quotes and begin with different words altogether; in these cases, prefix-matching would be totally ineffective. If we do decide to incorporate tries into our search algorithm despite these apparent drawbacks, we would first try to incorporate a pre-existing Python implementation of the data structure into our project before investing time into creating one ourselves.

Soundex

Unlike the other algorithms listed in this paper, the Soundex algorithm is unique in that it is a phonetic comparison, i.e., it is a measurement of the similarity of the pronunciation of two strings. The United States uses the algorithm to analyze census data and search for identical surnames that may have been spelled incorrectly due to sounding similar. Its current rule set is upheld by the National Archives and Records Administration (NARA). The basic premise is that every word is assigned a code based on its first letter and remaining consonants (except for h and w) [18]. For instance, according to the NARA rules, the name "Johnson" would be encoded as "J525". Once the Soundex codes of two strings have been determined, their similarity can be quantified by counting how many characters their codes have in the same positions. For example, if one were to compare the string "Johnson", with a Soundex code of "J525", to the string "James", with a Soundex code of "J520", the result would be 3.

The Soundex algorithm is a novel and extremely specialized tool, but this prohibits its usefulness for our project. The algorithm has been designed to facilitate comparison between proper nouns, and not between strings that contain multiple words. Since our quotes contain multiple words, one conceivable way to search for quotes using the Soundex algorithm would be to convert the string being searched and the string it is currently being compared against each into a list of words, and then compare the Soundex code of each word in the first list with the word at the matching position in the second list using the method described in the preceding paragraph. The results could then be averaged to obtain the overall similarity between the two strings. Finally, the average could be divided by 4 to normalize it (since 4 is the highest possible Soundex comparison value between

two strings using the algorithm detailed above). This method is too complex and error-prone to be practical however, as it fails to account for strings of different lengths or strings in which words have been transposed. This approach may be naive but illustrates the shortcomings of using such an algorithm to search for strings that it was not intended for (in this case, strings containing more than one word).

This all being said, the Soundex algorithm remains a unique approach to string-matching, and we may eventually find some practical application for it in our project. If we do, we could then use the jellyfish library to add Soundex searching functionality to our algorithm since it provides a function that returns the Soundex encoding of a string.

5.2.2 Ranking the Results

One of our requirements of the project is that our fuzzy search algorithm not only returns the top five search results, but also that these results be graded based on accuracy. This extra layer of complexity may deter us from factoring it into initial algorithm designs. This exclusion is further compelled by suggestions provided by Stanford researcher Anna Patterson in a 2004 ACM article on the challenges of writing a search engine. In the article, Patterson recommends that groups creating their own search engines do not incorporate a result ranking system into their initial designs [19]. Further research, however, reveals that measures of string similarity can often either be easily normalized (such as in the case of the Levenshtein distance), or are already normalized (as in the case of the Jaccard measure). This greatly facilitates the incorporation of a ranking system into our algorithm, precipitating its inclusion into design diagrams.

5.2.3 Adding Support for Synonyms

Quite possibly the biggest hurdle that our algorithm will have to overcome will be that of matching quotes that are paraphrases of original quotations. The difficulty lies in the fact that string-matching algorithms match strings based on their textual similarity, and paraphrases may not be textually similar to original quotes at all. One way to rectify this discrepancy would be to replace some of the words in the quote being searched for with synonyms before conducting the fuzzy search. Researchers have employed such tactics to solve fuzzy search problems before by using synonyms stored beforehand to refine search results and have been met

with success [20]. Instead of conducting one search per quote, a search could be conducted for the original quote plus all its variations containing synonyms. This pre-search replacement could be achieved by using an object of Python's built-in dictionary type [17] to store a mapping of common words and abbreviations to their synonyms and unabbreviated forms.

The drawback to this approach is that it could have drastic effects on the algorithm's runtime. Searching for all possible variations of a string using synonyms would make the algorithm's runtime at least exponential, which may still be unfeasible even if we were to process quotes beforehand. Furthermore, even though prior researchers were able to successfully employ this tactic to solve their problem, their paper suggests that they were searching for strings much shorter in length and containing far fewer potential synonyms than our quotes. We will need to determine whether synonym matching is genuinely needed to find paraphrased quotes, and if it proves to be so then further deliberation would be required to determine how feasible it would be (or how to make it feasible) to include it in our search algorithm.

5.2.4 Time Complexity

The time complexity of the fuzzy search algorithm we created is rather easy to compute, albeit slightly tedious. Let the length in characters of the quote to search for be n_c, the length in words of the quote to search for be n_w, the length in characters of the longest sentence in the input file to be p_c, the length in words of the longest sentence in the input file to be p_w, the length in characters of the input file to be q, and the number of sentences in the input file to be r.

First, the algorithm must convert the given quote to a list of words, and then convert that list into a set. To convert a string to a list of words each character in the string must be iterated over, incurring a runtime of $O(n_c)$; to then convert that list into a set each word must in turn be iterated over, resulting in a $O(n_w)$ runtime. Both operations are only performed once. Next, the program must convert the input file into a list of sentences by splitting each sentence by the designated delimiter character. This necessitates a runtime of $O(q)$ since each character must be iterated over to find all delimiter characters in the input file. Each sentence must then be converted to a set of words, requiring a runtime of $O(p_c) + O(p_w)$ per sentence since each sentence string must first be converted into a list of words before being converted to a set. Next, the Jaccard measure between the given

quote and each set of words must be computed. Due to the fact that Python stores the length of iterable objects as an attribute and does not compute it at runtime, the time complexity of this operation is simply O(r) (as opposed to O($n_w \times p_w \times r$) [21]. Comparison of each Jaccard index is simply a matter of arithmetic and can be done in constant time.

The computation of the Levenshtein distance between the original quote and a given sentence is the most expensive operation in the algorithm, with a time complexity of O($n_c \times p_c$). In the worst case, the Levenshtein distance would be computed between the entered quote and every sentence in the input file, for a total of r comparisons. This distance is then mapped to the sentence in a Python dictionary; in the average case Python dictionary insertions can be done in constant time, but in the worst case they have a time complexity directly proportional to the number of items in the dictionary [22]. For now, we will assume the worst case here as well, i.e., that each sentence in the input file is mapped to an edit distance and that the dictionary's hashing function will always result in a collision. This results in a time complexity of O(r^2). Once the loop is over, we can obtain the dictionary entries mapped to the top five Levenshtein distances in O(r) time in the worst case. Finally, since at most five insertions into the database will be made each time the algorithm is run, its time complexity is O(c), where c is the constant 5.

The sum of these complexities is:

$$O(n_c + n_w + q + p_c + p_w + r + n_c \times p_c \times r + r^2 + r + c)$$

This is the time complexity to search a single file for a given quote.

We can simplify this by making a few observations. First, $n_c \geq n_w$ and $p_c \geq p_w$, since the number of words in a string logically cannot exceed the number of characters in that same string. Next, q can be omitted in the final complexity, since $n_c \times p_c \times r$ is ≥ q ($p_c \times r$ alone is greater than q, since the input file is comprised of r sentences which may at most have a length of p_c). The linear and constant complexities can then be omitted since they are less than the dominant factor. These refinements yield a time complexity of:

$$O(n_c \times p_c \times r + r^2)$$

Recall that in determining this complexity, we assumed that each dictionary insertion resulted in collisions. If we are to assume the average O(1) case, then the complexity simply becomes:

$$O(n_c \times p_c \times r)$$

For each quote in each edition of *Johnson's Dictionary*, this algorithm will be executed over each file of each corpus at our disposal. Let *J* be the total number of quotes across all versions of *Johnson's Dictionary*, n_{cMax} be the length in characters of the longest quote amongst these quotes, p_{cMax} be the length in characters of the longest sentence amongst all our input files, r_{max} be the number of sentences in the input file amongst all our input files containing the greatest number of sentences, and *A* be the number of input files at our disposal. Therefore, the time complexity of preprocessing all quotes using the algorithm we have formulated is:

$$O(J \times A \times n_{cMax} \times p_{cMax} \times r_{max})$$

5.2.5 Dealing with Periods

In an earlier section, we explained that our approach for tokenizing the text files was to chunk them into sentences. To do so, we tokenize using sentence-ending punctuation as our delimiters (period, question mark, exclamation point). When it comes to question marks and exclamation points, we can be certain that any usage of these will end a sentence. However, periods can be much trickier, because we must deal with abbreviations, ellipses, and other uses that do not end a sentence. Obviously, if a period is used as an abbreviation or for something other than ending a sentence, we do not want to consider it as sentence-ending punctuation, but rather ignore it and proceed normally. The question is, how do we implement this? How can our algorithm distinguish between a period used in an abbreviation and a period used to end a sentence?

This is a significant problem in natural language processing, and we can borrow on the work that has already been done on the subject. There are two primary approaches to this problem. The first is considered the standard or “vanilla” approach and involves rule-based detection. The rules are as follows [6]:

(a) If it is a period, it ends a sentence.

(b) If the preceding token is an abbreviation, then it does not end a sentence.

(c) If the next token is capitalized, then it ends a sentence.

This ruleset requires that a predefined list of abbreviations be available with which the algorithm can compare tokens. The ruleset starts with (a), and if rule (b) is satisfied, it moves to (b), and if rule (c) is then satisfied, it moves to (c). It must go in order, and ends when the next rule cannot be satisfied, or there is no rule remaining.

The rules for ellipses are very similar:

(a) If it is a period, it ends a sentence.

(b) If the following token, previous token, or both are periods, then it does not end a sentence.

(c) If the next token is capitalized, then it ends a sentence.

The second method of sentence segmentation is to use some sort of neural network that has been trained to recognize patterns where periods do and do not end sentences. Two examples of this are Apache OpenNLP and SATZ. In both cases, a foundational framework is provided that can be trained on examples to become more accurate. For the purposes of this project, this option is too involved and time-consuming, so we will stick to the first option.

It is also worth mentioning that abbreviations are probably not used as much in the 1700s as they are today. Ellipses are probably way more common. Also, the abbreviations they may have used might not be used at all today, which means we have no way of putting them into a table for reference. Furthermore, it was also common for other punctuation marks (not just periods) to be used for denoting abbreviations, so it might not be much of a concern.

However, a system should still be in place to detect the cases where periods are used in a way that does not end a sentence, so that we can provide the most accurate results possible. It should be known that there are situations where the

rule-based implementation will fail. For example, shortened names such as “J. K. Rowling” that have whitespace between the letters will not be caught as abbreviations, even though they are. All in all, this method will likely be at least 95% effective, and since we are not expecting to encounter many abbreviations (and those we do see might not use periods) from works published in the 1700s, this is more than enough for our purposes.

5.3 Corpora Scraper

Since the quotes in *Johnson’s Dictionary* are based on published literary works, this project will require a way to find online transcriptions of works that were published at or before the time that the dictionary was published and then retrieve the contents of those pieces of literature. We will be using a variety of online corpora to find such works, and the corpora scraper will be designed in a way that efficiently retrieves the full text of every work within the correct time period as well as metadata related to that text. This section will go over the preliminary research describing the corpora we will be using as well as aspects of the design and implementation process.

5.3.1 Implementation Language and Libraries

To preserve continuity with the search algorithm implementation, the corpora scraper will be written in Python. Additionally, we will be using preexisting Python libraries that will give a good starting point for developing the corpora scraper, namely the *requests* and *BeautifulSoup* libraries.

The *requests* library simply adds functionality for performing HTTP requests to web pages to retrieve their HTML data. The responses from HTTP requests sent using this library can be captured in different ways, but for this project we will be utilizing the default capture that happens when the GET request is sent, as this captures the raw HTML of the web page that is passed in [23]. This HTML will then be passed into a call to the *BeautifulSoup* library to do the scraping of each page of text.

The *BeautifulSoup* library adds extensive functionality for web scraping. With this library, we will be able to easily extract the desired contents from each corpus we will need to scrape. Using this preexisting library will eliminate the need for us to

develop our own method for scraping each corpus as it provides methods to parse raw HTML into components like metadata and literature text.

Now that we have libraries for performing HTTP requests and scraping web pages, we need a library that will parse the information that the scraper gets. There are multiple libraries available that perform this task, but for this project we will decide between Python's built-in html parser, the *lxml* parser library, or the *html5lib* parser library.

The built-in html parser in Python provides basic capabilities when parsing through HTML documents. This means that it will not try to turn the received text into a fully-functioning HTML page, but it will try to fix tags that are seen as invalid. In contrast, both the *lxml* and *html5lib* parsers will make an attempt to fix broken tags or simply remove them, as well as converting the received text to a working HTML document. While converting the text into a valid HTML document is not needed for this project, the built-in parser will not be used because it is not as fast as the other two parsers. Similarly, the *html5lib* parses HTML in a way that runs significantly slower than the *lxml* parser does, and it parses pages in a way that will result in complete HTML5 code, which is unnecessary for this project. In addition, the *lxml* parser is recommended by the developers of *BeautifulSoup* due to its speed and parsing capabilities [24]. Due to the capabilities of *lxml* and its speed, we will be using it as the parser for the corpora scraper in this project.

5.3.2 Functionality

Before developing the corpora scraper, we must first understand the ins and outs of how a typical web scraper works. According to School of Data, a typical web scraper consists of three key aspects:

1. A queue of pages to scrape

2. An area for structured data to be stored, such as a database

3. A downloader and parser that adds URLs to the queue and/or structured information to the database [25]

Once these three aspects are established, the web scraper can begin going through the page queue, extracting the information that the developer is looking

for. This information is then formatted in a way that can be sent directly to the database for the developer to either manually go through or pass into another program for analysis.

The corpora scraper will be adapting this model into one that will fit the needs of this project. The queue of pages to scrape will be created from the pages contained in the corpus passed into the scraper. These pages will contain metadata about the piece of literature as well as the full written text. The metadata will be stored in the database, but due to the potential size of a full page of written text, the extracted text will be stored on the local file system.

5.3.3 Corpora

When deciding which corpora to use for this project, there are a few criteria we considered that were determined to be solid deciders for the usability of a corpus. First, we considered the amount of material contained in each corpus was considered. A corpus with less than 100 pieces of literature was not considered to be beneficial to this project, as there are other corpora with thousands of pieces of media that can be used which would likely contain the same information as the smaller corpus. Then, we considered how easy it would be to scrape a given corpus. The ideal corpus website would provide the user a way to retrieve a list of literature based on search criteria. If the corpus website does not contain this functionality, it must provide a way to determine the publication date of each piece of literature without opening the full text. Finally, we met with the project sponsor and discussed the corpora we were planning to scrape. The sponsor was the final determiner of whether we would use a certain corpus for this project. If we had found a corpus that looked like it would benefit us, but the sponsor did not want us to use it for any reason, we would not be using that corpus.

There are four corpora that the sponsor has suggested we use to extract literature data from: Project Gutenberg, Liberty Fund, Archive.org, and HathiTrust. As a team we have decided to add the Library of Congress as an additional corpus to scrape. We have tested these corpora against the above criteria and determined that scraping them will be beneficial to this project. Each of these corpora are designed differently, so they will require specific examination to decide how we will scrape them.

Project Gutenberg is a free online library of thousands of books “with focus on older works for which U.S. copyright has expired” [26]. These books are available in multiple formats including plain text and HTML. Looking into this corpus, it seems that scraping it may not be the best option. The corpus offers what they call a “bookshelf” of categories for every piece of literature contained in the corpus. This could be scraped one category at a time, checking to see if the data of each piece of literature fits into the dates that would be appropriate for our project. However, a better option may be to download the books to our local machine and process them that way. This reduces the amount of HTTP requests we would need to perform and may also reduce runtime.

Liberty Fund is like Project Gutenberg in that it offers a collection of online literature, but it also offers resources for those who may be interested in topics such as economics and law [27]. This corpus has two major limiting factors: cost and size. Each piece of literature contained in this corpus costs money to read, so scraping the site would be impossible without adding a large cost to the project. Additionally, this corpus only contains about 400 books [27], so while these books may be valuable resources for this project, there simply isn’t enough of them to justify paying for access. However, Liberty Fund has an alternative online library that is free to access called the Online Library of Liberty. This online library offers texts from a variety of time periods that are “focused on individual liberty and free markets” [28]. Since this portion of Liberty Fund is free to use, we will include it in the corpora that will be scraped for this project.

Archive.org is a robust Internet archive containing publications in multiple different types of media, including books, movies, and audio. Like Project Gutenberg, the books are available in multiple different formats, and each piece of literature’s metadata and full text are easily accessible. This corpus lends quite well to scraping, and due to its size of 28 million books and texts [29], it may be an invaluable resource for this project. The full text of each book is generated using a Python library that scans through images of the book, which could introduce some inaccuracies in the text. This may prove problematic for this project as we are trying to find best matches for quotes. Inaccuracies are also introduced in the metadata for books, which can be seen on the very first result listed in the book archive, showing a publishing year of 9199 [30]. This is not as important as the full text, but it does still influence the efficacy of our work.

HathiTrust is another online library that contains digitized versions of literature. This corpus was made with researchers in mind as there are tools created by the

HathiTrust Research Team (HTRC) to aid in scraping the corpus contents [31]. Extracting contents of texts from this corpus will be trivial, but the search functionality limits how those texts can be accessed. The search function requires that a search term be entered, so we will not be able to simply access all texts published within a certain time period using the search function. We can get around this by utilizing the tools created by the HTRC as those tools contain functionality for downloading the entire dataset of the corpus. We could also generate search results based on authors that were prominent at the time that *Johnson's Dictionary* was published and then scrape the texts that result from those searches.

The Library of Congress is a large library that is used by the U.S. Congress when conducting research. The physical library in Washington, D.C. contains millions of pieces of published media, and much of these media are digitized and contained in the online corpus [32]. This corpus makes scraping easy as it provides an API that allows a developer to append a string to the end of a URL and extract a JSON representation of that page. Accessing pages using this instead of the normal page is noticeably faster as the HTTP request only needs to retrieve a page of text versus all the HTML contents of the page. Once accessed, the JSON can be processed using dictionary manipulation to easily extract information such as metadata and another webpage containing the full text. The full text is encapsulated in a page of XML which can then be processed with *BeautifulSoup* to extract the full text.

5.3.4 Unit Testing

Unit testing is an important part of our testing process, and there are many ways to do this kind of testing in Python. We will consider two built-in Python libraries that offer unit test functionality, *unittest* and *DocTest*, and one external unit test library, *PyTest*. All three of these libraries were developed specifically for unit testing, but each one offers different functionality that may be more useful for this project.

The *unittest* and *PyTest* libraries are very similar in functionality as they both allow the user to create tests using simple assertion functions in the code. The biggest difference is that *unittest* organizes the tests using object-oriented methodologies while *PyTest* will simply require the developer to create a separate test class that utilizes the desired assertions. To create tests with *unittest*, the developer must first create a separate class that extends the `unittest.TestCase` class, either in

the same file as the code being tested or in a separate file. Within this class is where functions can be defined to perform the unit tests, and then that class can be invoked using command line arguments. In contrast, *PyTest* unit tests are structured as separate functions within the program being tested – there is no extra class that needs to be created to run them. Unit tests in *PyTest* can be run in the same way, but the output is much more detailed than what *unittest* gives. Output from *PyTest* will show each unit test, its run result, and what caused that test to fail, while *unittest* will only show each test and if it failed or not. This extra information is crucial for fixing the code that is behind failed tests [33, 34].

The *DocTest* library provides unit testing functionality in a way that emphasizes documentation. Instead of creating new functions for each unit test, this library allows the developer to create multi-line string comments for an already-existing function that outline the values that should be used to run tests. The function that the comment is above is the one that will be tested with the specified values. Each value outlined in the comment can either have an expected value that it should produce, or an error that passing the value should throw. This method of testing forces us to create detailed documentation surrounding each test which leads to us maintaining better documentation throughout all the code for the project. However, having the tests contained in the same files as each piece of code would not be the best idea for a large development effort [35].

For this project, we will be using the *PyTest* unit testing library. This library allows for the tests to be created in separate files than the code being tested, which is much better than having the tests in the same file as the code because after the testing part of development is complete, the test files can be easily removed or disabled. Despite the similarities between *PyTest* and *unittest*, *PyTest* offers concise test results that give much more information per test than *unittest* does. Additionally, while this library is not built into Python, it will provide us with the best possible unit test functionality that we will need for our project.

5.3.5 Error Logging

Keeping track of errors that occur during runtime is an important part of a complex software project. Typically, errors are logged to the terminal window that the program is running in, but for a project of this size it is better to output log messages to the local file system. The Python *logging* library is a robust library that offers various functions to log errors in a way that can be easily sent to log files in a

specific format. The library will allow us to specify the file name to output to, the format of each log message including a timestamp, and what level of error should get written to the file all from one function call [36]. This makes the process of logging errors to log files incredibly easy and eliminates the need for us to implement a way to do it manually.

5.3.6 Database Manipulation with Python

Creating entries in an SQL database is a large portion of this project, and this is made easy using a Python library called *mysql-connector-python*. This library is part of a larger set of Python database interfaces called the Python DB-API, which has interfaces for a variety of database tools such as PostgreSQL and GadFly [37]. It allows the user to manipulate a database using a few simple steps. First, a connection to the database is made by calling the `mysql.connector.connect` function, passing in a username, password, host IP address, and database name as parameters. Once the connection is established (an exception is thrown if it was unable to connect), the user calls the `cursor` function to get an object representing a pointer to where the next entry will be written in the database. The user can then call the `execute` function on that object, passing in normal SQL code as a parameter. Following these steps allows a user of this library to manipulate a database from within Python in a way that closely mirrors how one would do so manually from a terminal window.

5.4 Database

5.4.1 Acid Principles

ACID is an acronym that refers to the four principles of a transaction: atomicity, consistency, isolation, and durability. These four principles are described in more detailed as follows: [38]

Atomicity

Any time an update is made to the database, such as actually updating values of an existing table entry or inserting new entries into a table, this is called a transaction. The principle of atomicity refers to the condition that transactions should either always be fully completed or not completed at all. This is colloquially

referred to as “all or nothing” because the changes specified by any given transaction will only become available to anyone beyond the user if it is successfully completed in its entirety. Otherwise, the database will remain in the state it was in prior to the execution attempt of the transaction. When a transaction is fully executed and the changes to the database are finalized, this is referred to as the transaction being committed. Otherwise, the transaction is aborted.

Consistency

The data should remain consistent at the start and end of each transaction. There should not be any unexpected changes to the data during the execution of a transaction. Hence, a database that follows this principle is considered to have the property of consistency.

Isolation

Multiple transactions can occur at the same time. In this case, it’s vital to have a sensible policy for handling the execution of such transactions, with respect to whether the partial completion of one should affect the other. Serializability establishes the isolation property in the database management system. If the transactions are made to be serializable, they are handled in a manner where if more than one is occurring at the same time, the outcome is identical to if the transactions were strictly completed one after the other. In this sense, the “progress” of a transaction that is still executing is completely invisible to succeeding transactions.

Durability

System failures and problems with storage media can potentially threaten the state of the database by losing previous transactions that were committed. The durability property refers to the ability of the database management system to recover these committed transactions in any such adverse situation.

5.4.2 Relational vs Non-Relational Database Systems

Relational DBMS

Relational databases are very conventional and have a well-established credibility over the past few decades as a solid choice for data storage. These types of databases have tables that strictly define the structure of each entry by defining the data type of each column.

As the name suggests, one of the primary purposes a relational database serves is to illuminate the relationships between all the tables of data. The most common example seen is the usage of foreign keys, to specify how entries that are uniquely defined by a primary key in one table are referenced in another table.

For example, in one table, there may be entries that uniquely identify each purchase order, and then another table may serve the purpose of storing all the items that are a part of each purchase order as entries. In this case, this relationship would be easily explained by specifying a foreign key in the second “items” table referring to the unique identifier for purchase orders in the first table. This would clarify that all of the entries in the “items” table are related to an entry in the “purchase orders” table.

Non-Relational DBMS

Non-relational databases, otherwise known as NoSQL databases, have vastly grown in popularity over the last decade [39]. One of the main attractions of such a database is added flexibility given that information can be stored in various documents without a rigid structure. This varies from relational databases where each table defines the fields that every entry will have. There are four types of NoSQL databases [39]:

Documented-oriented Databases

Used to store document-oriented information where each key is paired with a complex data structure that resembles a “document”.

Key-Value Stores

This type of database is comprised of different keys where each key maps to exactly one value, similar to a dictionary.

Wide-Column Stores

Although tables, rows, and columns are used in this type of NoSQL database, unlike a relational one, the names and format of the columns can vary within the same table between entries.

Graph Stores

Graph structures with nodes, edges, and properties are used to represent and store data.

5.4.3 MySQL

MySQL is free and open source, first released in 1995 [39]. It is also one of the most common databases used from small websites to very high-traffic ones like Facebook and YouTube. Being free of charge and having a robust amount of documentation and community support, it is a great de facto choice for individuals and small teams to get started. However, one drawback to note is that MySQL can struggle if it is given too many operations at once. In these cases, it may be better to choose a database that has better scalability.

5.4.4 MongoDB

MongoDB is typically what first comes to mind when one thinks of NoSQL databases [39]. It is a document-oriented database, and the documents are comprised of JSON-like structures. It also has a sophisticated partitioning feature called “auto-sharding” where it separates very large databases into much more small and efficient parts called “data shards”. The ease of setting up MongoDB and the professional support surrounding it also makes it an attractive choice for non-relational databases. However, a major feature not available in MongoDB is the ability to join on tables. For simple and straight-forward tasks where there’s not much potential for needing to merge tables together, this is not much of a hinderance, but this functionality is often very useful in many business and data analytics applications.

5.4.5 Consensus

Our team decided early on that a relational database would be the best way to go. Our expectations of the kind of data we will be storing are already clear and easy to define, and we would not benefit much from the flexibility that a database like MongoDB would offer. In fact, relationships between data tables is paramount in our project because we are going to be storing exactly five matches for each quote in another table which we will be able to map using a foreign key.

Our database of choice is MySQL because it is robust, well-known, simple to work with, and free. PHP and Python are commonly used languages for writing an API that sends and receives data to and from a MySQL database when interacting with a front-end of a website. This naturally tends itself to a LAMP stack.

5.4.6 Indexing in MySQL InnoDB

Retrieving entries from a database without any indexing may not expose any noticeable slowdowns if the tables are relatively small, but as the number of entries grows, this inefficiency becomes more and more apparent. This is because without indexing, the query is forced to look through every entry in a table when selecting from it.

Indexing methods are aimed at massively reducing the number of entries that need to be searched for data retrieval. Indexes can be applied to one or more columns to organize entries by.

Our MySQL database will be using the default storage engine: InnoDB. InnoDB's indexing method of choice in most cases is using a *B+ Tree*. B+ Trees implement divide the data into sorting piles with respect to whatever column is being indexed on.

The following illustration is used to further illuminate the high-level implementation of a B+ Tree [40]:

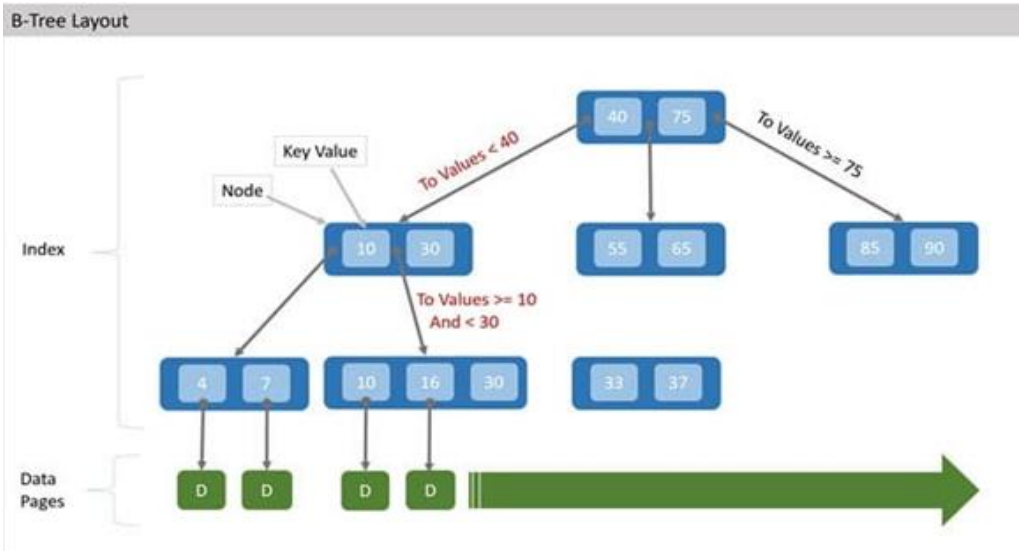


Figure 5: B+ Tree for Indexing Example [40]

In this example, the key-value 15 is being searched. Each node represents the sorting piles that contain the specified range of key values.

Generally, each node contains branches to navigate to the next level of nodes based on the key value.

From the first node, we navigate to the node consisting of key values less than 40, which is partitioned by less than 10, between 10 and 30, and greater than or equal to 30. From here, the next node visited is the one that comes from the branch specified for key values between 10 and 30, as key-value 15 falls in this range. Then we navigate to the branch represented for key values between 10 and 16.

Such segmentation of the data by various numerical ranges in this fashion allows for searches on hundreds of thousands of records to easily be done in less than 100 operations. In essence, the runtime complexity is being reduced from O(n) to O(log(n)) which is a huge reduction and in certain types of applications it can even be decisive when it comes to meeting certain goals.

It is quite possible that InnoDB's B+ Tree indexing in MySQL will turn out to be useful in speeding up queries that return matches for a given quotation.

5.5 API

5.5.1 Python Flask Framework

Flask is a web framework available in Python that makes deploying an API and/or developing web applications a much more accessible task [41].

There is very little “boilerplate” code required when it comes to simply kick-starting the development of an application. It is a lot leaner than the other Python popular framework known as Django and simpler to work with, though for more complex tasks, Django might be the preferred choice.

```
from flask import Flask, request, render_template

application = Flask(__name__)

@application.route('/', methods=['GET'])
def index():
    return render_template("index.html")

if __name__ == '__main__':
    application.run(host='localhost', port='10000')
```

Figure 6: Basic Flask application setup example

This code showcases the basic structure of an application written in Python utilizing Flask.

The first line after importing the Flask framework libraries creates an instance of the Flask class. The last couple lines of code simply ensure that the `run` method is called only when this file is executed as the main program. This is the only boilerplate code typically needed when it comes to writing a Flask script.

In the middle, different routes can be defined each with their own functions to complete specific tasks at the corresponding endpoints. In this basic example, we are simply navigating to the root route, and at this endpoint, the “index.html” view is rendered.

```
@application.route('/home', methods=['GET'])
def home():
    return render_template("home.html")
```

Figure 7: Flask application function example with `render_template`

The above piece of code could be added after the block of code for the root route. This function would be called when the “/home” route is requested and would return the “home.html” view instead.

These are very simple examples that illustrate the basic functionality and typical structure of a Flask application. Without a doubt, these functions can be expanded upon to change what is shown by modifying the html files before returning them. There can also be definitions for as many routes as needed.

Writing an API using Flask is a very flexible task. One of the most common methods of implementing such an API is to integrate the rendering of the front-end with back-end operations such as performing computations or querying a database. This style often utilizes the `render_template` function as shown in the basic example above. However, Flask can also be used to write endpoints that simply return JSON data, as shown below:

```
@application.route('/hello', methods=['GET'])
def hello():
    return jsonify({'message': 'hello'})
```

Figure 8: Flask application function example returning JSON data.

As one can see, the difference in this type of API is that when a request is made to a particular endpoint, instead of writing a function to render a page, data is simply being returned (typically encoded in a JSON). The function `jsonify` is another function that can be imported from the Flask framework which provides a convenient way to encode the data being returned in a JSON object.

Commonly, JavaScript is used on the front-end portion of the application to make ajax requests using jQuery at the corresponding endpoint, to execute the function to receive the data.

In either case, the Flask application in Python serves as a server for the website which fulfills requests to its various endpoints.

5.6 Python Libraries for Making Command Line Applications

5.6.1 sys and argparse

Command line arguments to Python programs can be accessed via the built-in *sys* module's *argv* list object [43]. This list contains each of the command line arguments passed to the program, in order, as strings. The first element of this list will usually be the path to the program itself, relative to the directory from where it was run. One could theoretically use this module alone to create a command-line application in Python, however it would be impractical to do so since the argv list lacks context; i.e., it is impossible to tell from the list alone what each passed argument represents. In order to solely use sys.argv, very strict limitations would have to be set on how the user could pass arguments to the application.

Enter *argparse*, another built-in Python module. This module offers functionality specifically geared towards facilitating the creation of command line applications [44]. To use argparse, one would first create an ArgumentParser object to parse arguments passed to the program from the command line. Then, one may call the object's `add_argument` method to add information about each program argument that they would like their program to support. This information includes information such as the argument's flag, conversion type, help text, default value, whether the argument is required, and more. Finally, one would call the ArgumentParser object's `parse_args` method to parse the passed command line arguments to their specified output types and assign them as attributes of a Namespace object (which is a new one by default). Figure 10 shows how one could use the argpase module to enable a program to parse command line arguments and configure parsing settings for each one of the arguments added.

```
import argparse

parser = argparse.ArgumentParser(description='Process some integers.')
parser.add_argument('integers', metavar='N', type=int, nargs='+',
                    help='an integer for the accumulator')
parser.add_argument('--sum', dest='accumulate', action='store_const',
                    const=sum, default=max,
                    help='sum the integers (default: find the max)')

args = parser.parse_args()
print(args.accumulate(args.integers))
```

Figure 10: An example Python program using the argparse library to parse command line arguments and print either the maximum or the sum of a series of numbers passed to the program [44].

Overall, argparse provides a rather low-level strategy of adding command-line arguments to a program. Its total features include more than those described above, allowing one to take quite a granular approach in deciding how they would like to add support for command line arguments. This advantage is also a drawback however, as reasoning about each detail of each specific argument to add can become burdensome. Other libraries ameliorate this issue by restricting the configurability of a program's command line arguments while still offering sufficient functionality to support command line arguments.

5.6.2 click

The *click* Python package takes a different approach than argparse does for adding support for command line arguments to a program. Instead of creating objects for parsing command line arguments, the click library uses *function decorators* to achieve the same goal [45]. A Python function decorator is essentially a function that is wrapped around another function to extend the wrapped function's functionality [46]. With the click library, one can use the click.command and click.option function decorators to specify functions that take command line arguments and add support for specific command line arguments to those functions. Figure 11 displays an example of how one would use the click library to add support for command line arguments to a program.

```
import click

@click.command()
@click.option('--count', default=1, help='Number of greetings.')
@click.option('--name', prompt='Your name',
              help='The person to greet.')
def hello(count, name):
    """Simple program that greets NAME for a total of COUNT times."""
    for x in range(count):
        click.echo('Hello %s!' % name)

if __name__ == '__main__':
    hello()
```

Figure 11: A sample Python program using the click library to enable support for command line arguments. This program takes two arguments, --count and --name, and prints the --name argument --count number of times.

While perhaps not as configurable as the argparse library, the syntax of the click library is arguably more expressive and readable since the command line arguments supported by a function are placed directly above it in the program's code. For most command line programs, which do not require the high-resolution level of detail that the argparse library provides, the click library offers an appealing balance between usability and functionality.

5.6.3 begins

Finally, the *begins* library elides much of the customizability options of the argparse and click libraries in exchange for making the process of adding command line arguments to an application as ergonomic as possible. Like the click library, begins uses function decorators to add support for command line arguments to an application. To use begins, however, one need only adorn a function with one function decorator, rather than with two when using click. After this is done, support for command line arguments can be added to a function by adding arguments to the function itself [47]. begins infers information about these arguments, such as their desired output type and help text, by exploiting Python's optional type hint features (which are built into Python versions 3.5 and later) [48]. Figure 12 illustrates how one could use begins to enable a program to support simple command line arguments.

```
import begin

@begin.start
def main(count: int, name: 'Enter your name'):
    count = int(count)
    for _ in range(count):
        print(f"Hello, {name}!")
```

Figure 12: A sample Python program using the begins library to add support for command line arguments. This program takes two required positional arguments, count and name, and prints the --name argument --count number of times. Note that the count variable needs to be cast to an int before being used as a number; begins offers support for automatic casting but that is not used in this example.

Begins is explicitly suited for small applications, or those which do not require complex command line arguments. Its author has even stated directly that it is "not intended to replace the rich command line processing needed for larger applications" [47]. That being said, its simple interface makes it an appealing choice for our purposes, since our program will not require many different command line arguments.

5.6.4 Conclusion

As we elected to use command-line arguments for the algorithm's inputs, we will not be able to benefit from a tailored user interface. With this in mind, deciding upon a powerful API to handle these inputs is a deceptively important step of the project's development.

Our team has decided to use the begins library for adding support for command line arguments to the fuzzy search algorithm program. We have chosen this library because our program does not require complex command line arguments, and parsing simple arguments is what the begins library excels at. For our project, begins' extremely ergonomic API more than makes up for its limited ability to customize command-line arguments.

6 Overall System Design

6.1 Front-end Web Application

6.1.1 Mockups

When finalizing the frontend design, we decided to keep our initial design philosophies for a multi-page layout. Search options will be dynamic based upon the chosen quote match grouping. A final frontend mockup can be seen here:

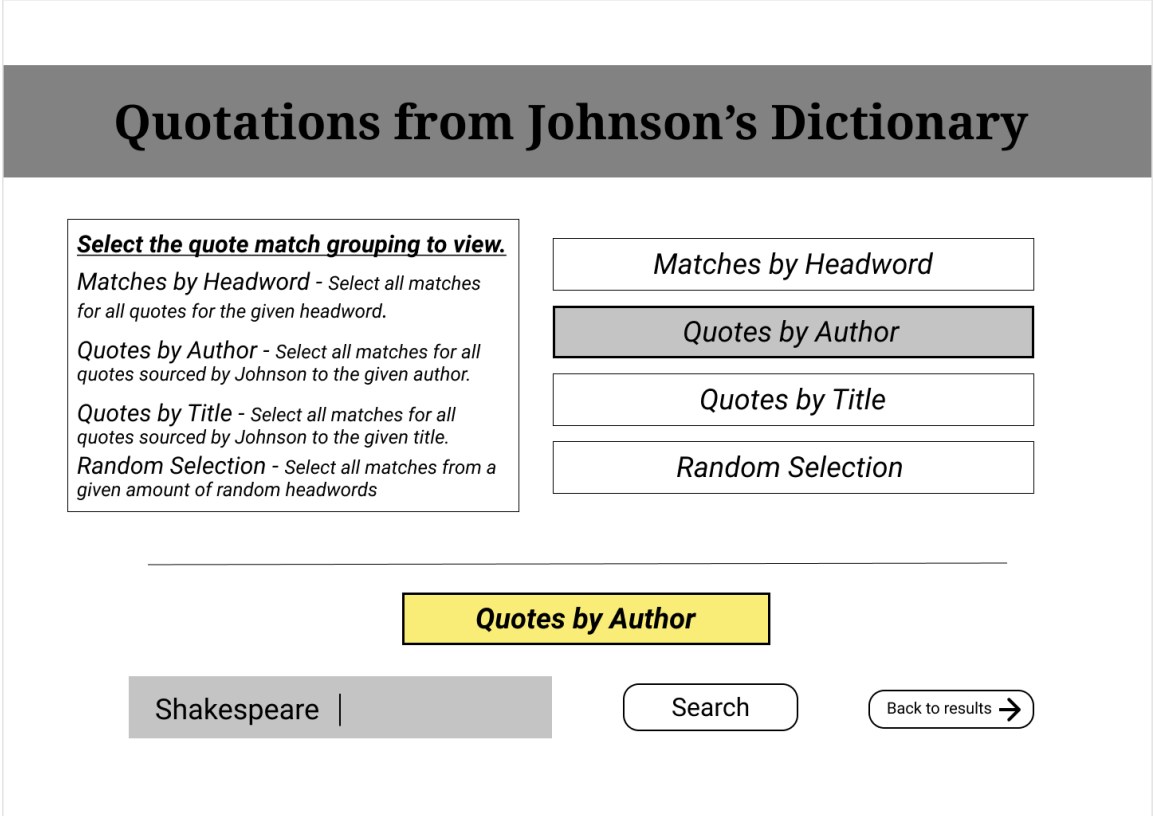


Figure 13: Final dynamic multi-page mockup

This home page mockup keeps the simple approach our group has maintained on this project since its inception while also being immediately digestible for even novice researchers.

Finalizing the search results page was slightly more difficult. The spreadsheet grid layout is not difficult to display, but it must also account for any uncommon match situations that the Quotations from *Johnson's Dictionary* project may run into.

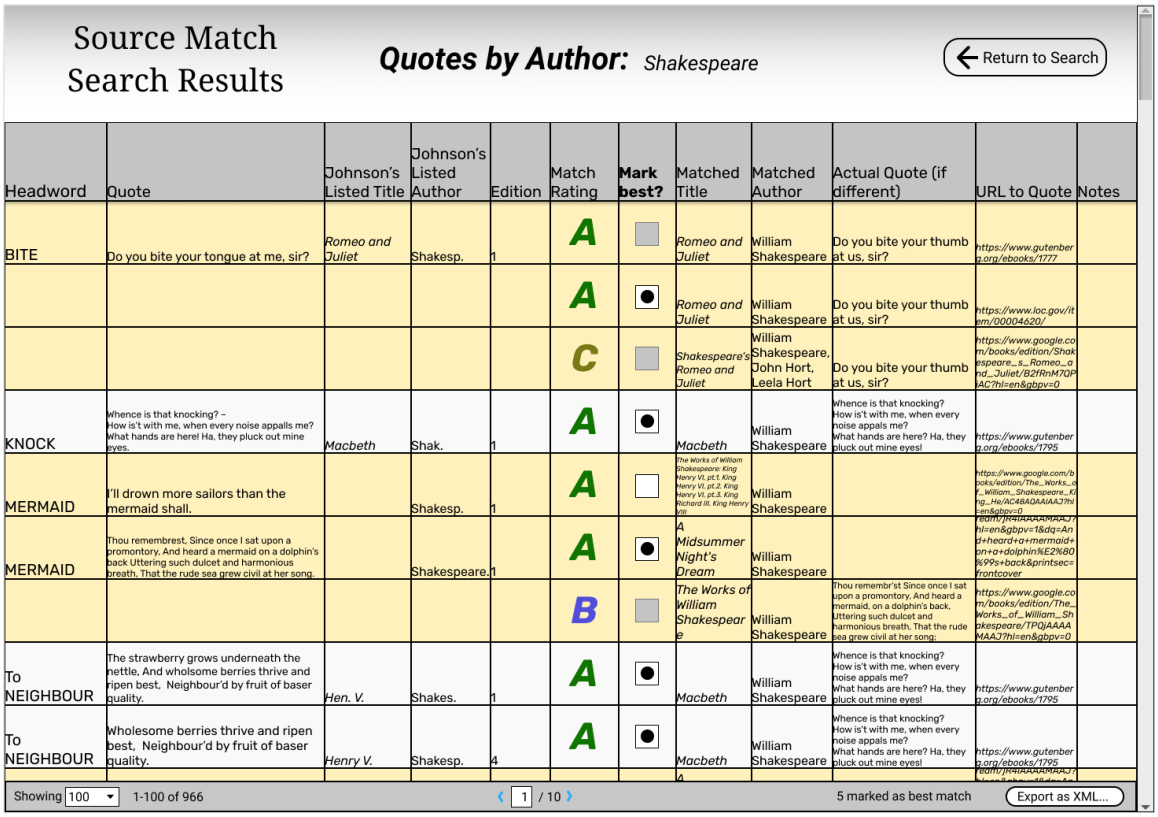


Figure 14: Final multi-page results mockup

This mockup will be the most faithful to the final deliverable frontend and contains a few changes from our initial works. Notable is the inclusion of the Edition column, which will be important when running into headwords that have different assigned quotations in different editions of *Johnson's Dictionary*. In this example, To NEIGHBOUR's quotations are slightly different from edition 1 to edition 4. Though the search algorithm will condense similar-enough edition quotes into the same entry, in some situations such as this the algorithm will treat the editions as different quotes. The frontend will handle this situation the same way, displayed consecutively as shown here.

Another key difference is the navigation panel. It has been reduced in size and improved with additional user features. The user will be able to modify how many results are displayed per page via a dropdown. Which entry numbers in particular are being viewed will be dynamically tracked next to the control.

The page count has been updated from a static “Load More” to a more concrete page system. Users will be able to use the left/right navigation buttons to change page, or they may choose to type a specific page number to instantly open any page. This page system will allow users to better keep track of different “groupings” of matches as they peruse the search results.

6.1.2 UML Use Case Diagram

The UML Use Case Diagram for the web app is as follows. This Use Case diagram demonstrates the uses of our frontend that the user will expect to encounter. These use cases were themselves used to frame the design and layout of the web app. To begin designing the frontend, the use cases any given user will expect out of our final product must be made explicit.

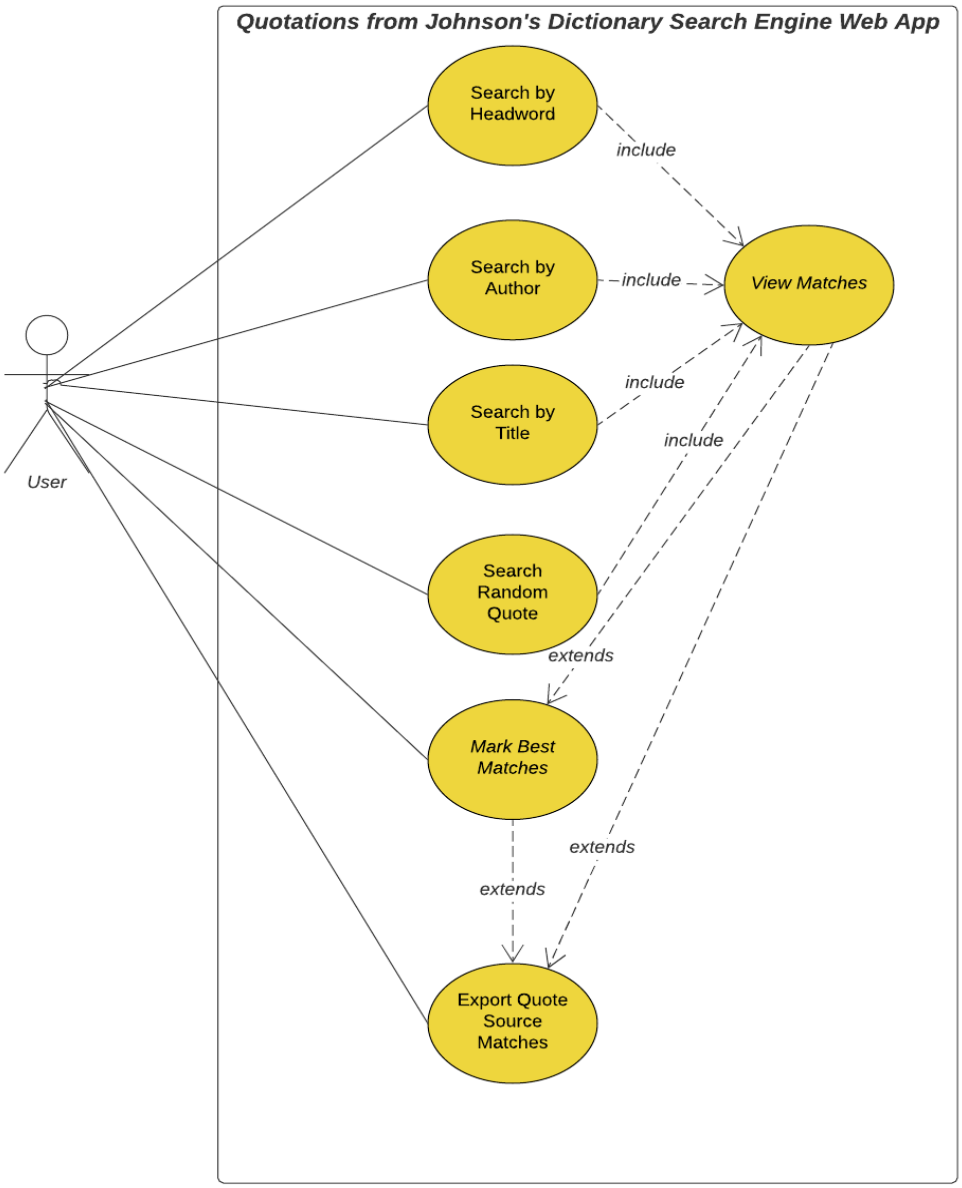


Figure 15: UML Diagram

Aesthetic decisions and the technical implementations used to achieve them could only be decided upon once this diagram was confirmed.

The second step in prototyping a UI is to relate the actual use of the search engine to those that will be using it. The Quotations from *Johnson's Dictionary* project is going to be predominately used by academics, and not the general public, and so there is a higher priority given to data display over aesthetics. Though everyone can appreciate a well-animated UI with user-controlled color

options or personal settings, in our case these features would make the rather simplistic input and output data more confusing. A balance must be struck between overwhelming the user with unformatted spreadsheet data and confusing the user with ‘pretty’ visuals.

The academics that will be using our search engine are almost certainly unfamiliar with shells, and so our project will be utilizing a graphical user interface (GUI) to better guide the user around our software. A simple, clearly marked window interface will vastly improve the end user experience of our project.

6.1.3 Final Frontend Design

The final design of the frontend is generally similar to the final mockups, though with support for features like the Best Match export and the addition of the DataTables search results grid.

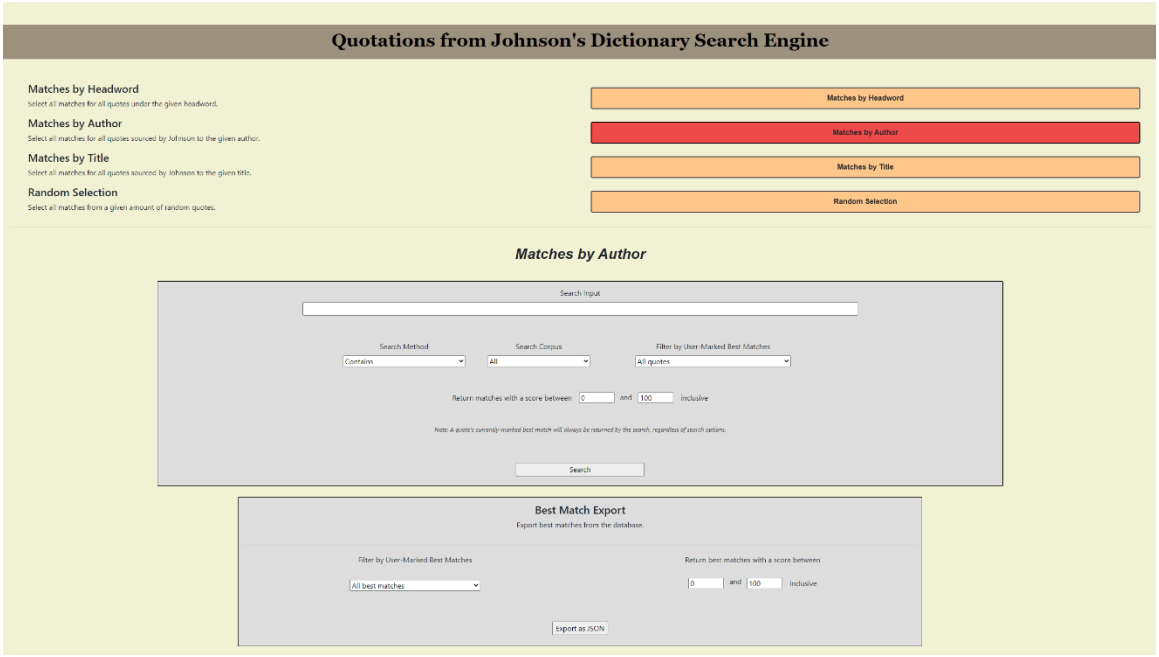


Figure 16: Home Screen

The results screen’s results grid looks notably different. To reduce search load times in the final implementation, the search only returns quotes, which are then

put into the DataTables results grid. Each row is a quote, and clicking the quote queries the API for that quote's matches, which are then put into a subtable.

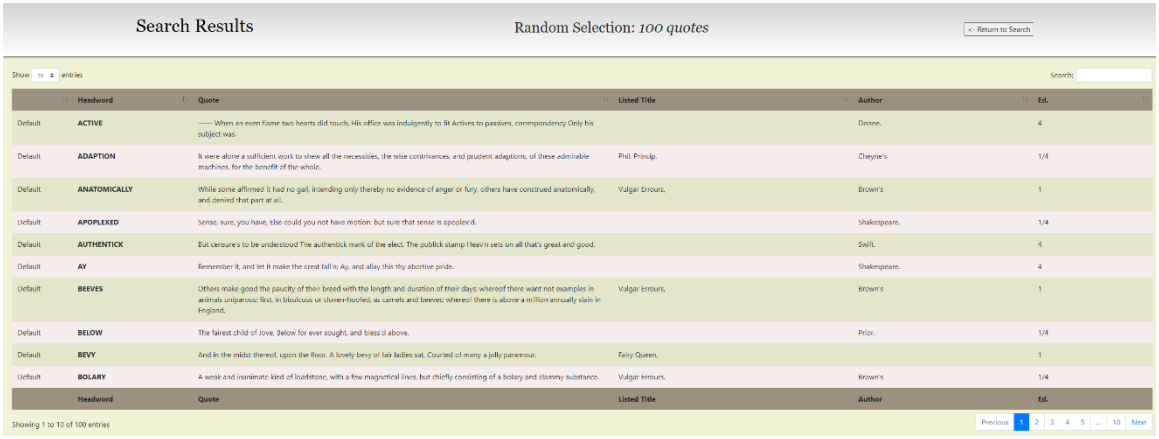


Figure 17: Results Screen Example – Random Selection

The matches subtable contains all the match information needed for the user to determine the Best Match. The matches are in descending order by match score; the default best match will be the first match in the table. The final grading system used in our website is the algorithm-assigned 0-100 score, discussed in greater depth later in this design document. The match subtable can be closed by clicking the quote row once more.

Figure 18: Matches Subtable

6.1.4 Implementation

Overview

The frontend is the part of the search engine that the user will be inputting

commands and viewing results from, and so its purposes can be listed into those two categories. The inputs and output functions of the frontend are of utmost importance and must be laid out before any aesthetic design is finalized.

Input from the user is not required to operate our developed quotation search algorithm. The algorithm will be first run independently before the project is complete, and the frontend will simply be an interface for the user to request results from the database.

Search Inputs

The user must first select which quotation matches to pull from the database. Users will be able to search through sets of quotations with a filter and choose which quotations they would like to see the results for. Quotation sets include:

- Matches by Headword, which will select all matches for all quotes for the given key word (e.g., all quotes for headword KEEN).

- Quotes by Author, which will select all quotes attributed by Samuel Johnson to the given author (e.g., all quotes from Shakespeare). *Johnson's Dictionary*'s various editions and the attributed quotations within them often have misspelled author names, so a warning must be given that some false negatives may have been left out of the results.

- Quotes by Title, which will select all quotes attributed by Samuel Johnson to the given text (e.g., all quotes from *Othello*). *Johnson's Dictionary*'s various editions and quotations have different spellings of certain document titles, so a warning must be given.

- Random Selection, which will select all matches from a set amount of headwords randomly. Uniquely with this option the user will be able to choose the number of headwords pulled (25, 50, 100, 200, 500).

This quote selection will by default choose a 25-word Random Selection.

Once the user selects a quotation set to search, they must then supply the search variables. This will always happen before any search results are pulled regardless of single- or multi-page implementation. In searches by Headword, Author, and

Title, the user will free type enter the search term. Due to Johnson's widespread abbreviation of quotation authors and source titles, these search terms are not feasible to convert into a list the user can search through. Headwords are identical between editions of *Johnson's Dictionary*, and do not need a list. If the user selects Random Selection, then the search input changes to a dropdown with preset amounts of headwords to search for. This defaults to 25, so no user search input is necessary to continue.

Quotes can also be filtered by whether or not a user set its Best Match. This allows for users to filter quotes that have already been manually evaluated.

Quote matches have options to restrict returned matches based on the match's sourced corpora and the match score. These were suggested based on sponsor feedback. Additionally, no matter the search options, quotes returned in the search results grid will always have their Best Match displayed, even if the Best Match does not fit into search criteria. This is to ensure that users can always see the default Best Match choice,

Once the user is satisfied with their inputs, the Search button is used to query match results.

Search Results

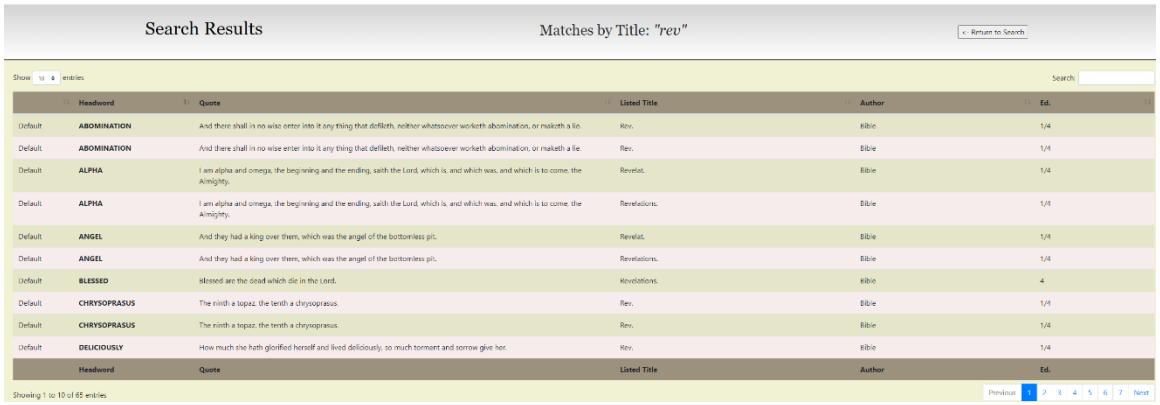


Figure 19: Example search grid

The search grid is designed in as user-friendly a manner as possible. Quote rows are striped in color and change color to red when hovered to make entries visibly distinct. Quote matches are queried dynamically according to the user's search options when the quote row is clicked, and the returned matches are put into a

subtable. This subtable significantly reduces visual clutter and its dynamically-queried nature makes the entire search results function faster and more responsive. An example of the matches subtable can be seen below. This example shows a user-set best match:

Figure 20: Match Subtable with User-set Best Match

The search results grid is implemented using the jQuery plug-in DataTables. Without any additional configuration, DataTables supports many beneficial functionalities such as results pagination, text filtering, and column sorting. DataTables also natively supports table creation from JSON, so little backend modification was needed to support its inclusion. The grid will display each quotation match's actual title and actual author, as well as a link (if possible) to the source text.

Match Ratings and Mark Best Match

There will also be a Match Rating system applied to quotation matches. This system will gauge how likely a certain match is to be the original source of the quote Johnson used in his dictionary. The system is currently outlined as a 0-100 grading system that uses the match score applied to matches by the fuzzy search algorithm.

These grades are only for the researcher's benefit and have no impact on the retrieval of search results from the server nor on how the algorithm runs.

The user will also be able to input Best Match decisions from the matches grid. The user will be able to mark which source match for each quote is the Best Match by checking the radio button in the results grid. Only one match may be marked as the Best Match for each given quote. Keywords with multiple different quotes can have a Best Match marked for each quote (still with only one Best Match per

quote).

Each quote will have a Best Match entered by default. The default-marked Best Match will always be the highest-scoring match, with only one match chosen in the case of a tie.

Whenever the user marks a match as Best Match, the web app will add the match to the BEST_MATCHES table in the database. Quotes that have their Best Match set by a user will be marked as such in the database, giving the quote a “User-set” flag in search results and allowing the user to filter by this flag in exports and future searches. The frontend will check the table whenever results are pulled and will update the table any time a match is marked as Best Match consistently in real time.

6.1.5 JSON Data Export

Users are able to export matches marked as Best Match from the Export section of the Home screen. The exported matches can be filtered by whether or not the Best Match was set by a user as well as by a range of scores.

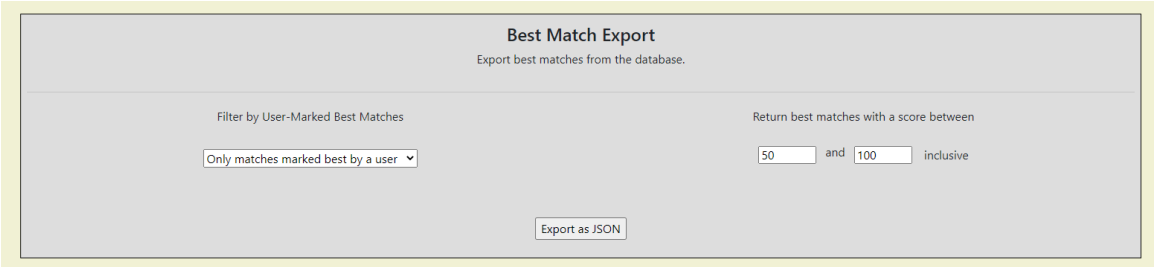


Figure 21: Best Match Export

When the Export as JSON button is pressed, the API will pull Best Matches matching the export criteria as well as their associated quote data and place the matches into a JSON array. The format of the JSON was decided with sponsor feedback.

```
[
  {
    "author": "Johnson's listed author",
    "content": "Quote from dictionary",
    "edition": "edition",
    "headword": "Quote Headword",
    "title": "Johnson's listed title",
    "sourceAuthor": "Source-matched author",
    "sourceContent": "Source-matched quote",
    "sourceLCCN": "LCCN (-1 if not LoC source)",
    "sourceTitle": "Source-matched title",
    "sourceURL": "URL to source-matched work"
  },
  {
    ...
  }
]
```


Figure 22: JSON Export Format

We discovered during implementation that the Best Match Export JSON was rather large in size: so large, in fact, that attempting to output the JSON directly caused the PHP API middleman files to crash. To fix this, the JSON of matches is compressed and saved as .zip file.

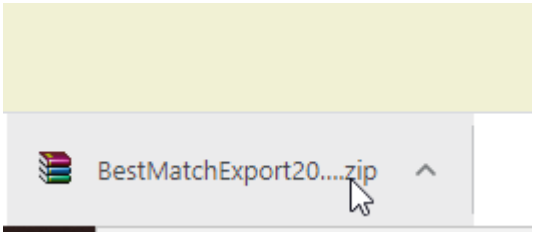


Figure 23: Example Export

Once the user has completed inspecting the search results, the entire grid will be exported in XML format. Matches in the results grid marked as Best Match will have a corresponding flag added to their XML export entry. The Export as XML… button will open the standard Windows prompt to save file output. This XML export will be designed to smoothly assimilate into the larger *Johnson's Dictionary Online*

project.

These user inputs and outputs constitute the entirety of the functionality of the *Quotations from Johnson's Dictionary Search Engine* web app. The tool's internal-use nature means that the average user's experience is the *only* experience: to that end, the average user experience must be made as efficient as possible.

6.1.6 Bootstrap

Overview

The frontend will be constructed using Bootstrap 4, an easy-to-use and universal frontend framework utilizing HTML, CSS, and JavaScript. Bootstrap's powerful built-in data manipulation features has made it a popular tool for designing frontends for projects dealing with large or variable volumes of data. This makes it perfect for integration into the Quotations from *Johnson's Dictionary* search engine.

Bootstrap's light nature also makes it great for the less-intensive areas of the frontend, such as the home screen. The use of JavaScript will allow the frontend to react to user commands and respond in real-time without requiring a page refresh. Given that the project's final design focuses on a few important pages rather than many different pages, this is a huge boon for development. The fact that it is the same framework as the more-important search results grid also lowers the framework's skill floor, enabling faster implementation.

Grids

Translating the finalized frontend designs into Bootstrap is fairly simple. Bootstrap columns are organized in sets of 12 'units' across. This means that we can allocate up to 12 different columns per row of the search grid before another row is created.

```
<div class="row">
  <div class="col-md-1">.col-md-1</div>
  <div class="col-md-1">.col-md-1</div>
  <div class="col-md-1">.col-md-1</div>
  <div class="col-md-1">.col-md-1</div>
  <div class="col-md-1">.col-md-1</div>
  <div class="col-md-1">.col-md-1</div>
  <div class="col-md-1">.col-md-1</div>
  <div class="col-md-1">.col-md-1</div>
  <div class="col-md-1">.col-md-1</div>
  <div class="col-md-1">.col-md-1</div>
  <div class="col-md-1">.col-md-1</div>
  <div class="col-md-1">.col-md-1</div>
</div>
```

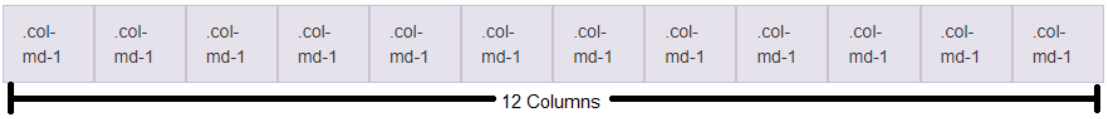


Figure 24: Base Column Layout

In this example, the ‘md’ flag denotes a medium-size column primarily for desktop display. We can map out the finalized mockup’s columns to match these columns:

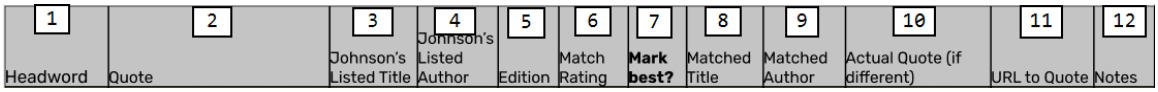


Figure 25: 1-to-1 Mapping

When implementing, however, this runs into issues. Bootstrap resizes columns according to how much space they are given (a column of size 4 will take up a third of the row) but cannot be less than 1/12 of the row. Though there are 12 columns to map and 12 available in Bootstrap’s grid system, the automatic spacing of the columns within its HTML container will make every one of these columns of the same width. As this mockup header shows, however, the more important columns are wider to draw attention from the user as well as to hold the longer

length of the quote string. Additionally, the Notes field is seldom used in the Quotations from *Johnson's Dictionary*'s search algorithm, so always displaying it as empty will in effect waste the last 1/12th unit of the row. There are a few ways to address these shortcomings.

Bootstrap's famous grid system was built with an exact 12-column row display in mind, so skirting around the restriction in keeping with project requirements necessitates some creativity. One way would be to separate each row into two containers, in effect having two sets of columns for one search result. This would enable more precise column size-tuning but would almost certainly cause undefined edge case behavior. We could also drop the Notes section entirely and integrate its contents (if they exist) into another column where it could still be viewed by the user without leaving the software. This option is less attractive since hiding required information for the user inside of another column makes the search engine less user-friendly: this flies in the face of the entire design so far, and so cannot be used.

The most likely solution will be to go above 12 columns in a single row while also using overflow scrolling.

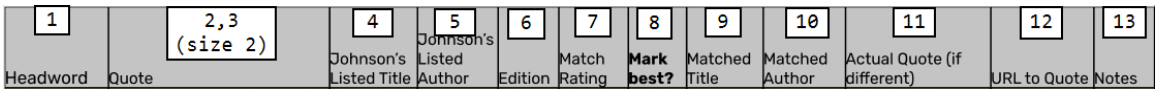


Figure 26: Column Overflow Solution

In Bootstrap, any columns whose size will extend out of the 12-unit display size will be inserted as another row in the grid. We can take advantage of this to only display the Notes column when it has data while also giving us another 1/12th of the screen to apply to the most important column, the original Quote. However, this leaves the Actual Quote column (or any other result column with lots of text) too small to possibly hold the info it needs to.

This is rectified with the base functionality `<div class="overflow-auto">` which will enable a scroll bar on column entries that overflow the box. Though the overflow is not responsive to window resizes, it will be sufficient for routine user results parsing.

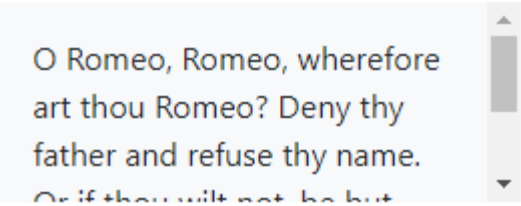


Figure 27: Overflow Solution

When a Notes check reveals that the field is not empty, we can add it as a 13th column. Bootstrap will see this and insert the column after every other entry, in effect being a new row on its own. Though this will disrupt the otherwise even nature of the search results grid as it has been designed, it allows for most search results to be displayed in an easier format for the user to interact with whilst also making the inclusion of the rare Notes field obvious.

JSON Parsing

The search results grid will be obtaining result entries in JSON format from our configured API, explained elsewhere in this document. JSON (JavaScript Object Notation) is the industry standard for sending packets of data to and from users. Standards aside, we have decided that a JSON data packet would translate most directly to the search results grid of the frontend.

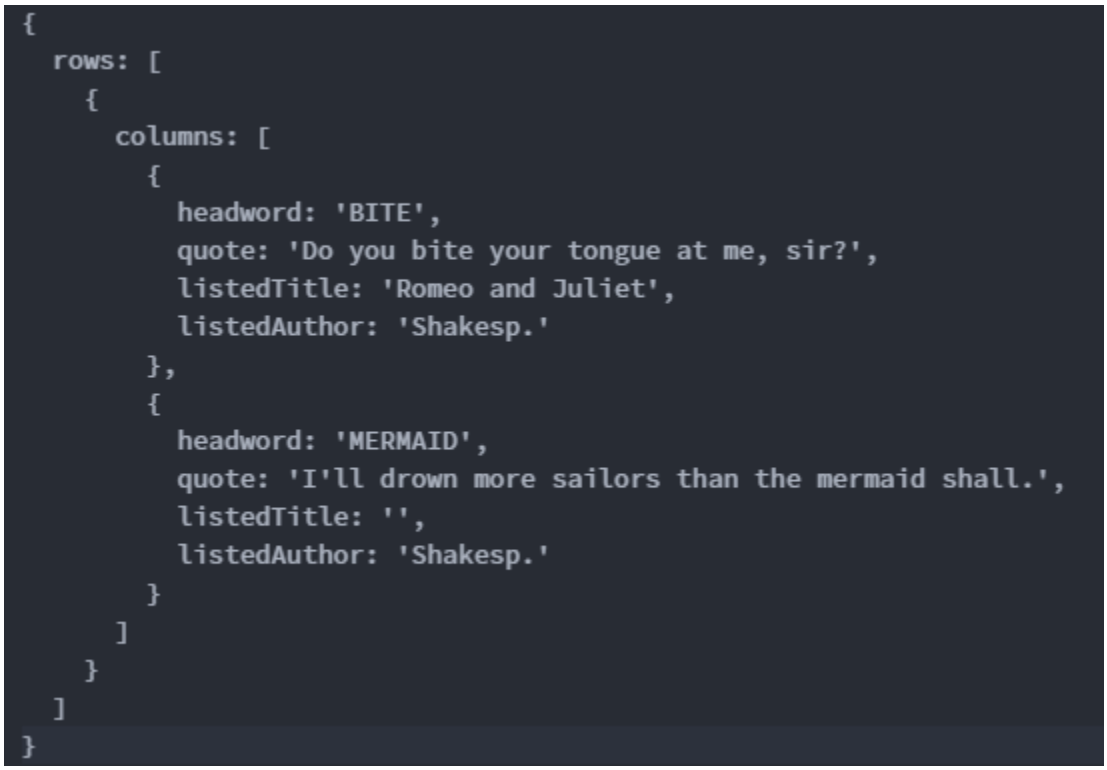
As its name suggests, the object format of the JSON will be most appropriate for translation into a search grid. With each API call, the frontend will receive a list of rows (search results) as well as the data for each column of each row. This will be a more-or-less transliteration of the search results as they are stored in the database: there is not much difference between a SQL data table and a search results grid, so a direct JSON mapping will make data assimilation easier.

Parsing with Ajax

The actual handling of JSON retrieval from the frontend is simple and will be done with jQuery. JQuery's `getJSON` method will enable easy API requests that allow

for the frontend to send user-specified search criteria, lowering the performance needs of the frontend once the results JSON is returned. These requests do not require a page refresh [49].

The JSON rows list will be of variable size, even though some options like Random Selection will have a static entry return size. This is due to the existence of multiple-match scenarios, which are treated as additional rows for a single headword, and the fact that variable entry return sizes for options like Quotes by Author will require additional care anyway. The solution proposed is a recursive JSON parser that will be able to sort through a returned JSON of theoretically any length and map them into Bootstrap columns.



```
{
  rows: [
    {
      columns: [
        {
          headword: 'BITE',
          quote: 'Do you bite your tongue at me, sir?',
          listedTitle: 'Romeo and Juliet',
          listedAuthor: 'Shakesp.'
        },
        {
          headword: 'MERMAID',
          quote: 'I'll drown more sailors than the mermaid shall.',
          listedTitle: '',
          listedAuthor: 'Shakesp.'
        }
      ]
    }
  ]
}
```

Figure 28: JSON Format

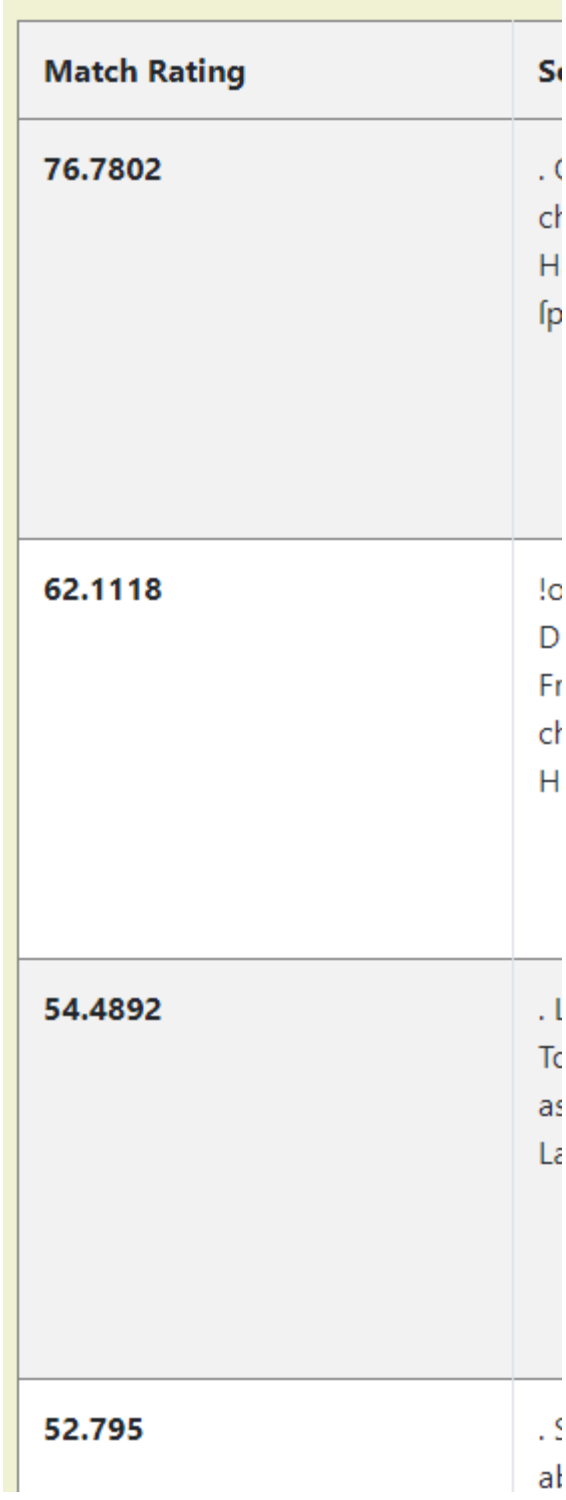
This recursive parser was later adapted into the DataTables search results grid. With the addition of DataTables, we could modify the match retrieval to retrieve only quotes, with no matches and no nested JSONs for those matches. The quotes are filtered according to search settings and fed via successful AJAX call into the DataTables constructor. DataTables then automatically creates the results grid.

Quote source matches are viewed by clicking the associated quote row, which retrieves the matches and places them into a subtable created below the quote row. This subtable is not DataTables, but base HTML, as we decided that the functionalities provided by DataTables were not worth the additional hassle of implementing the DataTable dynamically. This also meant that we could apply the old recursive parser's logic to creation of the new subtable, as the matches returned are still returned in the same format.

Grading System

The search results match ratings are assigned by the fuzzy search algorithm a normalized score between 0 and 100. These score values are stored and sent as a float. While we initially opted for a standard ABCDF letter grading system, we later decided to simply use the normalized score as-is. Not only was this simpler to implement, but we decided that the normalized score was more accurate and immediately understandable for the user.

An additional benefit of this scoring system is that we can sensibly use it to organize matches stored in the quote match subtable. Matches are listed in descending order based on match score, which is not only easy to understand but also brings attention to matches with the highest scores: these matches are significantly more likely to be a true source match.



Match Rating	S
76.7802	. C ch H ſp
62.1118	!c D Fr ch H
54.4892	. L To as La
52.795	. S ab

Figure: Match Score and Descending Order Example

6.2 Backend Database and API

6.2.1 Design Summary

Each of the 299,588 quotations are stored within two tables, Quotes and Quote Metadata. The Quotes table simply stores the actual quotation, while the Quote Metadata table stores the source information – headword, title, author, and edition - of each quotation, since sometimes there can be multiple entries of source information for the same quotation.

One of the first tasks is to scrape all the written works of every designated corpus. These are stored on the server as text documents, of which the information is fed to the database and placed in the Work Metadata table. Each entry points to a specific work and contains the relevant information such as its title, author, and the URL and file path so that the text document of the work in question can be accessed.

Since the idea is to execute the fuzzy search algorithm beforehand to collect the matches prior to deployment, the top five matches for each quote are stored in its own table. The foreign keys both for the original quotation that is being referenced as well as the specific work in which the match was found are included for each match. The exact quotation of the match is also be included for each entry.

Finally, a table for the users and their credentials is also be maintained. This will be simple as the application is intended to be internal and available only to a predefined list of users.

The Best Matches table also contains the identified best match for each quotation. The id from the Quotes table, id from the Matches table, and id from the Work Metadata table are all included for each entry. Additional information stored is the id from the Users table, which is null if the best match is currently automatically generated or contains the id of the user who has marked the best match.

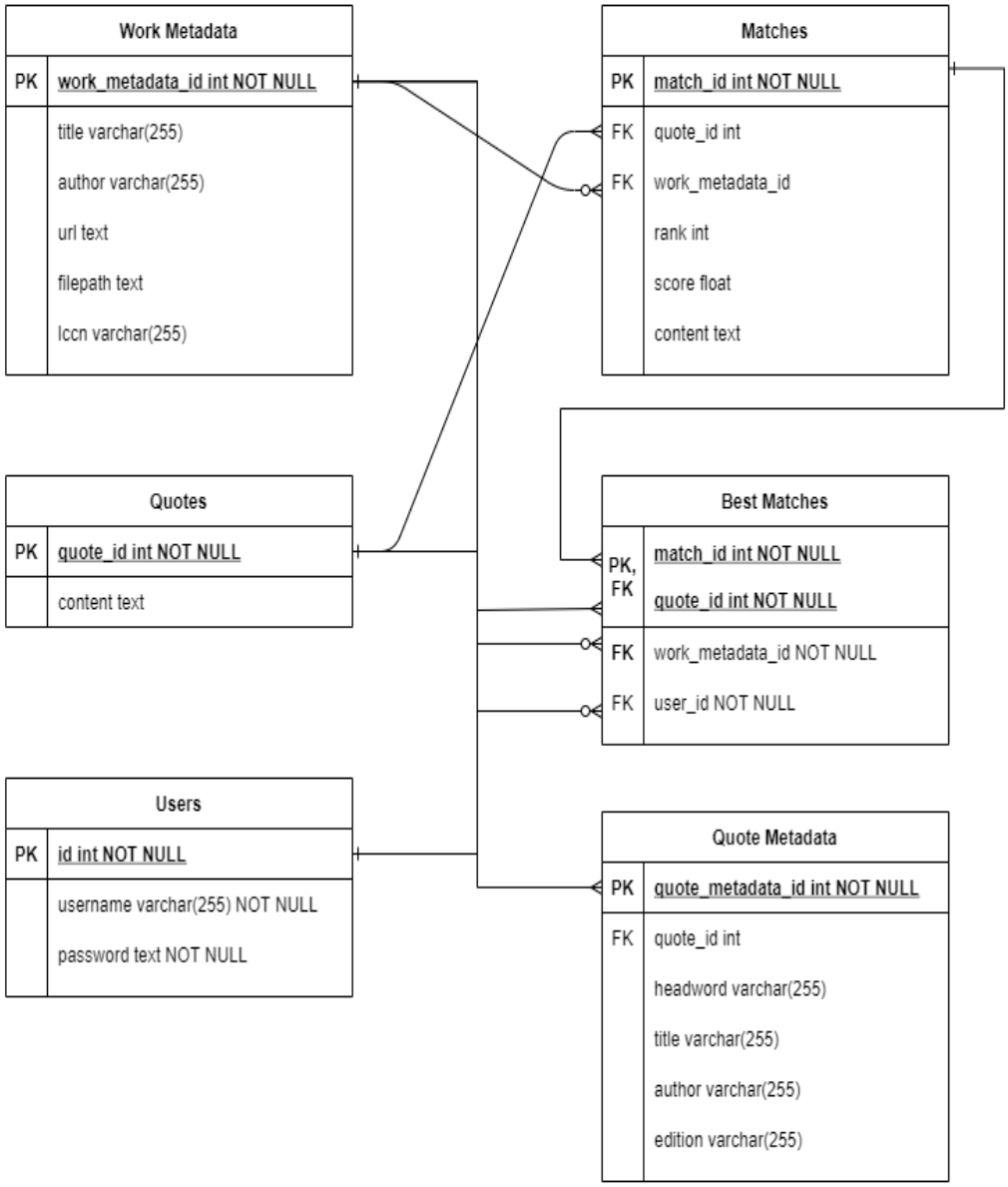


Figure 19: Database Entity Relationship Diagram

6.2.2 API Methods

The data that is kept in the database needs a reliable and secure way of being accessed. We do not want client-side JavaScript making calls to the database as this would compromise the security of the database. Instead, from the front-end, typically using JavaScript, calls will be made to the back-end API which will retrieve the data from the database. In this section, we will briefly discuss the different functions that are included in the API that the front-end or any other client will be able to use to have access to the data it needs:

Login

This function simply takes in the username and password that is provided in the forms and perform a query on the users table to see if exactly one match is found with that username and password. A message encoded in JSON indicates whether the login was successful and is returned to the client.

Home

This function renders the homepage if the user is logged in, checked for by the presence of a user token. Otherwise, the user is redirected to the login page.

Results

This function renders the results page if the user is logged in, checked for by the presence of a user token. Otherwise, the user is redirected to the login page.

Logout

This function deletes the current token assigned to the user and redirects the user to the login page.

Get Matches by Title

This function takes in the title being searched for and any other search criteria, and then performs a query on the quotes table accordingly. The query returns the results, and their information is provided to the client within a JSON object. Then, in the future, additional request, “Get Quote Matches” can be made on each quote one by one to display the top five matches for each quotation and the best match information.

Get Matches by Author

This function fulfills the same purpose as “Get Matches by Title”, but instead takes in the author being searched for.

Get Matches by Headword

This function fulfills the same purpose as “Get Matches by Title”, but instead takes in the headword being searched for.

Get Matches by Random

This function fulfills the same purpose as “Get Matches by Title”, but instead serves a random specified number of quotations.

Get Quote Matches

Retrieving matches is done in a two-step process overall for each quotation, and after a client calls one of the “Get Matches by…” functions to get a list of quotations satisfying the search criteria, it can then call this function to retrieve the top five matches and best matches information. This helps keep query times reasonable overall as to return all quotations and matches at once could become a massively slower process due to the sheer amount of data.

Unset Best Match

This function unsets any best match marked by a user and resets the best match as the one automatically set based on highest score.

Set Best Match

This function allows the best match for a quotation to be changed to the one marked by the user and override automatically set one which is the initial state.

Export

This function returns a zipped JSON file containing the best match information of every quotation and the source information of both the quotation and the match. Results can be restricted by minimum and maximum match rating as well as whether the best match is user-marked or automatically set.

Get Active Tokens

This function returns the list of all currently active user tokens.

Delete Old Tokens

This function deletes all user tokens older than 1 day.

6.2.3 API Accessories

PHP Scripts

It has been found that the Flask API must be mounted on the server's localhost, which cannot be called directly by the user's browsers. As a result, the frontend must call files on the server to interact with the remote localhost.

The implication of this wrinkle is that requests cannot be made directly to Flask API endpoints by the frontend. It requires the addition of intermediate PHP scripts, because these can be directly called by the site, which each are delegated to make a CURL request to their assigned API endpoint.

User Tokens

Another effect of the inability to interact directly with the Flask API by the frontend is that it requires a custom implementation of session management logic, since the Flask's built-in functionality cannot be accessed in a proper manner.

The Flask API maintains a list data structure which tracks active user tokens. The token itself is a unique and random 9-digith number, and the list houses the token numbers, along with the database user_id associated with each and their UTC timestamps of creation as elements.

A presence of a valid token, one that exists, is required to navigate the website. Most importantly, it's required to set and unset best matches, as it needs to be tracked in the database who has marked a best match, if anyone at all. If there is no user token present, then the frontend will not supply such a token number in which case the Flask API will then serve the login page.

Cleanup Script

A continuously running script in the background is used to periodically remove old user tokens and export files.

Due to the limitations of how much data can be sent through PHP, the export files must be served indirectly where the Flask API first saves a zip file locally, and then the frontend serves the file to the user. Tokens and timestamps are used to avoid any race conditions and to ensure the correct export file is delivered. Over time, if

these files are not removed however, they will take up space in the server. Additionally, there are some instances where old and unused user tokens can accumulate in the data structure.

Both potential issues are resolved using a separate script that runs with an infinite loop. The time.sleep() function is used to keep this thread suspended, and it is kept in this state for one hour at a time. As a result, the thread by default doesn't execute, except every hour – at which all tokens older than 1 day and export files older than 2 hours are deleted.

6.3 Algorithm

6.3.1 Design Summary

Once the preprocessing of the text has been completed, we need a way to accurately compare two strings and obtain a score that grades their similarity. There are many different metrics available for use, but our initial instinct is to settle on a combination of two metrics that provide a broad coverage of possible string similarities: Jaccard index and Levenshtein distance.

The first metric is the Jaccard index, and in general, this value indicates the overlap (commonality) between two sets. For our purposes, this value will tell us how many words are shared between the token and the original quote. Alternatively, it can also be used to calculate character similarity, but we have decided to stick with word sets for now. It seems likely that calculating word-level overlap will give us more accurate results than character-level overlap for the goal of our project, but we will test both to make sure this is true. The actual calculation is simply the intersection of the sets divided by the union of the sets.

So, we have a metric that covers the overlap between words but gives no indication of their order. To supplement this deficiency, we will next calculate the Levenshtein distance, which is essentially the edit distance between the two strings (the minimum number of single-character edits required to transform one string into another). Edits are defined as insertions, deletions, and substitutions.

It is important to note that the values returned by the Levenshtein distance calculation are going to be normalized. The Jaccard index naturally returns a value between zero and one because of its fractional nature, but the Levenshtein

distance is different. Consider the difference between a two-character edit in a four-letter word and in a ten-letter word.

Obviously, the edits are much more impactful in the four-letter word than the ten-letter word (port → pact VS. singularly → singular). To represent this difference in importance, we will normalize Levenshtein distances with respect to the length of the larger string, first by subtracting the distance from that length and then dividing by that length [6]:

$$LD_{normalized} = \frac{lengthOfLargerString - LD}{lengthOfLargerString}$$

The general algorithm will make two passes over the list of tokens. The first pass will compare each token with the quote using the Jaccard index. This calculation will produce a similarity score between zero and one. If the score is below a certain threshold, we will discard the token. Otherwise, it goes into a new token list to which the Levenshtein distance will be applied. From this new list, we will take the five tokens with the highest scores generated by calculating the Levenshtein distance and normalizing between zero and one. We start with the Jaccard index because the Levenshtein distance calculation has a much slower runtime. Thus, we make our algorithm more efficient by running Levenshtein on a smaller set of tokens.

One issue with this approach is that the Jaccard index arguably serves no purpose in the algorithm. Our test results from section 7.1.3 show that the algorithm is more accurate, and faster, when it only uses the Levenshtein distance. This is surprising, since the Jaccard Index was introduced in the first place to reduce the minimize the calls to the more expensive Levensthein distance function. We will keep the initial design listed above for testing purposes, but the final version algorithm of the algorithm will not employ the Jaccard index.

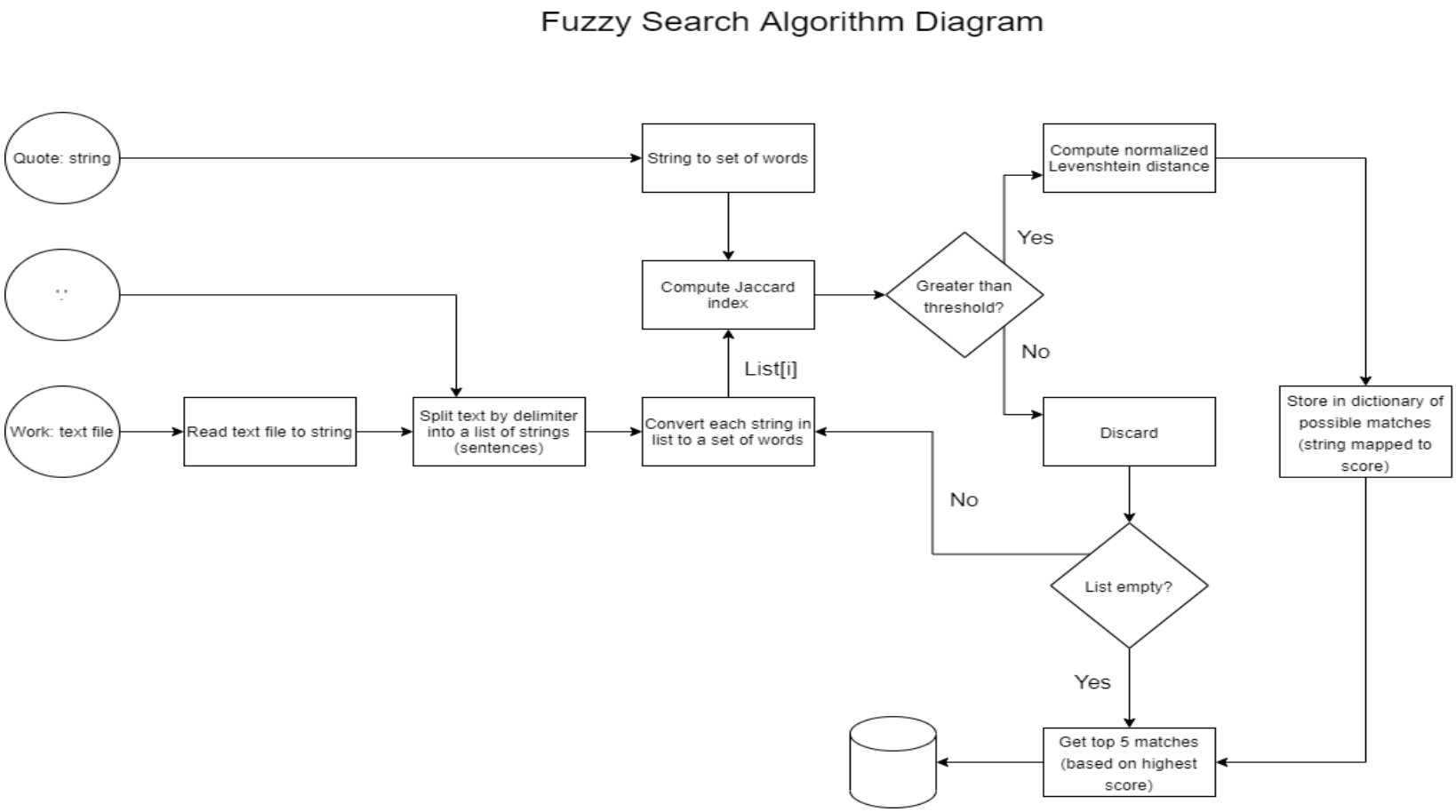


Figure 30: Logic diagram for the initial design fuzzy search algorithm

```
constant THRESHOLD = value in range [0-1]

function fuzzySearchInFileString(quote, textFileString)
 possibleMatches = {}

 quote = quote.toLowercase()
 textFileString = textFileString.toLowercase()

 quoteWordSet = stringToWordSet(quote)
 sentences = splitByPunc(textFileString)
 quotePuncSeq = getPuncSeq(quote)
 Quote = quote.removePunc()
 combinedSentences = combineByPunc(sentences, quotePuncSeq)
 combinedSentencesNew = combinedSentences.removePunc()

 for each item in combinedSentencesNew
  jacIdx = jaccardIndex(quoteWordSet, stringToWordSet(item))
  if jacIdx < THRESHOLD
   continue
  editDistance = levenshtein(quote, item)
  possibleMatches[item] = editDistance

 topFive = getTopFive(possibleMatches)
 return topFive

function newTopFive(oldTopFive, currentTopFive)
 topTen = currentTopFive.update(oldTopFive)
 topFive = getTopFive(topTen)
 return topFive

function fuzzySearchOverCorpora(quote, filePaths)
 topFive = {}

 for each fp in filePaths
  fileString = readFile(fp)
  currentTopFive = fuzzySearchInFileString(quote, fileString)
  topFive = newTopFive(topFive, currentTopFive)

 return topFive

function main(filePath)
 server.connect()
```

```
 data.read()
 topFive = {}
 quotes = readQuotesFromFile(filePath)
 filePaths = getTextFilePaths()

 for each quote in quotes
  if quote.flag == true:
   continue
  topFive = fuzzySearchOverCorpora(quote, filePaths)
  writeToDatabase(topFive)

function stringToWordSet(s)
 s = s.removePunc()
 s = s.split()
 s = set(s)
 return s

function splitByPunc(s)
 s = re.split('([.?!])', s)
 return s

function getPuncSeq(q)
 t = ""

 for c in q:
  if c == '.' or c == '?' or c == '!'
   t += c

 return t

function combineByPunc(list, seq)
 ret = []
 index = 0

 if len(seq) == 1:
  return list;

 if len(seq) == 2:
  for i in range(len(list) - 1):
   if list[i][-1] == seq[0]:
    if list[i+1][-1] == seq[1]:
     ret[index] = list[i][-1] + list[i+1][-1]
     index+=1
  else:
   ret[index] = list[i]
```

```
   index+=1

 if len(seq) == 3:
  for i in range(len(list) - 2):
   if list[i][-1] == seq[0]:
    if list[i+1][-1] == seq[1]:
     if list[i+2][-1] == seq[2]:
      ret[index] = list[i][-1] + list[i+1][-1] + list[i+2][-1]
      index+=1
   else:
    ret[index] = list[i]
    index+=1

 return ret

function getTopFive(d)
 topFive = {}

 for i in range(5):
  maxKey = max(d, key=d.get)
  maxValue = max(d.values())

  topFive[maxKey] = maxValue

  del d[maxKey]

 return topFive
```

Figure 31: Pseudocode for the initial fuzzy search algorithm and its related functions

6.3.2 Potential Pitfalls

There are some potential issues with this first approach. The most immediate concern is regarding how well the algorithm will be able to identify the source of a quote that is a paraphrase of its original source quote. We attempt to mitigate this issue via the first pass of the algorithm by assuming that paraphrased quotes will contain mostly the same words as the source quote. This should also handle the case that a quote uses synonyms or abbreviations, but only to a certain point. If our threshold value is too high, then the Jaccard measure between a heavily paraphrased quote and its source quote may be too low for the source quote to be

considered in the second pass of our algorithm. If the measure is too low, however, then we may end up checking too many quotes and incur a far greater runtime.

A similar issue arises when we examine our first method of splitting the input file into sentences. Presently, we are planning on splitting each input file by punctuation marks. However, some quotes contain multiple punctuation marks, so in this case simply splitting the input file up by punctuation may be insufficient. We may be able to address this problem by instead splitting the input file into sentences around a specific delimiter, such as the period character ('.'). All the dictionary quotes we have observed thus far contain only one period at their end, so this may be a valid option.

Improving the Runtime

A final, serious issue is with the algorithm's runtime. Although we analyzed the algorithm's time complexity in section 5.2.4, we could not be certain of the algorithm's actual runtime until we ran it for testing. Our initial tests showed that the algorithm would be much slower than we had hoped – it would have taken years for the algorithm to completely search for all the quotes over all the corpora. To solve this problem, we identified several methods of reducing the algorithm's runtime. We can use them all together to reduce the algorithm's total runtime to several weeks instead of years.

Concurrency

The easiest way to accelerate the algorithm's runtime would be to have it search for many quotes in parallel instead of searching for a single quote at a time. We can leverage the Pool class in Python's built-in multiprocessing module to easily accomplish this goal. Objects of the class expose a method called map to perform functions using multiprocessing. Map takes two arguments: a function which expects one argument, and a Python iterable. The function will then be executed once for each element in the iterable, taking the current element as input. The starmap method is the same as map, but for functions that expect multiple arguments.

We can use the starmap methods of a Pool object to reduce the algorithm's runtime by a factor equal to the number of processors on the machine running the algorithm. For instance, running the algorithm concurrently on the UCF development server, which 16 processors, would reduce its runtime by a factor of 16. This is a significant gain from adding only a few lines of code.

The only caveat with this approach is that it is impossible to log the algorithm's progress towards completion without adding some shared state to the algorithm function. To get around this, we can divide the algorithm's work into "chunks" and log out the algorithm's current progress whenever it finishes a chunk. We will generally want larger chunks as this will mean improved runtime at the cost of less frequent updates on progress.

External Servers

The next step from running the algorithm on multiple processors is to run it on multiple machines. To do this, however, we first must alter the algorithm to output its results to some intermediate format before writing to the database (this also makes the algorithm more robust in that results will not be lost if an error occurs while writing to the database). This can be easily implemented by storing resulting matches as Python NamedTuple objects and marshalling them into JSON files using Python's built-in json module. Next, we must add functionality to enable the algorithm to search over a specific subset of all the quotes. This can be implemented by allowing the user to pass a path to a JSON file containing a list of ids of quotes to search for to the algorithm. The command line argument itself can be added easily using the aforementioned begins library.

Finally, we must acquire actual machines to run the algorithm on. After conducting research on the more popular hosting providers such as AWS, DigitalOcean, Heroku, and Google Cloud Platform, we found Contabo [62]. Contabo provides virtual private servers for much cheaper monthly costs than its competitors, and its VPS XL SSD instances offer 10 virtual CPUS each. After performing some runtime calculations and discussing how much we would all be willing to financially contribute to the project, we settled on purchasing 4 Contabo VPS XL SSD instances to run the algorithm on in parallel. This gives us 40 more processors to run the algorithm on.

Quick Search Options

The corpora consist of thousands of works, so the algorithm could be made orders of magnitude faster if it were to only search for quotes within a subset of the corpora. We can safely do this by taking advantage of one key insight: most of the quotes, especially those from well-known authors, are correctly attributed. We can leverage this knowledge to first only search for a given quote over all the works by its attributed author, and check if any of the resulting matches meet some threshold value. If they do, then we can mark the quote as searched, store its matches, and never search for it again. If not, then we can add the quote's SQL record ID to a table of quote IDs corresponding to quotes that failed the "quick lookup" and need to be searched for over all the corpora.

We can use a LIKE query to search the works table for all works by a given quote's attributed author. It is better to perform a LIKE query since even if the attributed author for a quote is correct, the spelling of their name may not be. Next, we can use our testing data to determine what the threshold value should be for determining whether a quote passes or fails the quick lookup. According to figure 83, the average top match score for all the quotes that were correctly identified is about 54. We will add slight leeway to this number and use 53 as our threshold. If this proves to be far too lenient a threshold, we can change it and rerun the algorithm.

Adjustments to the Algorithm and the Text Tokenization Method

Finally, we can alter the algorithm itself to optimize its runtime. The two most significant improvements would be to remove the Jaccard Index from the algorithm, and to divide the work files into text using the "sliding window" technique outlined in section 6.10.1. We have experimented with various combinations of text tokenization string comparison methods (see figure 83), and this combination appears to be the most viable.

6.4 Corpora Scraper

6.4.1 Design Summary

The corpora scraper is a component of the system that will take a corpus URL and extract needed information to save in the database and file system. Figure 32 gives a logical overview of how the corpora scraper will go through its scraping process.

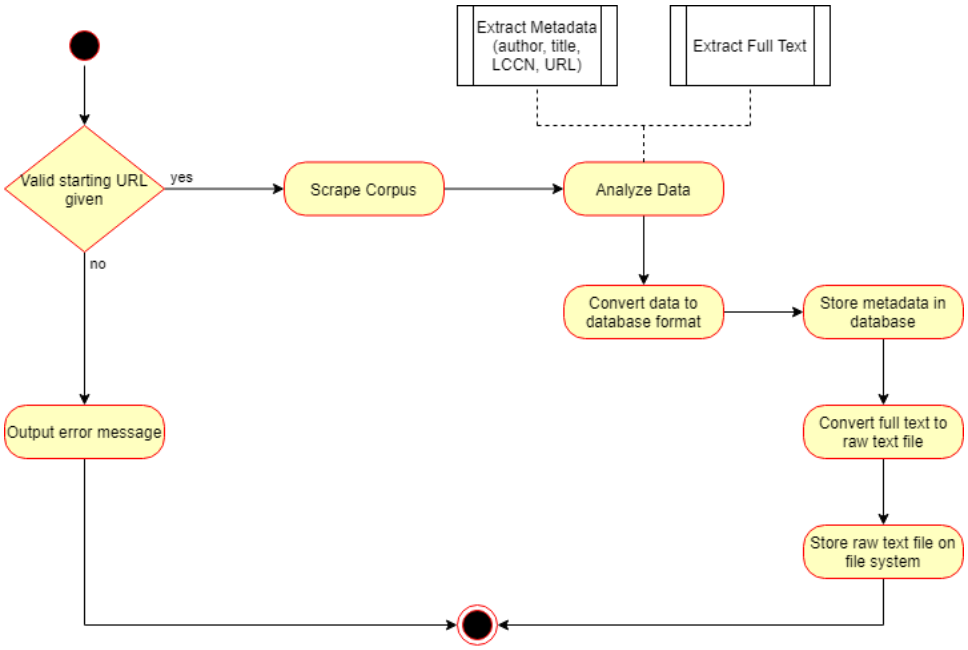


Figure 32: Corpora Scraper logic diagram.

6.4.2 Detailed Design Content

The goal of the corpora scraper is to extract metadata (i.e. the author, title, and LCCN of the text as well as the URL for each page) and raw literature text (i.e. the full written work) from a corpus given to it by the system. The URL of the corpus to be scraped will be provided to the scraper, and then the scraper will load each page of the corpus to scrape them individually.

There will be four different Python files, each containing a scraper written specifically to process one of the four corpora. As outlined in Figure 32, before each scraper begins parsing a corpus, it will check whether the given URL is from the correct corpus before trying to process it. If the URL is from a different corpus than the one that the scraper is associated with, then it cannot process it correctly, so the program will simply output an error message and exit.

Figure 33 is a state diagram outlining how the corpora scraper will traverse a given corpus and extract the metadata from the work of literature and raw literature text from each page. The states were used when implementing each scraper with each state becoming an explicit function call in the main scraper function.

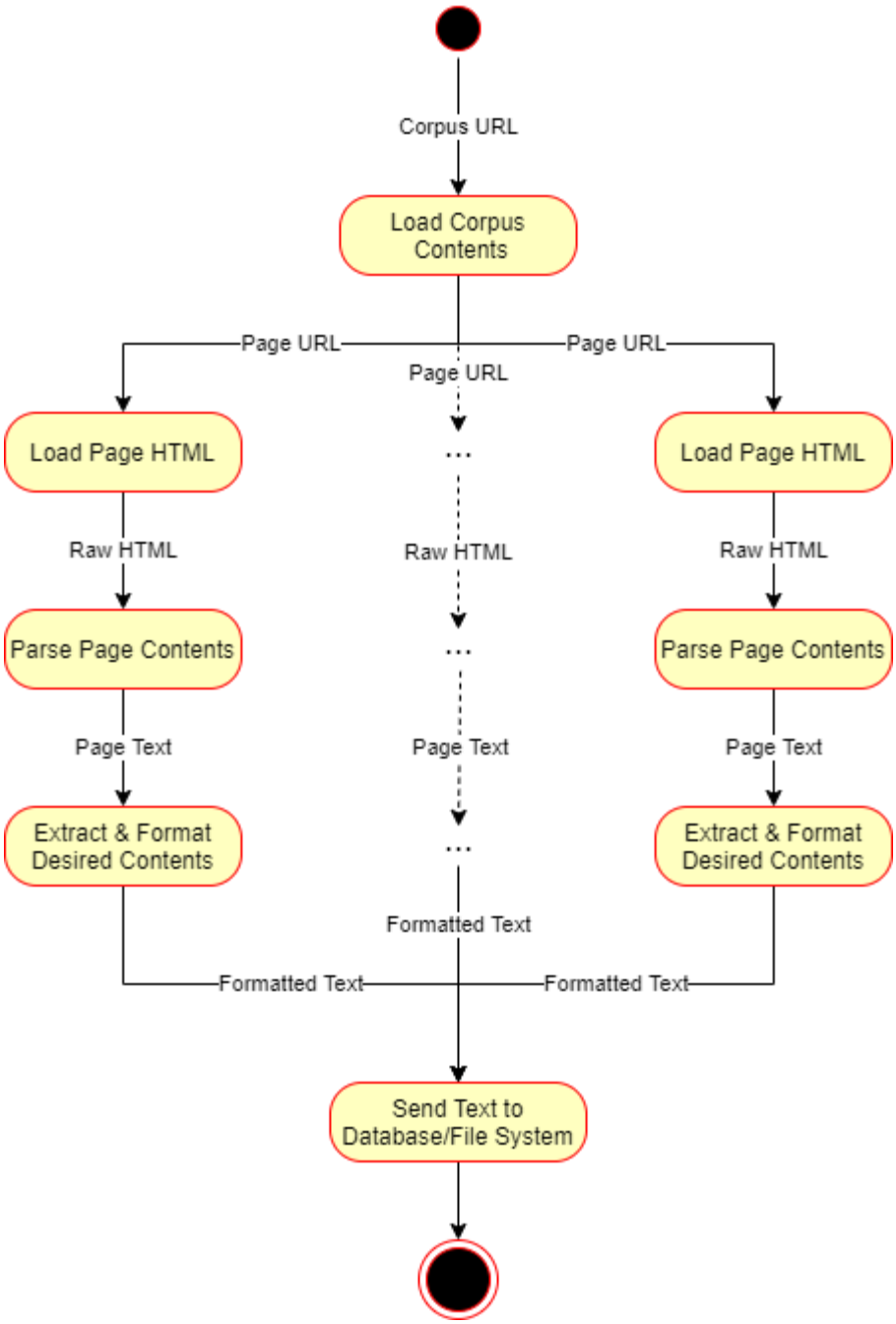


Figure 33: Corpora Scraper scraping state diagram.

Figure 34 describes how the corpora scraper will traverse a given corpus URL and extract the contents and metadata of each written work contained within the corpus.

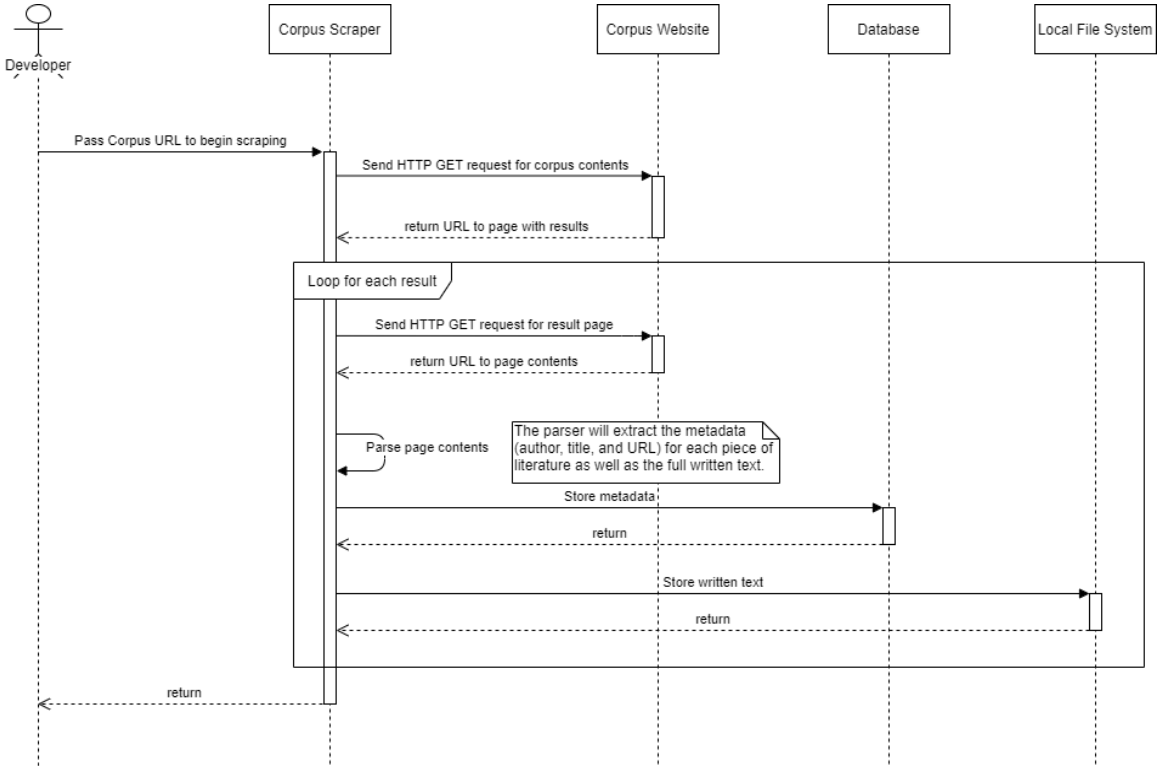


Figure 34: Corpora scraper sequence diagram.

Traditionally, a web scraper will start on the homepage of a desired website and collect links to traverse through, collecting data. Instead of doing this for the corpora scraper, we will set the starting page to be a webpage in the corpus that contains a list of pieces of literature (i.e., a page of search results). This removes the need to find where literature is stored in the corpus and limits the amount of HTTP requests we will need to make. This starting page will be loaded and then

scraped to find the links to each of the pieces of literature. Figure 35 is an example webpage from the Library of Congress containing pieces of literature.

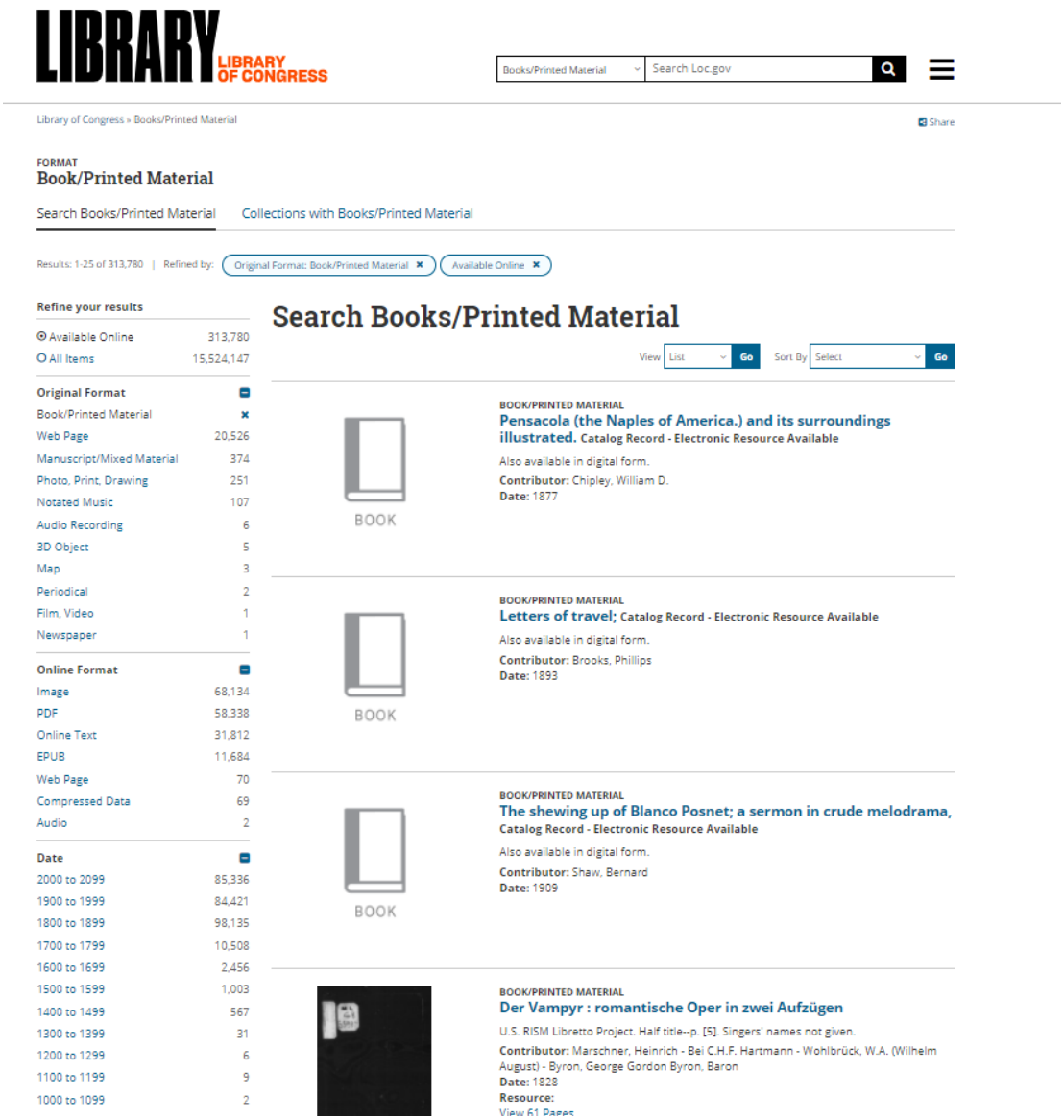


Figure 35: A page containing pieces of literature to be scraped, taken from https://www.loc.gov/books/ .

In Figure 35 there are four pieces of literature contained on the page that will be fed into the corpora scraper. This is done by first using *BeautifulSoup* to parse the page's HTML looking for each list item containing the class name "item". Once

those list items are found, the contents of the item are again parsed looking for all hyperlinks. Each list item only contains one hyperlink, which leads to the literature item's content webpage. Each link is then added to the scraper's queue of pages to continue processing. Figure 36 shows a list item and the hyperlink that leads to the content page.

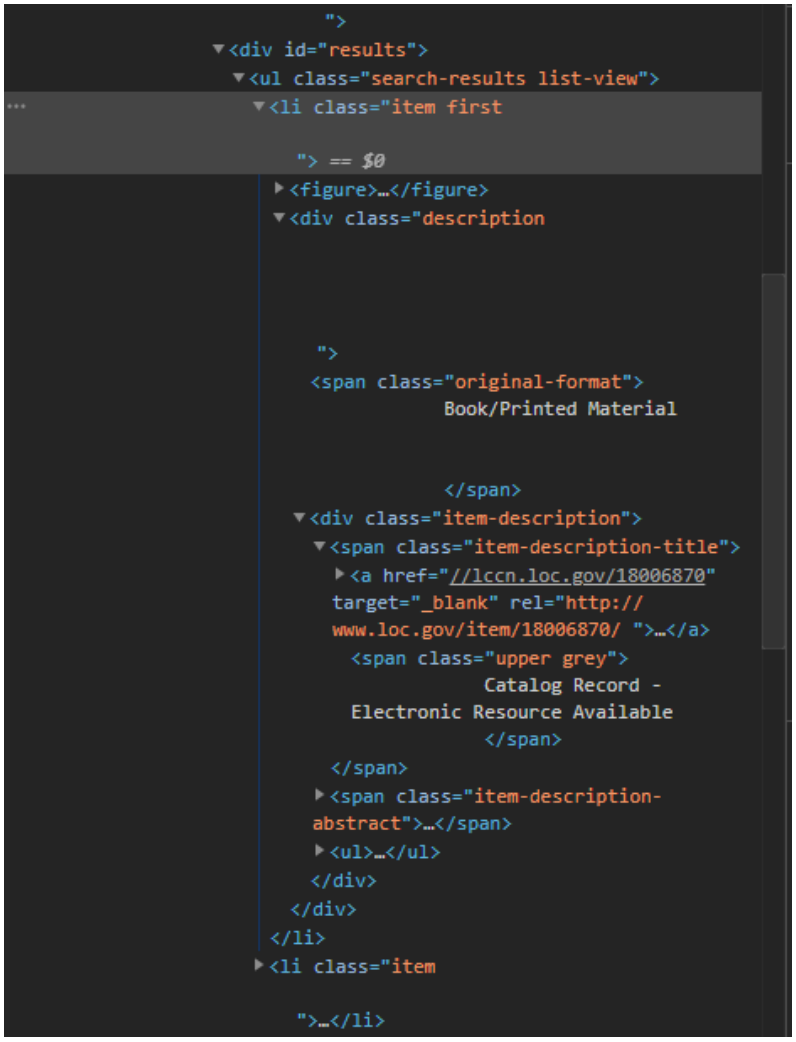


Figure 36: HTML of a list item and its associated link.

Once all the links from the starting page have been added to the queue, the scraper then starts pulling each link from the queue and processing it. This starts by loading the webpage, extracting the literature's metadata, and storing that data in the project database. Then the link to the literature's XML representation is extracted

and loaded. From this XML page (shown in Figure 37), the full text of the literature is extracted and stored on the local file system in a text file. This process is repeated until the queue is empty. At that point, the scraper will load the next page of results, the link to which is stored when the initial page is loaded.

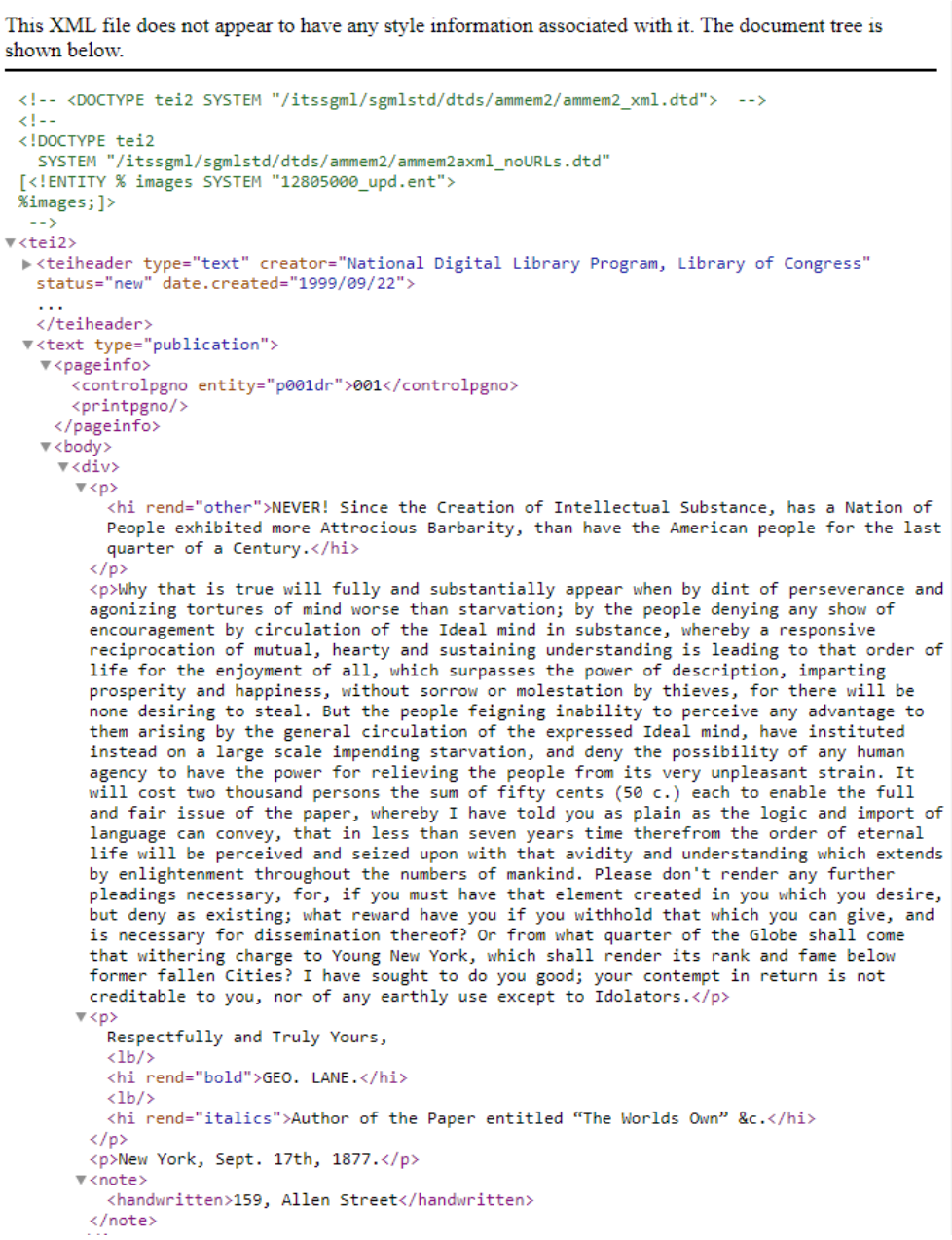


Figure 37: An example XML page highlighting how the full text of a piece of literature is extracted, taken from https://tile.loc.gov/storage-services/service/rbc/rbpe/rbpe12/rbpe128/12805000/12805000.xml.

The process for scraping a corpus is generally the same as outlined here for any corpus, but the process described here is specific to scraping literature contained in the Library of Congress corpus. Other corpora are not guaranteed to have the same HTML structure, which is why we decided to create a separate Python file to scrape each corpus. We will also need to manually examine each corpus to determine the best way to parse its HTML to extract the information we need.

Handling HTTP Requests

Due to the nature of how the corpora scraper will be processing the various corpora, it will be sending a large amount of HTTP GET requests to each of the corpora being scraped. This could cause a problem if a corpus website has protection in place against large amounts of HTTP requests coming from a single IP address in quick succession. To account for this, the scraper will catch the exception thrown by the *requests* library when an HTTP request fails, and put the URL of the page that failed into a secondary queue to be processed after the contents of the corpus has been exhausted. After a second round of processing, if that URL fails again, a log entry is generated containing the URL along with the exception thrown when trying to send a request to it. Figure 38 is a state diagram detailing how this process occurs during the normal scraping procedure.

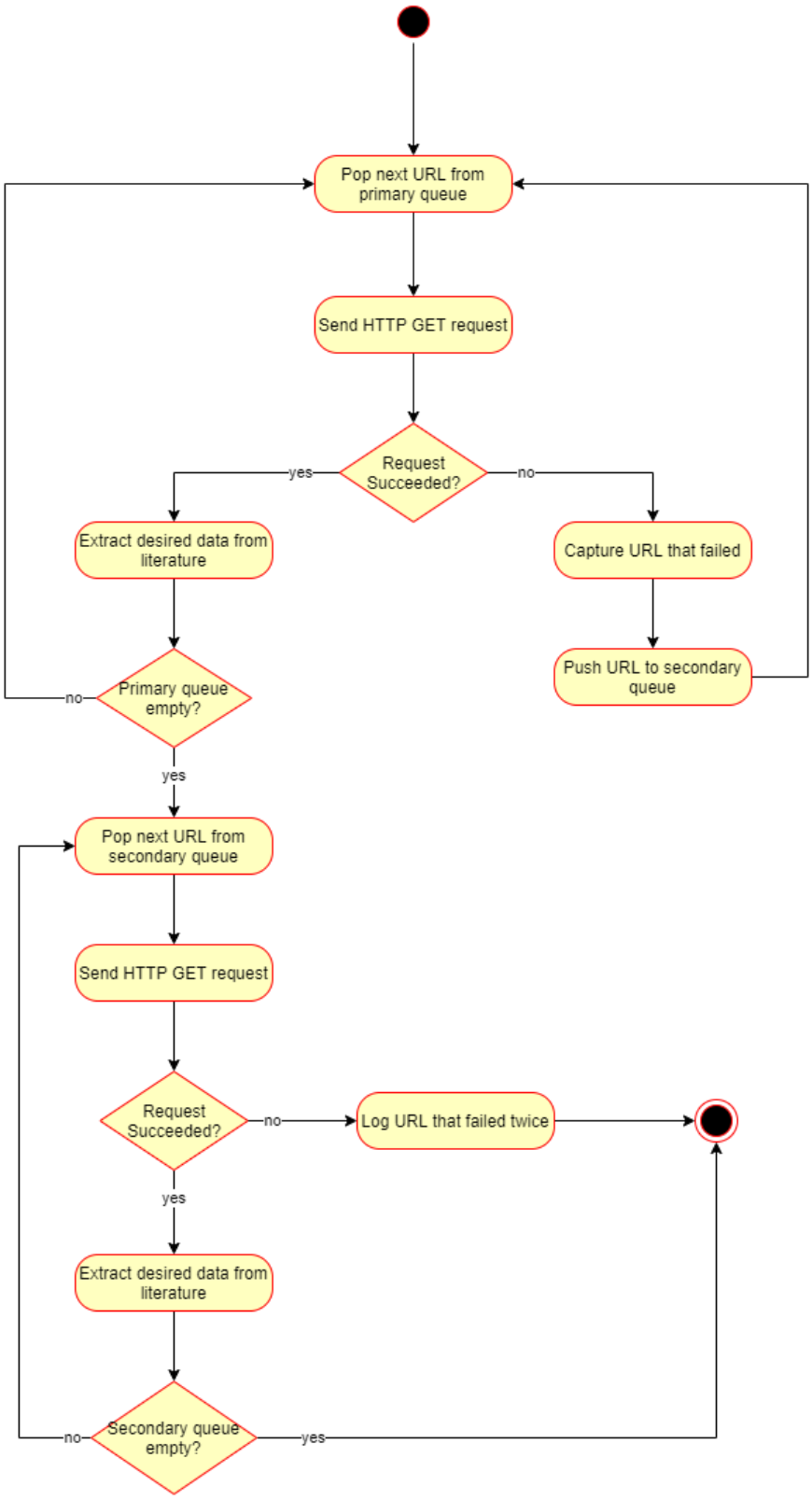


Figure 38: State diagram showing how the HTTP requests are handled, including failures.

Differences in Corpora

Naturally, each online corpus will have a different website layout, causing there to be differences in how each corpus is scraped to extract the desired information. These differences are easily noticeable when examining each website to determine which HTML elements lead to the content we will be extracting. To increase efficiency, an attempt was made to find a list of literature within each corpus that resembles the search result page in Figure 35. However, in addition to the Library of Congress, only the HathiTrust library offers a way to retrieve such a page. The other corpora will be further examined to determine how to scrape them in the most efficient way possible.

Project Gutenberg

Downloading all of the pieces of literature contained in a corpus to the local file system would be preferable to scraping the corpus website, as it will limit the amount of HTTP requests being sent to the website and potentially reduce processing times significantly. Project Gutenberg is the only corpus that offers this option, and for this corpus we will be using the locally downloaded version instead of scraping the website. First, the terminal command `wget` is executed on a special URL provided by the Project Gutenberg maintainers to begin the download, which can be seen running in Figure 39.

```
chris@ubuntu:/media/chris/large/gutenberg/download$ wget -m -H "http://www.guten
berg.org/robot/harvest?filetypes[]=html&langs[]=en"
--2020-11-17 18:17:50--  http://www.gutenberg.org/robot/harvest?filetypes[]=html
&langs[]=en
Resolving www.gutenberg.org (www.gutenberg.org)... 152.19.134.47, 2610:28:3090:3
000:0:bad:cafe:47
Connecting to www.gutenberg.org (www.gutenberg.org)|152.19.134.47|:80... connect
ed.
HTTP request sent, awaiting response... 200 OK
Length: unspecified [text/html]
Saving to: ‘www.gutenberg.org/robot/harvest?filetypes[]=html&langs[]=en’

www.gutenberg.org/r     [ <=>                ]  13.35K  --.-KB/s    in 0s

Last-modified header missing -- time-stamps turned off.
2020-11-17 18:17:50 (310 MB/s) - ‘www.gutenberg.org/robot/harvest?filetypes[]=ht
ml&langs[]=en’ saved [13673]

Loading robots.txt; please ignore errors.
--2020-11-17 18:17:50--  http://aleph.gutenberg.org/robots.txt
Resolving aleph.gutenberg.org (aleph.gutenberg.org)... 65.50.255.20, 2604:3200:0
:3:1618:77ff:fe49:8a7
Connecting to aleph.gutenberg.org (aleph.gutenberg.org)|65.50.255.20|:80... conn
ected.
```

Figure 39: `wget` shown downloading the contents of Project Gutenberg.

There are two suffixes added to the provided URL to specify what types of files should be downloaded from the corpus. The suffix `filetypes[]=html` limits the download to all literature in an HTML format. This will allow us to still use *BeautifulSoup* to process the literature contents and easily extract the desired information. The suffix `langs[]=en` ensures that all literature downloaded is written in English. Since *Johnson's Dictionary* is written in English, there is no need to download literature in any other language as no quotes will be matched in literature not written in English.

Once the download finished, a quick inspection of the downloaded files reveals that all the literature files are compressed into zip files. Due to the large volume of data, it would be impossible to extract each of the zip files by hand. Instead, a terminal command was executed to extract each zip file and copy the contents to a central folder containing each zip file's contents. The command and its output are shown in Figure 40.

```
chris@ubuntu:~/Documents/gutenberg-dl$ find . -name '*zip' -exec unzip -o {}
 -d ./extracted \;
Archive:  ./aleph.gutenberg.org/8/86/86-h.zip
  inflating: ./extracted/86-h/86-h.htm
  inflating: ./extracted/86-h/images/00-017.jpg
  inflating: ./extracted/86-h/images/00-021.jpg
  inflating: ./extracted/86-h/images/00-023.jpg
  inflating: ./extracted/86-h/images/01-025.jpg
  inflating: ./extracted/86-h/images/01-027.jpg
  inflating: ./extracted/86-h/images/01-029.jpg
  inflating: ./extracted/86-h/images/02-031.jpg
  inflating: ./extracted/86-h/images/02-033.jpg
  inflating: ./extracted/86-h/images/02-035.jpg
  inflating: ./extracted/86-h/images/02-038.jpg
  inflating: ./extracted/86-h/images/03-041.jpg
  inflating: ./extracted/86-h/images/03-043.jpg
  inflating: ./extracted/86-h/images/03-046.jpg
  inflating: ./extracted/86-h/images/03-047.jpg
```

Figure 40: Extracting the contents of the Project Gutenberg download.

The extraction command utilizes two powerful programs built into Unix systems: `find` and `unzip`. First, `find` is told to search for all files that have names ending with the suffix "zip." This will descend into the directory structure starting at the current directory and ending with every file it can find that meets the requirement. The `-exec` option is then used to execute another operation on the files found – in this case, `unzip`. In the execution of `unzip`, two options are used to determine how the program will act on each file. First, the `-o` option tells it to ignore any extracted files that have the same name. Without this option, the user would be prompted to decide between ignoring, renaming, or overwriting each encountered duplicate.

Then, the `-d ./extracted` option tells the program to store all extracted data within the local directory named “extracted.” Without this addition, the files would simply be stored in the same directory that the zip file was found, causing more work to be needed to go and find all those files again.

After the extraction finished, a quick examination of the extracted files made it clear that the majority of what was downloaded were image files. These image files are not necessary for our project, as we only need to extract data from the HTML files containing a raw text version of each piece of literature. To increase efficiency when processing these files later, another terminal command was executed to copy each HTML file into one local directory. This works similarly to the command executed in Figure 40 but using the `cp` program instead of `unzip`.

Once the files are organized into a centralized directory, extracting the contents of each piece of literature becomes a simple process. Each HTML file will be opened like a normal text file and read into a buffer. That buffer will then be fed into a *BeautifulSoup* object to be parsed like it was a live web page. Since each HTML file was not checked during extraction for the publication date of its written text, this will need to be done during the parsing process. Each HTML file has the same layout, containing a section at the start of the HTML block of the full written text that lists important metadata about the piece of literature. The layout of this section can be seen in Figure 41.

```
35  <pre xml:space="preserve">
36
37  The Project Gutenberg EBook of Anne Of The Island, by Lucy Maud Montgomery
38
39  This eBook is for the use of anyone anywhere at no cost and with
40  almost no restrictions whatsoever.  You may copy it, give it away or
41  re-use it under the terms of the Project Gutenberg License included
42  with this eBook or online at www.gutenberg.org
43
44
45  Title: Anne Of The Island
46
47  Author: Lucy Maud Montgomery
48
49  Release Date: March 7, 2006 [EBook #51]
50  Last Updated: October 6, 2016
51
52  Language: English
53
54  Character set encoding: UTF-8
55
56  *** START OF THIS PROJECT GUTENBERG EBOOK ANNE OF THE ISLAND ***
57
58
59
60
61  Produced by Charles Keller and David Widger
62
```

Figure 41: How the metadata is contained in each HTML file.

As seen in Figure 41, the author, title, and publication date (listed as “Release Date” in this example) are all contained in a `<pre>` block. This localization under one tag allows the *BeautifulSoup* object to trivially extract the metadata, and this extraction process can then be directly replicated for each piece of literature. The publication date of each text will then be checked, and if it is in the range that we are looking for, its full text will be extracted and stored along with the metadata. Once the parsing process is complete, the buffer is emptied, the HTML file is closed, and the metadata and full text are stored in the same way as they are stored for the literature processed from each of the other corpora.

Liberty Fund

The Online Library of Liberty portion of the Liberty Fund corpus offers a list of texts similar to that which the Library of Congress offers, but the main distinction is how the web page is structured, which can be seen in Figure 42. Instead of being able to sort the texts by a user-decided field, the texts are simply listed based on the letter that the title of each text starts with. This difference requires us to manually check the publication date of each piece of literature before sending another HTTP request to open the text’s web page. Once that page is processed, the scraper can then move on to the next page for the next letter in the alphabet. This significantly reduces the amount of HTTP requests we will need to send, as there will only be a total of 26 pages of lists of text, and it is likely that not all texts on each page will be from the time period we are checking for.

Titles that begin with "A"

a b c d e f g h i j k l m n o p q r s t u v w x y z

Title	Author	Published	Edition	Type
The A B C of Finance	Simon Newcomb	1877	1877	Book
A Concise History of the Common Law	Theodore Frank Thomas Plucknett	1956	2010	Book
A Treatise on Metallic and Paper Money and Banks (1858)	John Ramsay McCulloch	1858	1858	Encyclopedia Article
A Treatise on Political Economy (LF ed.)	Antoine Louis Claude, Comte Destutt de Tracy	1817	2011	Book
An Account of Denmark, With Francogallia and Some Considerations for the Promoting of Agriculture and Employing the Poor	Robert Molesworth	2011	2011	Collection
Acton-Creighton Correspondence	John Emerich Edward Dalberg, Lord Acton	1887	1887	Letter
Adam Smith and the Wealth of Nations (DVD)	Adam Smith	2005	2005	Audio
Address of the Free Constitutionalists to the People of the United States (1860)	Lysander Spooner	1860	1860	Book
On the Admission of Women to the Rights of Citizenship	Marie-Jean-Antoine-Nicolas Caritat, Marquis de Condorcet	1790	1912	Book
The Advancement of Learning	Sir Francis Bacon	1605	1901	Book
The Aeneid (Dryden trans.)	Virgil	1697	1909	Book
Aida by Antonio Ghislanzoni, music by Giuseppe Verdi	Giuseppe Verdi	1871	1911	Opera
The Alchemy of Happiness	Al Ghazali	1873	1873	Book
The American Commonwealth, 2 vols.	Viscount James Bryce	1888	1995	Set
The American Nation: Primary Sources	Bruce Frohnen	2008	2008	Collection
American Political Writing During the Founding Era: 1760-1805, 2 vols.	Charles S. Hyneman	1983	1983	Set
The American Republic: Primary Sources	Bruce Frohnen	2002	2002	Collection

Figure 42: Example web page of texts from the Online Library of Liberty, taken from https://oll.libertyfund.org/titles.

Archive.org

Initially, Archive.org was thought to be an invaluable source of literature for this project, as it contains millions of pieces of media to sort through. However, upon further inspection, this corpus does not contain any media published before the year 1800. Since *Johnson's Dictionary* was published in 1755, it would be impossible to find any texts within Archive.org that could contain a match to any of the quotes in the dictionary. Therefore, while this corpus is an invaluable resource

to anyone interested in the time period from 1800 to present, we sadly will not be able to utilize its contents for the purposes of this project.

Reducing Number of Extracted Texts

In order to reduce the runtime of the fuzzy search algorithm, we will need to ensure that the number of texts it has to search through is as small as possible. We will be doing this by removing unnecessary texts, such as duplicates between corpora, texts that have no author associated with them, and texts that were published outside of the desired time period. While it would be best to simply check the publication date of each text before the scraper saves it, the Project Gutenberg and Liberty Fund corpora do not provide this ability.

Removing duplicate texts and texts with no author is a trivial task, but verifying the publication date of texts that are not provided with in in their online counterparts is more complex. To accomplish this, we first decided to utilize an API that Wikipedia offers which allows users to send lookup queries and get information back. We sent a query to this API containing the authors of each text, and from there we extracted the birth date of the author and removed the texts of authors who were born after 1755. If the author could not be found using this API, we instead searched for the text within the HathiTrust library and checked the publication date, removing texts that were published after 1755. To ensure this process ran as efficiently as possible, we only verified the texts that came from Project Gutenberg and Liberty Fund.

Writing Metadata to the Database

Since we will be using the *mysql-connector-python* library to add entries in our database, we can add literature metadata to the metadata table in our database by simply executing SQL code from within our Python code. To show a proof of concept, a test user and database was created to exemplify how we will be taking extracted literature metadata and writing it to the production database. Figure 43 shows the connection to the test database being made from within the scraper code.

```
56  # Connect to SQL database
57  db_conn: mysql.connector.MySQLConnection = mysql.connector.connect(
58      user=os.environ.get('DB_USER'),
59      password=os.environ.get('DB_PASS'),
60      host=os.environ.get('DB_IP'),
61      database=os.environ.get('DB_DB'))
62
```

Figure 43: The connection to the test database being made in Python.

Once the connection is established, the `db_cursor` object shown in Figure 43 is then used along with a pre-defined string to write entries into the database. Figure 44 shows how this is done with the *mysql-connector-python* function calls.

```
69      sql_insert_stmt: str = (
70          "INSERT INTO work_metadata(title, author, url, filepath, lccn)"
71          "VALUES (%s, %s, %s, %s, %s)" )
 .                          .
 .                          .
 .                          .
113     # Put the metadata in the database
114     try:
115         data: tuple = (page[1], page[2], page[0], filepath, '-1')
116         db_cursor.execute(sql_insert_stmt, data)
117         db_conn.commit()
118     except Exception as err:
119         db_conn.rollback()
120         logger.warning("Error occurred when writing to database", exc_
```

Figure 44: Using the *mysql-connector-python* library to write to the test database.

In Figure 44, the `sql_insert_stmt` string is the actual SQL code that will be executed. Before it is that happens, the call to `db_cursor.execute` replaces each `%s` seen in `sql_insert_stmt` with the title, author(s), text URL, and file path of the full written text, respective to their arrangement in the `data` tuple. The call to `db_conn.commit` then finalizes the query and attempts to write to the database. If the write was unsuccessful, an exception is thrown, and the call to `db_conn.rollback` reverts any changes that may have occurred before an error was caught. Figure 45 shows the first two entries in the database, after all of the scrapers had been run.

```
mysql> select * from work_metadata\G
*************************** 1. row ***************************
      id: 47359
   title: An historical essay concerning wichcraft. With observations upon matte
rs of fact; tending to clear the texts of the sacred Scriptures, and confute the
 vulgar errors about that...
  author: Hutchinson, Francis, 1661-1739.
     url: https://catalog.hathitrust.org/Record/000584513?filter%5B%5D=publishDa
teTrie%3A%5B%2A%20TO%201755%5D&fqor-language%5B%5D=English&fqor-language%5B%5D=E
nglish%2C%20Middle%20%281100-1500%29&fqor-language%5B%5D=English%2C%20Old%20%28c
a.%20450-1100%29&fqor-format%5B%5D=Book&ft=ft
filepath: hat_texts/hat1An-hi.txt
    lccn: -1
*************************** 2. row ***************************
      id: 47360
   title: Evphrates, or, The waters of the east : being a short discourse of tha
t secret fountain, whose water flows from fire, and carries in it the beams of t
he sun and moon / by...
  author: Vaughan, Thomas, 1621-1666.
     url: https://catalog.hathitrust.org/Record/000584899?filter%5B%5D=publishDa
teTrie%3A%5B%2A%20TO%201755%5D&fqor-language%5B%5D=English&fqor-language%5B%5D=E
nglish%2C%20Middle%20%281100-1500%29&fqor-language%5B%5D=English%2C%20Old%20%28c
a.%20450-1100%29&fqor-format%5B%5D=Book&ft=ft
filepath: hat_texts/hat2Evphr.txt
```

Figure 45: Visualization of the metadata stored in the test database.

Output Logging

As mentioned previously, this project will employ the Python *logging* library to manage error logging to log files stored on the local file system. The name of each log file will be of the format `log<current_date>.log`. Each log entry will begin with a time stamp showing when the entry was written, followed by the log level, what file created the entry, and the information of the log entry. Figure 46 shows how the *logging* library is used in the corpora scraper code.

```
165     # Go through the queue of results, first extracting their metadata then the full text
166     while len(q) > 0:
167         litpage: str = q.popleft()
168 
169         # Encapsulate this call in a try block to capture all exceptions thrown because of HTTP req
170         try:
171             processPage(litpage, fileindex)
172         except Exception as err:
173             # With an error found, put the page that caused it into the failure queue so it can be
174             fail_q.append(litpage)
175             logger.warning(f'Processing of {litpage} failed, adding to fail queue', exc_info=True)
176 
177         fileindex = fileindex + 1
```

Figure 46: Example usage of the *logging* library.

In the example shown in Figure 46, the call to `logger.warning` outputs a general message explaining what happened, and due to the added parameter `exc_info=True`, the stack trace of the exception raised as well. Figure 47 shows

the contents of a log file that contain an error message from this function call. The `logger` object was previously configured to choose between the level of log entries that get written to a file versus printed to the terminal window. In this component of the project, each log entry will be printed to both the terminal window and the log file.

log2020-11-15.log

```
2020-11-15 11:24:33 : loc_scraper : WARNING -- Processing of https://www.loc.gov/item/
2011656481/ failed, adding to fail queue
Traceback (most recent call last):
  File "scraper.py", line 149, in scrape
    processPage(litpage, fileindex)
  File "scraper.py", line 87, in processPage
    page = requests.get(fulltext_link)
  File "/usr/lib/python3/dist-packages/requests/api.py", line 72, in get
    return request('get', url, params=params, **kwargs)
  File "/usr/lib/python3/dist-packages/requests/api.py", line 58, in request
    return session.request(method=method, url=url, **kwargs)
  File "/usr/lib/python3/dist-packages/requests/sessions.py", line 506, in request
    prep = self.prepare_request(req)
  File "/usr/lib/python3/dist-packages/requests/sessions.py", line 449, in
prepare_request
    hooks=merge_hooks(request.hooks, self.hooks),
  File "/usr/lib/python3/dist-packages/requests/models.py", line 305, in prepare
    self.prepare_url(url, params)
  File "/usr/lib/python3/dist-packages/requests/models.py", line 379, in prepare_url
    raise MissingSchema(error)
requests.exceptions.MissingSchema: Invalid URL 'page level only': No schema supplied.
```

Figure 47: Log file contents showing the output of the example from Figure 46.

6.5 Exporting the Sponsor-provided Excel File to JSON

6.5.1 Design Summary

The sponsor initially provided the full list of the dictionary's quotes across all available editions (1 and 4) in an Excel file. To pass the quotes to the fuzzy search algorithm, they first had to be converted into a more usable file format.

	A	B	C	D	E	F	G	H	I	J	K
1	HEAD	EDITION	POS	DEFINITION	QUOTE	TITLE	AUTHOR	BIBLSCOPE			
2	A	1		The first letter of the Europear	A hunting Chloë went.		Prior.				
3	A				They go a begging to a bankrupt's door.		Dryd.				
4	A				May pure contents for ever pitch their tents Upon these downs, the		Wotton.				
5	A				Now the men fell a rubbing of armour, which a great while had lain		Wotton.				
6	A				Another falls a ringing a Pescennius f on medals.		Addison				
7	A			A has a peculiar signification, (	The river Inn, that had been hitherto on Italy.		Addison				
8	A				For cloves and nutmegs to the line- a, And even for oranges to Chi		Dryden.				
9	A			A, in composition, seems to h	If this, which he avouches, does appe	Macbeth.		Shakespeare's			
10	A				And now a breeze from shore began	Ceyx and Alcyone.		Dryden's			
11	A			A, in abbreviations, stands for	artium, or arts; as, A.B. bachelor of arts, artium baccalaureus; A.M. master of arts, artium ma gister; or, anno; as, A.D. anno domini.						
12	A	4		The first letter of the Europear	A hunting Chloë went.		Prior .				
13	A				They go a begging to a bankrupt's door.		Dryden .				

Figure 48: The first 13 rows of the Excel file containing all the dictionary's quotes, as initially provided by the project sponsor.

After some discussion, we decided on converting the quote data to a JSON file format. We chose this format for a few reasons. First, because it allows us to map each headword in the dictionary to an array of quote objects associated with them. This in turn enables us to much more quickly determine which quotes are the same across multiple editions of the dictionary, so that we may avoid processing them twice (as one of the project specifications is to only list one edition of a quote that does not change across different editions of the dictionary). The second reason is that Python has a built-in library for importing and exporting data to and from JSON files, facilitating the conversion process. These two factors made converting the Excel file to JSON an appealing option.

```
{
  headword1: {
      "edition": number (not null),
      "definition": string (not null),
      "quote": string (not null),
      "title": string,
      "author": string,
      "flag": boolean
  },
  headword2: {...},
  ...
}
```

Figure 49: Structure of the quotes JSON file. This structure facilitates rapid quote lookup based on headword.

The first step of the conversion process is to read the quotes Excel file into a Python object. This can be done easily using Pandas data analysis library for Python [50], which provides a function named *read_excel* to convert a given Excel file to a Pandas DataFrame object [51]. Next, the object's *itertuples* method [52] is called so that each row of the Excel file may be iterated over as a Python namedtuple object. Each of these namedtuples are then converted to a Python dictionary containing the quote's edition number, definition, and quote text, as well as a flag value stating whether the algorithm should search for the quote. The flag value of each quote is initially set to false when the quotes are first read from the Excel file, and are later set to True if it is determined that the quote should not be searched for. Once the dictionary for the quote is made, it is added to a list of

dictionaries associated with the quote's headword in another Python dictionary. Finally, this dictionary of headwords to list-of-quote-dictionary mappings is exported to a JSON file which is pre-formatted for easy readability.

6.5.2 Caveat: Inconsistent Excel File Formatting

One minor issue with the way in which the Excel file was formatted somewhat hinders the implementation of this process. To save time, the sponsor's assistants who originally compiled the document tried to avoid re-entering duplicate information whenever possible. Accordingly, the edition number, definition, and quote text of a quote is left blank whenever the value is the same as that of the preceding quote. When the Excel file is first converted into a DataFrame object, these blank cells are recorded as NaN (not a number) values in Python. Thankfully, these blank values are easily rectified by a small amount of logic. First, we use a Python dictionary to keep track of the most recently found non-empty edition number, definition, and quote text. Each time a new quote is iterated over, we then check if its edition number, definition, or quote text is empty. We can do this by using Pandas' *isnull* method, which returns True when the value passed to it is a value that Pandas treats as a "missing" value (e.g. NaN, None, and NaT) [53]. If any of these fields are empty, then we simply replace that field with the most recently found non-empty value before creating the dictionary object for the quote. If any of these fields are non-empty, then we update the variable storing the most recently found non-empty value to store the current field's value.

A final caveat that we must be aware of when converting the Excel file of quotes to a JSON file is ensuring that the encoding of the characters does not change. The given Excel file contains some non-ascii characters, and requires UTF-8 encoding, so the output file must be written using that same encoding as well. This can be achieved by passing the string "utf-8" as the value of the *encoding* keyword argument to Python's built-in function *open* when opening the output file for writing [54]. Additionally, the *ensure_ascii* keyword argument of the *json.dump* method must be set to False to ensure that the non-ascii characters in the program's output will not be escaped before being written to the output file [55].

```python
def excel_to_df(fn: str) -> pd.DataFrame:
    '''
    Reads the quote excel file and converts it to
    a Pandas DataFrame
    '''
    columns = ["HEAD", "EDITION", "POS", "DEFINITION",
               "QUOTE", "TITLE", "AUTHOR", "BIBLSCOPE"]
    data = pd.read_excel(fn)
    return pd.DataFrame(data, columns=columns)
```

Figure 50: Python code to convert the Excel file containing the full list of quotes to a Pandas DataFrame object.

```python
def quote_to_dict(edition: int, definition: str, quote: str, title: str, author: str):
    '''
    Converts a quote with the given information into a
    Python dictionary object
    '''
    return {
        "edition": edition,
        "definition": definition,
        "quote": quote,
        "title": title,
        "author": author,
        "flag": False
    }
```

Figure 51: Python code to convert a quote with the given information into a Python dictionary object.

```python
def df_to_dict(df: pd.DataFrame) -> dict:
    '''
    Converts the Pandas DataFrame of the quote excel file
    to a Python dictionary object
    Prequisites:
        - The excel file must be formatted correctly
        - The headword of the first row in the file must not be empty
        - The edition number of the first row in the file must not be empty
        - The definition of the first row in the file must not be empty
        - The quote text of the first row in the file must must not be empty
    '''
    repeat_columns = ["HEAD", "EDITION", "DEFINITION", "QUOTE"]
    non_repeat_columns = ["TITLE", "POS", "AUTHOR", "BIBLSCOPE"]
    result = dict()
    metadata_for_this_quote = {heading: "" for heading in repeat_columns}
    for tup in df.itertuples():
        if tup.HEAD not in result:
            result[tup.HEAD] = []

        for heading in repeat_columns:
            current_value = getattr(tup, heading)
            if not pd.isnull(current_value):
                metadata_for_this_quote[heading] = current_value

        for heading in non_repeat_columns:
            current_value = getattr(tup, heading)
            metadata_for_this_quote[heading] = "" if pd.isnull(
                current_value) else current_value

        metadata_for_this_quote["EDITION"] = int(
            metadata_for_this_quote["EDITION"])
        to_add = quote_to_dict(edition=metadata_for_this_quote["EDITION"],
                               definition=metadata_for_this_quote["DEFINITION"],
                               quote=metadata_for_this_quote["QUOTE"],
                               title=metadata_for_this_quote["TITLE"],
                               author=metadata_for_this_quote["AUTHOR"])

        result[tup.HEAD].append(to_add)
    return result
```

Figure 52: Python code to convert the Pandas DataFrame containing the information of all the quotes contained in the original Excel file into a Python dictionary object.

```python
def main():
    fn = r"FullQuotes.xlsx"
    df = excel_to_df(fn)
    result = df_to_dict(df)
    with open("quotes.json", "w", encoding="utf-8") as fp:
        json.dump(result, fp, indent=4, ensure_ascii=False)
```

Figure 53: The main method of the program to convert the Excel file containing the full list of quotes to a JSON file. Notice that the *encoding* argument is set to "utf-8" in the call to `open`, and that the *ensure_ascii* argument is set to False in the call to `json.dump`.

6.6 Handling Editions

6.6.1 Design Summary

The quotation data we are processing has been extracted and combined from both the first and fourth editions of the dictionary. For many of the headwords, the same quote can be repeated verbatim across both editions. Other times, the quote can reappear in the later edition with subtle differences (as little as a one-word difference). And finally, entirely new quotes can appear in the later edition.

If a quote is repeated identically across both editions, we will ignore the second appearance and process it only once. We do not actually care which edition a quote comes from for the purposes of the algorithm, just that we process every sufficiently distinct quote and do not process quotations more than once.

The most important case is when a quote reappears in a later edition with minimal changes. This happens quite often, so it is important that we eliminate as many duplicates as we can, so that the user is not wasting time by evaluating the same quotes twice. We need to create a way to determine whether the quote is similar enough to be considered identical and discarded, or different enough to be processed. To do this, we will basically use a contained version of the general algorithm to compare the similarity of the quotes. If the calculated value exceeds a certain threshold (meaning they are too similar), we will discard the repeated quote. Otherwise, we will retain the second quote and continue to process it as normal.

Because it is impossible to know beforehand which quotes might be identical or very similar, we will have to compare all combinations of quotes between the two editions. The presence of the editions also has different implications for the other components of the project. For the database, it only means that each quote will need to have an edition attribute associated with it. For the user interface, it means displaying the edition associated with each quote. A user should also be informed if a quote has a discarded sibling.

6.7 Dependency Management

6.7.1 Design Summary

Our team will be using the pipenv [56] Python module to track the dependencies of the project components written in Python. This tool is more accurate and efficient than tracking dependencies manually would be and offers several features that would make it more ergonomic for future teams to extend this project's functionality in the future.

6.7.2 Motivation

Once our team finishes working on the project, the sponsor may want to view the source code themselves or pass it on to a future senior design team for them to improve upon it. As such, it is critical that we maintain an accurate list of our project dependencies at all times and provide clear instructions on how to install those dependencies. Thankfully, there exist several tools which make both these tasks easy to perform.

6.7.3 Analysis of Available Dependency Management Tools

The most popular package management tool for Python is PIP; it is the package installer recommended by the Python Software Foundation [57]. PIP makes it almost trivial to install a package from the bevy of packages available from PyPi, the Python Package Index [58]. Once PIP is installed, one can install a new package by entering the command *pip install*, followed by the name of the package they would like to install, at their terminal. The downside to this method is that it attempts to install the entered package system-wide, which is usually prohibited when working on a development server. This issue can be resolved by passing the --user flag to the install command. This will indicate to PIP that the package should be installed in the current user's home directory, which should avoid any permission issues.

PIP greatly facilitates dependency installation but does not help us maintain a list of installed dependencies. We can leverage another package, *pipenv* [56], to help us achieve this goal. Pipenv simplifies the process of creating virtual environments for Python projects. A virtual environment is essentially "a self-contained directory

tree that contains a Python installation for a particular version of Python, plus a number of additional packages" [59]. Once a virtual environment has been created, pipenv can then be utilized to install more packages for use in that environment, instead of installing them system-wide or to a user's home directory. Moreover, pipenv maintains a list of all packages installed in the environment for each project in a special file called a *pipfile*, which pipenv automatically creates when a new virtual environment is made for a project [56]. If another developer would like to contribute to a project but does not have all the necessary dependencies, all they need to get started is pipenv and the project's pipfile. From there they can open a terminal, navigate to the directory that the project's pipfile is located in, and enter the command *pipenv install*. Pipenv will then create a new virtual environment for the project and install all the project's dependencies as listed in the pipfile. Ultimately, pipenv provides an easy means of installing packages, maintaining lists of project dependencies, and preventing issues that would arise from having multiple packages installed system-wide, by using virtual environments. Because of these reasons, we will be using Pipenv for managing our project dependencies.

6.8 Algorithm Runtime Details

6.8.1 Design Summary

One of the most important decisions we had to make was about *when* the algorithm would run. Initially, we had decided on letting the user run the algorithm on a specific set of quotes they would provide (through the frontend user interface) and allowing them to choose what corpora to use for comparison. However, we ran into a few problems with this approach. Firstly, depending on how many quotes are provided and how many corpora are selected, the algorithm could take hours to finish running. This is what initially led us to the idea of processing all the quotes first. In addition, since all the quotes need to be processed anyway, why make people wait for their quotes to finish processing? Why not just do it all beforehand? And finally, by running the algorithm first, it does not even have to run on the server, which means we do not have to store the written works on the server but can instead store them on the local machine that will run the algorithm. This would give us much more storage space to work with and reduce the strain on the server. All these details made us realize that running the algorithm *before* the user interface enters the picture is the best approach, even if we still store the corpora contents on the server.

6.9 Anonymous Quotes

6.9.1 Design Summary

Many of the quotes in the dictionary are attributed to an author or written work, but many are simply labelled as anonymous. For the authors and works we know are included in the dictionary, we will make sure to include corpora that contain these authors and their works. However, for the anonymous quotes, we have no indication of what sort of work they came from, or who wrote them. This means we must get somewhat lucky and hope that the corpora we use contains the quote somewhere.

Ideally, we would be able to use as many corpora as we want, but certain facts impose limitations on our ability to use many different corpora:

1. A different scraper must be written for each corpus we use. It must be tailored specifically to the format and content of each individual corpus to work and extract the right information.

2. The more corpora we extract from, the more storage is required. It is also very likely that we maintain duplicate records of many written works that are contained in more than one corpus, which puts extra strain on both the filesystem and the database.

3. The more corpora our algorithm must look through, the longer it will take to run.

Thus, we will attempt to strike a balance between the number of corpora and diversification of content, to best match as many of the anonymous quotes as we can.

6.10 Text Tokenization

6.10.1 Design Summary

There are two primary challenges in the development and implementation of the fuzzy search algorithm:

1. How do we break the written works into logical chunks (tokens) for comparison?

2. What is the best way to compare each chunk with a quote, and what metric(s) should be used for evaluation and scoring?

Here, we will focus on the first challenge. As a note, we store the written works in the server filesystem (as opposed to the database) to take advantage of the efficiency of the filesystem and avoid putting extra stress on the database by overloading it with calls. In addition, the ".txt" file extension was chosen because of its widespread compatibility and the fact that it is much easier to extract data from simple text files than any proprietary format.

A brief analysis of the quotes in the dictionary showed us that every quote ends its sentences in one of three punctuation marks: period, question mark, and exclamation point. Ideally, each chunk would be its own potential quote that can be directly compared to the original (in other words, if the correct match exists in the document, we want to ensure that it ends up in its own token). Since the sentence is the most basic logical structure in the written works, we decided to tokenize based on sentences. To make this more accurate, we decided to follow the punctuation marks of the original quote. We wanted to ensure that the order of sentence-ending punctuation in each token matches the original quote wherever possible. This method also prevents a sentence from being split into two different tokens, something that should never happen.

So, the general process would start with tokenizing the text file into individual sentences (where the delimiters would be the period, question mark, and exclamation point). Then, we go through the list of tokens and combine adjacent sentences that match the sentence-ending punctuation order of the original quote (e.g., if the quote has three sentences, we look at each group of three adjacent sentences in the list).

The significance of this method is that it gives us the greatest guarantee that one of the tokens we generate will contain a match for the original quote. By matching the number of sentences, as well as the specific sequence of sentence-ending punctuation, we greatly increase the likelihood of getting the right match into its own token.

The downside to this method is that it has a significant runtime overhead, as our charts in section 4 illustrate. A more performant, though less sophisticated, text segmentation technique would be to break the text up into equally sized substrings for comparison against the original quote. The substring size would be equal to the length of each quote. The pitfall to this approach is that it is highly likely to split the matching quote up across multiple segments, making it less likely that the algorithm will correctly identify it. For example, imagine one were searching for the quote “Behemoth biggest born” within this excerpt of John Milton’s *Paradise Lost* [42]:

> Bore up his branching head: scarse from his mould
>
> Behemoth biggest born of Earth upheav’d

The text would be segmented into the following substrings:

> “Bore up his branching”, “ head: scarse from hi”, “s mould Behemoth big”,
>
> “gest born of Earth up”, “heav’d”

Here, the matching quote has been split into two chunks: “s mould Behemoth big” and “gest born of Earth up”

A slightly slower, but much more reliable, version of this strategy is a “sliding window” technique. This technique introduces a new parameter called a “slide distance”, which determines how far to advance the start index of the next substring to extract from the source text. For instance, with a slide distance of 10 characters and a window size (substring size) equal to the length of the qoute being searched for, the above excerpt would instead produce these substrings:

> “Bore up his branching”, “s branching head: sca”, “g head: scarse from h”, “arse from his mould B”, “his mould Behemoth bi”, “Behemoth biggest born”, “iggest born of Earth”, “n of Earth upheav’d”, “upheav’d”

In this example, the resulting list of substrings includes the entire quote. While this may have been a rather serendipitous example, it remains a valid demonstration of the improved accuracy that the sliding window technique offers. There are two drawbacks to using this approach versus the simpler method of dividing the text into equally sized substrings. First, it necessitates more memory, as it produces more substrings. Secondly, its time complexity is inversely proportional to the slide distance, i.e., the smaller the slide distance, the slower the algorithm becomes (O(n) in the worst case with a slide distance of 1).

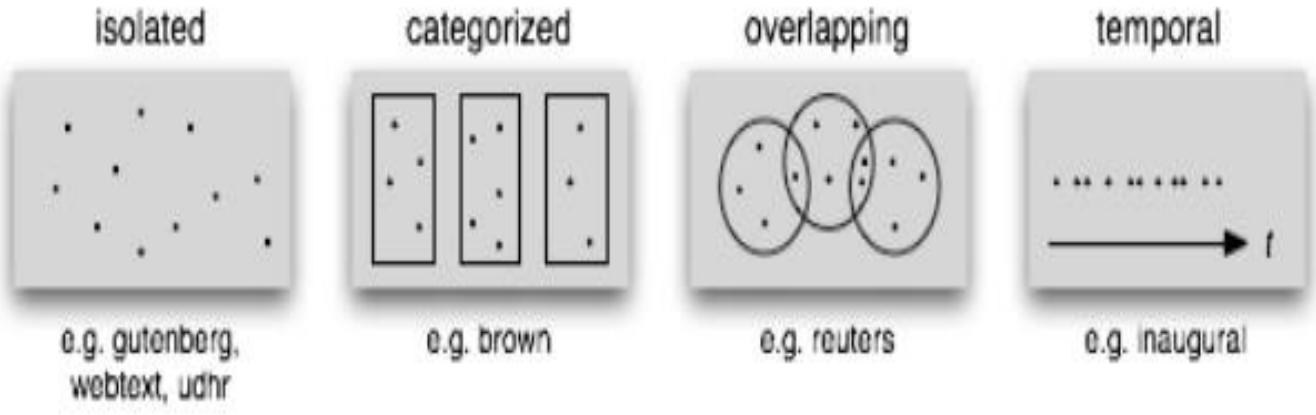
The final downside to these latter two techniques is that instead of returning cohesive sentences as matches, they return disjointed substrings. It would be more useful to the researchers that will be using this project to return the entire sentence that the match was found in. We can solve this problem by first keeping track of the start and end indices of each substring within the text, and then decrementing and incrementing these indices, respectively, until either a period is found or the start/end of the text is reached. In the above example, this type of sentence reconstruction would result in the entire excerpt being returned since there is no period in the text. This allows the algorithm to not only identify quotes which are paraphrases, but also return the entire unabridged quote.

6.11 Accessing Corpora

6.11.1 Design Summary

Most corpora store their contents in one of four different ways. This is extremely relevant to how we will automate the process of obtaining their data. The most ideal scenario is that we can leverage some sort of prebuilt API that the corpus offers, which would allow us to obtain its data easily and in a structured way. Unfortunately, as far as we know, Google Books is the only corpus with this feature. Thus, for all other corpora, we will need to scrape all necessary data.

However, we know that we most likely will not need all the data stored in a corpus. We will not have to obtain every single document they have. For starters, we know that any work published after 1755 cannot be the source of a quote in the



dictionary, since the dictionary itself was published in 1755. For the corpora that have some sort of temporal structure (sorted or searchable by publication date), we can utilize this information to narrow down the texts we need to extract and find them all quickly for download. Furthermore, if the texts in a corpus are categorized, and even if those categories overlap, that would give us an easier way of obtaining the metadata for groups of text and accessing many texts at once.

Lastly, for the corpora for Gutenberg with no internal structure for their text, we will just need to grab as much as we can, although it might still be possible to identify their publication date before we download. Basically, anything we can do to ease our storage needs and make the scraper more efficient is a priority.

6.12 Implementing the Algorithm as a Command Line Application

6.12.1 Design Summary

As discussed in the research section of this paper, our team will be using the begins library to implement the fuzzy search algorithm as a command line

Figure 54: The different categorization schema of online corpora [65].

application. Figure 55 demonstrates what the implementation of the fuzzy search algorithm as a program supporting command line arguments would look like. This program will require a number of command line arguments, some of which may be

optional and/or have default values. We currently plan to provide support for the following arguments:

- json_file_path (necessary): The path to the JSON file containing all of the sponsor-provided quote data, after being converted from its original Excel file format. We may decide to provide a default value for this argument so that the program will search for the quote JSON file in the same directory it was executed in by default.
- database_connection_string (necessary): The connection string to use to connect to the project database, so that the algorithm results may be written to the database.

```
import begin


@begin.start
def fuzzy_search(
    json_file_path: "Path to the JSON file containing the quote data",
    database_connection_string: "Connection string to the project MySQL database"
):
    ...

```

Figure 55: A draft of what the function signature for the fuzzy search algorithm would look like if the above command line arguments were supported.

We may later decide to change the variable database_connection_string from being a command line argument to being an environment variable instead. This is because this value is not likely to change frequently and storing it as an environment variable is a more secure way of recording it. See section 6.13 of this paper for details on how our team may access environment variables through Python.

A more detailed explanation of the project's use of command line arguments and environment variables can be found in the project GitHub repo.

6.12.2 Stretch Goals

One stretch goal would be to add support for the following command line argument to the program:

- Corpora (optional): The designated corpora that the algorithm will search through. If this option is omitted, then the algorithm will search through all available corpora. Adding support for this command line arguments may

improve the algorithm’s run time, since it will not have to search through the entire set of corpora when processing each quote.

6.13 Accessing Environment Variables through Python

6.13.1 Design Summary

Currently, our team is developing some of the project’s Python programs to accept certain values, such as the database connection string, as command line arguments. As we develop the project, we may find it easier to store these variables as environment variables and read them from the environment instead. Not only would this be more secure, but it would also be a more ideal method of accessing variables that do not change often. We can leverage two Python modules in particular to accomplish this task.

6.13.2 Accessing Environment Variables via Python's OS Module

If we decide to store certain variables as environment variables, Python's built-in *os* module provides an easy way to access them via the *environ* mapping object. This object allows one to obtain the values of set environment variables by a call to its `get` method (alternatively, we could use the os module's `getenv` function to obtain the value of an environment variable as well) [60].

6.13.3 Simulating Environment Variables via the Dotenv Module

Storing environment variables on our local machines while developing our Python programs could get a little messy. Furthermore, manually keeping track of which variables are and are not set could lead to confusion. Thankfully, there exists the *dotenv* module to resolve this issue.

The dotenv module allows one to store environment variables in a special file named ".dotenv", load them to a program's environment at runtime, and access them via the os module as if they were normally placed environment variables. Usage is straightforward: first one imports the dotenv module, then calls the its

`load_dotenv` function (optionally specifying the file path to the .dotenv file if it is not in the same directory as the program being run), and finally the environment variables are ready to access [61]. Figure 56 encapsulates all the steps of this process in one succinct screenshot.

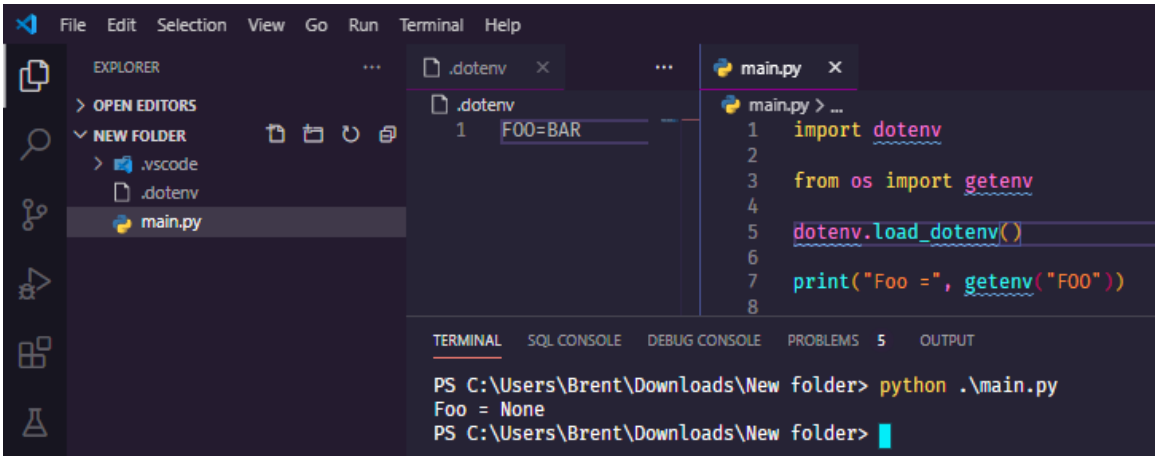


Figure 56: A simple Python program which uses the dotenv module to store and load environment variables.

6.14 Build Plan

6.14.1 Design Summary

There are a number of software development models that we could have chosen to adhere to while developing this project. Some models favor incremental, and iterative approaches to project development; others offer more structured development strategies; and a few emphasize responsive and dynamic development methods. Our project has clear and static requirements, making a structured development approach a more likely candidate for selection. After examining some of the more popular structured development processes available to us, we decided to adopt the Waterfall Model of software development, in conjunction with test-driven development. The Waterfall Model consists of seven phases: planning, requirements, system and software design, implementation, testing, deployment, and maintenance and updates. The largest benefit of choosing this model is its simplicity, as we need only implement its seven stages sequentially. The drawback to this model is that it is not very flexible to major or sudden changes in requirements, but this should not be a problem for our team

since our project has clear requirements and is unlikely to suffer from sudden changes in specifications. More details about test-driven development are provided in section 7 of this paper, *Testing Details*.

We have spent the Fall semester of 2020 in the planning, requirements, and design phases of the development process, and some of us have already taken the first steps into the implementation phase by developing small parts of our assigned segments of the project. In the Spring semester of 2021, we will complete the implementation of the project and then begin testing. We have already created a number of test cases during the past three months of planning, so the testing phase should proceed quickly. Since the project is to be an internal application and is to remain on the sponsor's server space at UCF, the deployment stage will require little additional effort beyond simply logging into the server and cloning the code from the project's GitHub repository. Finally, if the sponsor would like the project to be maintained or updated, our project's documentation and dependency-management tools should enable future teams to do so.

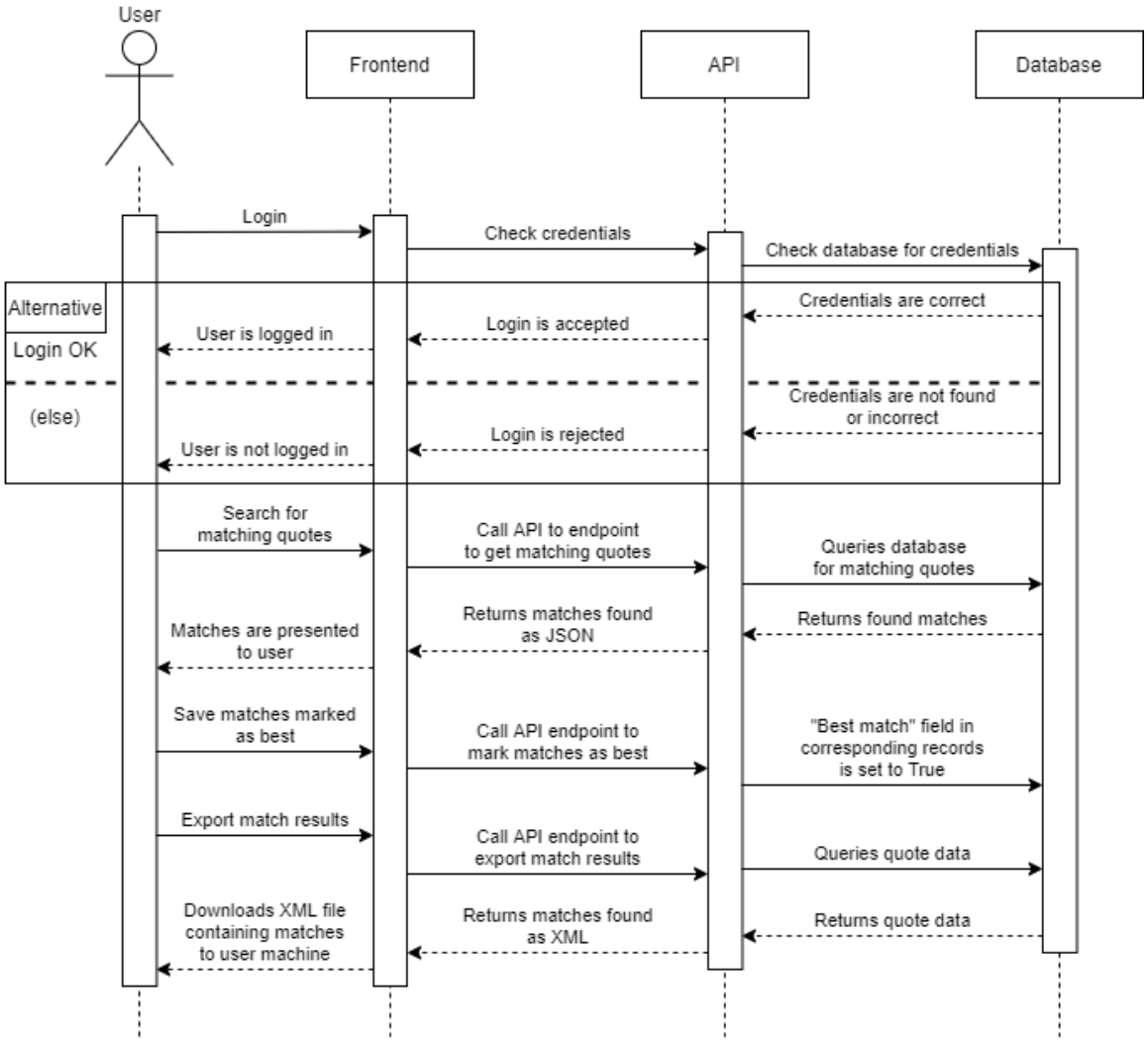


Figure 57: Project Sequence Diagram

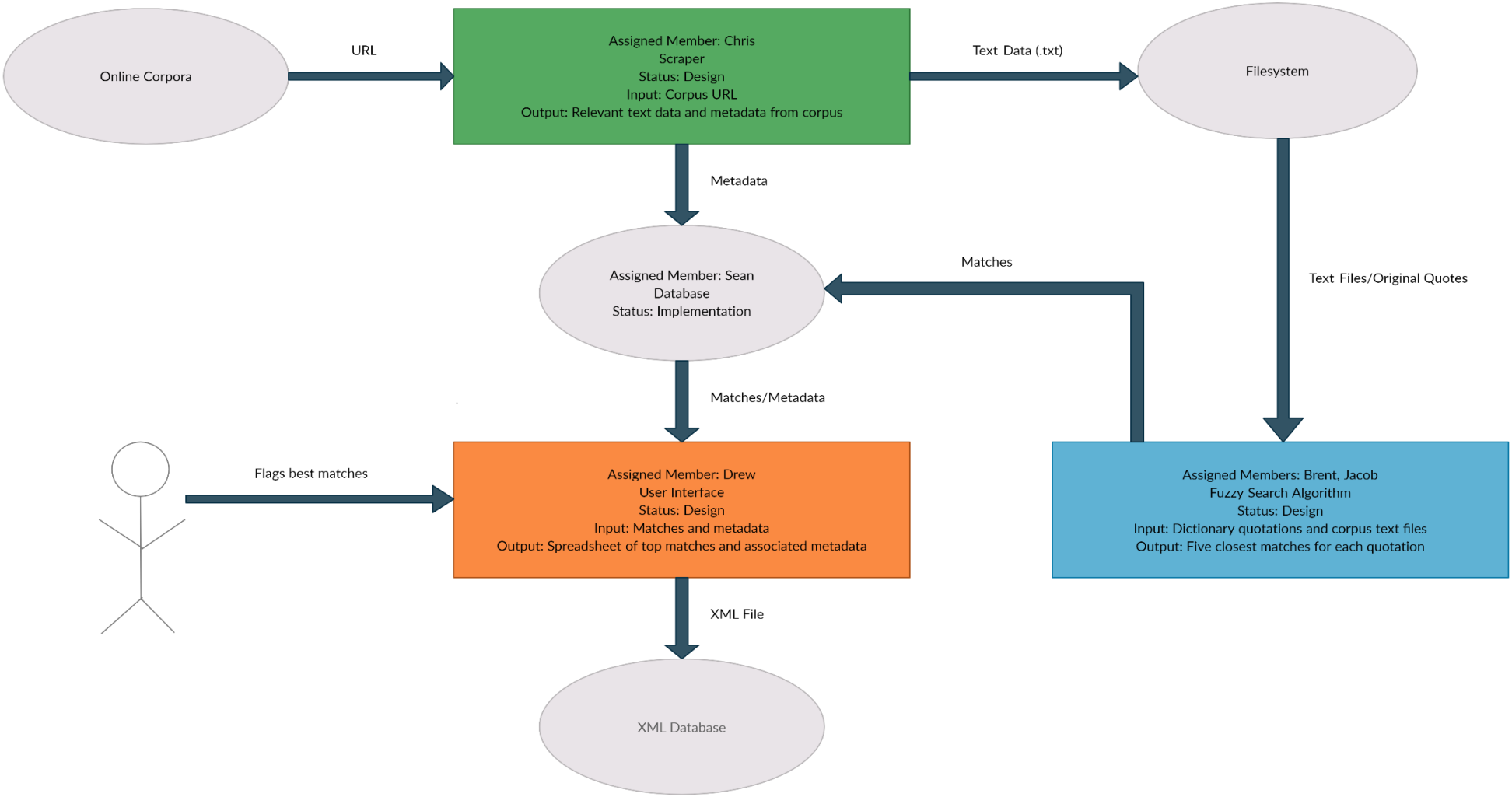


Figure 58: Project Block Diagram

7 Testing Details

7.1 Test-Driven Development

There are many different methodologies for how developers should design and implement their programs, and we felt the one most suitable for developing the algorithm was test-driven development. It felt extremely suitable for this component of the project because it is almost impossible to know if our code will work until we test it. The theory might agree with our code and tell us it should work, but until we run it against many test cases, we can never be certain, due to the inherent ambiguities of natural language processing and the fact that we cannot know what our results should actually be (in a way, it is almost random). This is why it makes so much sense to design our algorithm with a test-driven approach.

Test-driven development is a development process in which the test cases are created before any code is written, and so the test cases essentially *become* your software requirements. Once you satisfy all the test cases, you have written successful code. This is a subtle but important difference from normal software development. Instead of the focus being on what the code is supposed to do, we hone in on writing code that will produce the right results for the test cases.

Our first step in the test-driven development process was to establish the function signatures required to write our code. Once we had those, we wrote test cases for each function. Since our algorithm is going to be written in Python, we have taken advantage of an extremely useful module called *PyTest*, which makes testing a breeze. Then, we went ahead and designed pseudocode to give us a head start on the coding process. In Senior Design II, we will begin writing actual code according to the test-driven development approach.

7.1.1 Unit Testing

During the development of the various components of this project, we will be performing unit tests using the *PyTest* library for Python. This will allow us to create tests and upload them to GitHub so we can develop the tests as a team. Additionally, the output of a test run can be sent to a local file which can then be uploaded to GitHub for the team to view. The unit tests that we create will help us find minor discrepancies in how we wrote our function definitions, allowing us to

maintain a continuous development cycle without having to worry about whether small parts of the project are functional. Figure 59 is an example of a basic unit test made with *PyTest*.

```python
import pytest

def add(x, y):
    return x + y

def raise_exception():
    raise SystemExit(1)

def test_add():
    assert add(3, 6) == 10

def test_exception():
    with pytest.raises(SystemExit):
        raise_exception()
```

Figure 59: A simple example of a *PyTest* unit test.

Figure 59 shows two of the many different features that *PyTest* offers for unit testing. This example demonstrates how a *PyTest* unit test can verify that the return values of functions are what they are expected to be, as well as how a developer can also test whether a function throws a desired exception or not. Figure 60 shows the output from running this test with the terminal command `pytest unit_test.py`.

```
==================================== test session starts ====================================
platform linux2 -- Python 2.7.17, pytest-3.3.2, py-1.5.2, pluggy-0.6.0
rootdir: /home/chris/Documents/_School/Fourth_year/Sem1/COP4934/python_examples, inifile:
collected 2 items

unit_test.py F.                                                                      [100%]

========================================= FAILURES ==========================================
_____________________________________ test_add ______________________________________

    def test_add():
>       assert add(3, 6) == 10
E       assert 9 == 10
E        +  where 9 = add(3, 6)

unit_test.py:10: AssertionError
============================== 1 failed, 1 passed in 0.02 seconds ===============================
```

Figure 60: Output from running the example *PyTest* unit test.

This example illustrates the general procedure we will follow when performing unit tests for this project. Each Python class we create will have an associated unit test class stored in a separate file that will contain the tests being performed for that class. Each file of unit tests will be run when each class is completed as well as any time significant modifications are made to the class. This test schedule will ensure that we are creating robust classes and functions and that our modifications to them do not introduce new problems.

Corpora Scraper

Unit testing the for the corpora scraper will involve checking the return values when calling the scraper function with various types of parameters passed in. When calling the corpora scraper function, the only thing that should be passed in is a string containing the URL to begin scraping at. If anything other than a string is passed as a parameter, the function should immediately return an error code and log an error message. If a string is passed in but the string cannot be loaded through an HTTP request (i.e. it is an invalid URL or not a URL at all), the function will return a different error code and log a relevant error message. Similarly, if the scraper can finish processing an entire corpus, it will return a success code. All error codes will be in the form of a negative integer, and all success codes will be in the form of a positive integer. Figure 61 shows two of the *PyTest* unit tests written for the corpora scraper, including the main scraping function and a helper function written to facilitate exception handling more efficiently.

```
1   import pytest
2   from scraper import scrape
3   from scraper import process_page
4
5   fileindex: int = 1
6
7   def test_scrape_none():
8       assert scrape(None) == -3
9
10  def test_process_page_none():
11      with pytest.raises(Exception):
12          process_page(None, fileindex)
```

Figure 61: Two test cases from the corpora scraper unit tests

To show proof of concept, the scraper was modified to only process a single page of 25 pieces of literature and then return. This modification speeds up the process of verifying whether the unit tests capture the desired results, as processing an entire corpus may take hours depending on its size. Figure 62 shows the output of running all unit tests written for the corpora scraper using the modified scraper code.

```
$ python -m pytest
============================= test session starts ==============================
platform linux -- Python 3.6.9, pytest-6.1.2, py-1.9.0, pluggy-0.13.1
rootdir: /home/chris/Documents/_School/Fourth_year/Sem1/COP4934/git-repo/scraper
collected 8 items

unit_test.py ........                                                   [100%]

============================== 8 passed in 13.56s ==============================
```

Figure 62: Output from running the corpora scraper unit tests.

Back-End API

There is a general testing system devised using *unittest* to ensure the functionality of the function for each API endpoint.

```
@application.route('/get_matches_by_title', methods=['POST'])
@cross_origin()
def get_matches_by_title():
    db = pymysql.connect(host="chdr.cs.ucf.edu", user="sjd_quotes", password="SJDquotes2020", db="SJDquotes")
    mycursor = db.cursor()
    supplied_textFormat = str(request.form.get('textFormat'))
```

Figure 63: '/get_matches_by_title' endpoint and function in Flask.

We will use this function as an example for the sorts of unit tests that will be written for many other functions. The first few lines of its implementation are shown in Figure 63. This function is used to capture quotations when searching for a title, along with any other search restrictions. At the end, a JSON containing all the quotations and their information is returned.

```python
import unittest
from johnson_api import application
import json

class FlaskTest(unittest.TestCase):
    #check for response 200
    def setUp(self):
        application.config['TESTING'] = True
        application.config['DEBUG'] = False
        self.application = application.test_client()

    def tearDown(self):
        pass
```

Figure 64: Class setup for unit testing in Python in *unittest*.

This is the general setup for the class that will hold all the unit tests for all functions written. The *unittest* and json libraries are imported and a class is created to house all test cases.

```python
if __name__ == "__main__":
    unittest.main()
```

Figure 65: Code to execute unit tests.

After the end of the class, this is added to the end of the script to execute it.
The above two snippets of code are simply boilerplate for the testing script. Aside from this, a function simply needs to be written for each unit test in the class. There are three types of unit tests used to ensure the basic functionality of each API endpoint.

```
def test_matches_by_title(self):
    response = self.application.post('/get_matches_by_title', data=dict(
        title='something',
        condition='all',
        textFormat='contains'
        ), follow_redirects=True)
    self.assertEqual(response.status_code, 200)
```

Figure 66: Unit test to check for successful HTTP request.

The first test depicted above ensures that the HTTP request has succeeded by checking to see if the status code is equal to 200.

```
def test_matches_by_title_content(self):
    response = self.application.post('/get_matches_by_title', data=dict(
        title='something',
        condition='all',
        textFormat='contains'
        ), follow_redirects=True)
    self.assertEqual(response.content_type, 'application/json')
```

Figure 67: Unit test to check for correct return type.

The above test makes sure that the return type of the API function is an “application/json”.

```
def test_matches_by_title_on_medals(self):
    response = self.application.post('/get_matches_by_title', data=dict(
        title='on medals',
        condition='all',
        textFormat='contains'
        ), follow_redirects=True)

    data = response.data
    data_string = data.decode('utf-8')
    data_dict = json.loads(data_string)
    self.assertEqual(len(data_dict), 63)

def test_matches_by_title_garbage_input(self):
    response = self.application.post('/get_matches_by_title', data=dict(
        title='asdf;lajsdf98qefj1opil4j2139f81pjuf12f',
        condition='all',
        textFormat='contains'
        ), follow_redirects=True)

    data = response.data
    data_string = data.decode('utf-8')
    data_dict = json.loads(data_string)
    self.assertEqual(len(data_dict), 0)
```

Figure 68: Unit tests to ensure correct function behavior in various scenarios.

The third type of test is to ensure that the behavior of the function called, when a request is made at the endpoint, is consistent with what is expected for different scenarios. In this case, we have tested the scenario in which the username and password are correct, only the user is correct, and only password is correct and check to see if the right JSON message is returned in each case.

In general, there are three things to test for in each endpoint:

1. HTTP requests are successful
2. The return-type of the API function is correct
3. The API function behaves in correct and consistent ways for an exhaustive set of scenarios that it may encounter

Excel to JSON Conversion

The pytest file for this part of the project will reside in the same directory as *excel_to_json.py*, the Python file responsible for converting the sponsor-provided quote data from Excel to JSON. A test Excel file was made to test this program.

Several functions will be tested in this file. One of them will be the function `quote_to_dict`, which takes quote data and metadata as input, compiles it all into a Python dictionary adhering to a schema that the function `excel_to_df` expects, and returns that dictionary as the result. This function is simple to test since it is *pure*, meaning that its output is based entirely on its input and it does not have any side-effects.

```
6  def test_quote_to_dict():
7      definition = "The first letter of the European alphabets, A, an article set befor
8      assert quote_to_dict(1, definition, "A hunting Chloë went.", "", "Prior.") == {
9          "edition": 1,
10         "definition": "The first letter of the European alphabets, A, an article set
11         "quote": "A hunting Chloë went.",
12         "title": "",
13         "author": "Prior.",
14         "flag": False
15     }
```

Figure 69: One of the tests used to verify the function to convert a quote with the given metadata to a Python dictionary. The full value of the variable *definition* has been omitted for brevity.

Another function that will be tested in the excel_to_json pytest file is the function `df_to_dict`. This function takes as input a Pandas Dataframe object containing quote data from the original Excel file, and returns the quote data in a JSON format

according to the schema outlined in the design summary of the Excel to JSON conversion process. This function is somewhat complicated to test since it needs a correctly formatted Excel file to create the requisite Dataframe object input. A short Excel file was created for this purpose, and its data is nuanced enough to test the function's ability to process correctly formatted quotes as well as incorrectly formatted quotes. Another issue that makes this test more difficult to test is the fact that it calls the function `quote_to_dict`; so, at a certain level the test for `df_to_dict` could be considered an integration test rather than a unit test. For our purposes, however, we will consider the tests for this function to be unit tests if two conditions hold true:

1. The Pandas library successfully converts Excel files into Dataframe objects without errors.

2. All of the test cases for the function `quote_to_dict` are passing.

As long as these two requirements are met, we can be confident that tests for `df_to_dict` only test the functionality of the function `df_to_dict`.

```python
def test_df_to_dict():
    fn = r"test.xlsx"
    df = excel_to_df(fn)

    result = df_to_dict(df)
    expected = {
        "A": [
            {
                "edition": 1,
                "definition": "The first letter of the E
                "quote": "A hunting Chloë went.",
                "title": "",
                "author": "Prior.",
                "flag": False
            },
            {
                "edition": 1,
                "definition": "The first letter of the E
                "quote": "And now a breeze from shore be
                "title": "Ceyx and Alcyone.",
                "author": "Dryden's",
                "flag": False
            },
            {
                "edition": 4,
                "definition": "Letter 'a'",
                "quote": "And now a breeze from shore be
                "title": "",
                "author": "",
                "flag": False
            },
        ]
    }
    assert result == expected
```

Figure 70: One of the tests used to verify the functionality of excel_to_json.py. The full values of some dictionary entries are not shown for brevity.

	A	B	C	D	E	F	G	H
1	HEAD	EDITION	POS	DEFINITION	QUOTE	TITLE	AUTHOR	BIBLSCOPE
2	A	1		The first lette	A hunting	Chloë wer	Prior.	
3	A				And now a	Ceyx and A	Dryden's	
4	A	4		Letter 'a'				

Figure 71: The contents of the test Excel file made to help test excel_to_json.py. Some column values are omitted to test the program's ability to handle empty edition numbers, definitions, and quote texts.

Fuzzy Search Algorithm Functions and Test Cases

The function that does the string comparison is `fuzzySearchInFileString`. It is meant to be called for each text file that needs to be compared with a quote. It

takes as input the original quote and a string that contains the entire contents of a text file. It will tokenize and compare both strings by the previously described method and yield a dictionary that contains the top five matches for that text file. The `topFive` dictionary contains the five highest scoring tokens (the keys) mapped to their normalized Levenshtein distances (the values). Thus, for the test, we will provide a file string that is known to contain the input quote, and we check to ensure that the output dictionary contains the match.

```
def test_fuzzySearchInFileString():
    q = "Yet from those flames no light, but rather darkness visible."
    s = "Him the Almighty Power Hurled headlong flaming from th' ethereal sky With hideous ruin and combustion down " +
    "To bottomless perdition, there to dwell In adamantine chains and penal fire, Who durst defy th' Omnipotent to arms. " +
    "As far as angels' ken. Yet from those flames No light, but rather darkness visible."

    d = fuzzySearchInFileString(q, s)

    assert "Yet from those flames no light, but rather darkness visible." in d
```

Figure 72: Function to test `fuzzySearchInFileString`

The idea is that, for each quote, it runs once per text file in the filesystem. It is called by the function `fuzzySearchOverCorpora` that, for each quote, runs once per corpora folder in the filesystem. The job of `fuzzySearchOverCorpora` is basically to run the loop that calls `fuzzySearchInFileString`. However, it does something else important. It manages the top five matches list for the entire corpora on which it runs. It does this by taking both its current top five and the new top five returned by the most recent call of `fuzzySearchInFileString`, combining them into a dictionary of ten, and then extracting the new top five. It takes as input a quote and the filepath of a corpus folder, and returns the top five matches for the corpora.

```
def test_newTopFive():
    d1 = {"a":10, "b":25, "c":75, "d":2, "e":120}
    d2 = {"f":100, "g":30, "h":50, "i":67, "j":90}

    d3 = newTopFive(d1, d2)

    assert d3 == {"e":120, "f":100, "j":90, "c":75, "i":67}
```

Figure 73: Function to test `newTopFive`

The purpose of the function `newTopFive` is simply to extract the new top five matches from two dictionaries of five matches. It does this by first combining them

into a single dictionary of ten matches, and then calling `getTopFive`, which does the actual evaluation and deletion. It takes as input two dictionaries of size five, and returns one dictionary of size five containing the five highest key-value pairs from both dictionaries. The test simply ensures that the right values are returned.

```
def test_stringToWordSet():
    s = "Awake, arise, or be for ever fall'n."
    ws = stringToWordSet(s)

    assert ws == {"awake", "arise", "or", "be", "for", "ever", "falln"}
```

Figure 74: Function to test `stringToWordSet`

The function `stringToWordSet` simply takes a string as input and returns a set, where each member of the set was a word in the string. The purpose of this function is to convert each quote and token to word sets to calculate the Jaccard index. Once it removes punctuation, it must first convert the string to a list of words before converting into a set of words. If you convert a string directly to a set, each set member will be a character, not a word. Thus, the test case simply ensures that the created set contains each word in the original string as a member.

```
def test_getTopFive():
    d1 = {"a":10, "b":30, "c":40, "d":20, "e":70, "f":50, "g":60}
    d2 = getTopFive(d1)

    assert d2 == {"e":70, "g":60, "f":50, "c":40, "b":30}
```

Figure 75: Function to test `getTopFive`

The function `getTopFive` is a helper function to `newTopFive` that does the actual evaluation of pairs. It takes as input a single dictionary, and from that dictionary extracts the five key-value pairs with the highest values. It does this by copying the highest pair to a new dictionary, deleting that pair from the old dictionary, and then pulling the new highest pair and repeating until five pairs are extracted. The test cases simply ensure that the correct values are extracted and returned.

```
def test_getPuncSeq():
    q = "Wow! So cool! But why? Nice. Thanks!"
    s = getPuncSeq(q)

    assert s == "!!?.!"
```

Figure 76: Function to test `getPuncSeq`.

This is a simple but important function that obtains the sentence-ending punctuation sequence from a quote. It takes a string as input and returns a string that contains as characters the sentence-ending punctuation, as seen above. This sequence string is necessary for the `splitByPunc` function, which is described next.

```
def test_splitByPunc():
    s = "Hello! How are you? I'm good! Okay."
    p = splitByPunc(s)

    assert p == ["Hello!", "How are you?", "I'm good!", "Okay."]
```

Figure 77: Function to test `splitByPunc`

The function `splitByPunc` is one of the most important helper functions in the code. It takes as input a string and returns a list of sentences from the string, delimiters included. It does this by delimiting based on periods, question marks, and exclamation points. This function will also have to implement some sort of ruleset for sentence segmentation to ignore periods that do not end a sentence, as described previously. It makes use of the *re* Python module to tokenize strings using multiple delimiters. In the regular expression, the parentheses are used to tell the function that it should keep the delimiters in the list, rather than discard them. Thus, the test case simply checks that the output is a list where each item is a sentence that ends in its punctuation mark.

This function executes the sentence segmentation aspect of the code, and so is one of the few parts that involves true natural language processing. It is important that this code works in most cases, if not all, to obtain the most accurate results from the rest of the algorithm. Basically, this code must work properly before anything after it matters.

```
def test_combineByPunc():
    list = ["Yes!", "Indeed, sir.", "Are you okay?"]
    seq = ".?"

    r = combineByPunc(list, seq)
    assert r == ["Yes!", "Indeed, sir. Are you okay?"]
```

Figure 78: Function to test `combineByPunc`

Another extremely important function is `combineByPunc`, which takes a list of sentences and a punctuation sequence, and combines adjacent sentences based on the sequence. Basically, it looks at the length of the punctuation sequence and checks the last characters of that many adjacent list items, and if all those characters match, then the sentences themselves are combined and stored in a new list. If the original quote had one sentence, then no action is taken. Thus, the test cases will confirm that the function identifies all occurrences in lists that require combination based on the sequence.

The idea behind this is to enhance the probability that, if the actual quote exists in the file text, it will end up by itself in its own token. That way, it is guaranteed to produce a match when compared with the original quote. If we could not guarantee

that the match would have its own token, then there would be no value in comparing tokens, since any token could contain arbitrary text that is not logically consistent (unlike a sentence).

7.1.2 Integration Testing

Corpora Scraper

To ensure that the corpora scraper works well with the overall system developed for this project, we will perform a series of integration tests. These tests will test the functionality of the scraper itself and how the system processes the information it produces. To start, the scraper functionality will be tested by simply giving the scraper a corpus to process. Initially this corpus will be limited to 5 pieces of literature, and once that test produces the desired result consistently, we will increase the size of the corpus until the scraper can successfully process a substantially large corpus, such as the Library of Congress.

Once the basic functionality of the scraper is verified, we will then focus on the data that the scraper produces, namely the literature metadata and full text. The metadata must be formatted in a way that can be stored in the database, and this is quite simple to verify as we will clearly be able to see the contents of the database and whether the format we chose is correct. The full text will need to be stored on the file system in a way that can be most efficiently accessed by the search algorithm. The literature text itself can simply be stored in plain text, but we will need to develop a file naming system to ensure that the search algorithm can quickly find the text it is looking for. This data produced by the scraper will be fed into the search algorithm to verify that the way it is formatted allows the search algorithm to run at its full efficiency.

While there are many tools available to perform integration testing, the nature of this project does not lend well to using these tools. As previously stated, the corpora scraper will simply extract the full written text to the local file system and the literature metadata to the database. The data will not be fed directly into the search algorithm. The search algorithm will go through each written text tile looking for a match to a specific quote, and when it finds a match, retrieve the metadata for that piece of literature.

The fuzzy search algorithm itself is perhaps the most difficult part of the project to test, as it integrates many of the project's other functions. Because of this, the tests for the fuzzy search algorithm may be considered more along the lines of integration tests rather than unit tests. As with the other functions, a number of test cases will be made to test this component of the project; however, each of these test cases will involve either searching over a file, a set of files (a mock corpus), or a set of sets of files (mock corpora). Due to the intricate nature of the algorithm, it is difficult to manually predict the exact outcome of running the algorithm over even a modest number of average-sized files. It is possible to predict the general expected outcome, however, so long as we can be sure beforehand that the quote the algorithm is searching for is within the designated search corpora. In these cases, we can write automated tests to check that running the algorithm over the corpora returns the expected quote in the list of top five matching quotes. We can then experiment with the algorithm's effectiveness at identifying similar quotes by slightly altering the entered quote before passing it to the algorithm. If the algorithm works well, then it should return the original quote in its list of top five matches despite the alterations. For example, we could search Shakespeare's *Hamlet* for the well-known quote "To be, or not to be, that is the question" to verify that the algorithm can correctly identify perfect quotes, and then adjust the quote slightly (e.g., by changing the phrase "not to be" to "not be") to confirm the algorithm's ability to identify quotes that contain errors.

```
function test_fuzzy_search_finds_quote(quote, corpora)
 top_five_matches = fuzzy_search_for_qoute_in_corpora(quote, corpora)
 if quote is in top_five_matches
  test passes
 else
  test fails
```

Figure 81: High-level pseudocode for an integration test of the fuzzy search algorithm for a quote over some corpora, when the entered quote is known to be in the corpora.

In the case that the quote that is being searched for is not actually in the corpora being searched through, however, it is not immediately clear what the desired outcome should be (and therefore what to test our results against). For these cases, we may simply print the results of the algorithm to a file and manually check the results to roughly gauge how similar they are to the quote that was searched for. Even if the quote to be searched for is not actually present in the corpora, we must still ensure that the algorithm returns the closest matching quotes that it can find without returning false positives. Testing for this second case is arguably more critical than the first, since when the algorithm searches for each quote in *Johnson's Dictionary*, we cannot be entirely certain that any of the quotes will exactly match any of those in our prepared corpora.

To test the accuracy of the algorithm itself, we created 100 test cases from a list of quotes that we had manually found matches for beforehand. To test the accuracy of the algorithm when searching for a given quote in a single file, we ran the algorithm to search for a quote within the file that contained its match. We then checked the top five matches that the algorithm found to see if any of them were the correct match. If any of them were, the test would pass; if not, then it would fail. This test was implemented in code by iterating through each of the matches returned by the algorithm and checking if any of them contained a substring of the correct match which uniquely appeared in the text file. An example test case is shown below.

```
# Richard III
(
    "Up with my tent, here will I lie to night; But where to morrow? — Well, all's one for that.",
    "tests/test-txts/richard-iii.txt",
    "Up With my tent! Here will I lie to-night;"
),
```

Figure 82 An example test case, consisting of a quote from SJD, a path to the text file to search for the quote in, and the correct match.

To test the accuracy of the algorithm when searching over corpora, we ran the algorithm to search for the same quotes from the file tests above but searched over all the files used for testing instead of only searching for a quote within the file containing its correct match.

Throughout the implementation phases of the project, we experimented with different methods of segmenting the text and various algorithms for determining string similarity. We tested various combinations of these text segmentation

methods and string comparison algorithms to determine which combination would produce the most accurate results in the shortest time possible. On the following pages are a spreadsheet and bar graphs detailing our test results.

Text Tokenization Method	String Comparison Metric	Percentage of Test Cases Passed	Average Score of Top Matches	Average Runtime for Testing (Seconds)
Split by ending punctuation	Python Levenshtein	15	37.54275439	27.988
Split by ending punctuation	Python Levensthein and Jaccard Index	15	46.52404098	30.514
Split by ending punctuation	Rapidfuzz Levenshtein	14	40.38929932	23.616
Split by ending punctuation	Rapidfuzz Levenshtein with Sentence Reconstruction	14	40.38929932	24.908
Sentence segmentation	Python Levenshtein	94	69.46216022	209.372
Sentence segmentation	Python Levensthein and Jaccard Index	97	75.03601815	620.95
Sentence segmentation	Rapidfuzz Levenshtein	97	73.09745559	287.542
Sentence segmentation	Rapidfuzz Levenshtein with Sentence Reconstruction	0	0	0
Sliding window (slide distance = quote length)	Python Levenshtein	62	52.54322267	20.664
Sliding window (slide distance = quote length)	Python Levensthein and Jaccard Index	60	59.86352614	57.532
Sliding window (slide distance = quote length)	Rapidfuzz Levenshtein	59	55.79981572	21.78
Sliding window (slide distance = quote length)	Rapidfuzz Levenshtein with Sentence Reconstruction	91	55.79981572	20.546
Sliding window (slide distance = 1 / 2 quote length)	Python Levenshtein	99	57.54480458	28.49
Sliding window (slide distance = 1 / 2 quote length)	Python Levensthein and Jaccard Index	98	63.69153157	87.478
Sliding window (slide distance = 1 / 2 quote length)	Rapidfuzz Levenshtein	100	59.97129936	38.254
Sliding window (slide distance = 1 / 2 quote length)	Rapidfuzz Levenshtein with Sentence Reconstruction	100	59.97129936	36.378

Figure 83: Test results table. For the sliding window tests, the window size is equal to the length of the quote being searched for. Testing was conducted on an Intel Core i7-7700K CPU @ 4.20GHz, 4200 Mhz, 4 Cores, 8 Logical Processors, with 32GB RAM, running Windows 10. Scores for rapidfuzz Levenshtein with and without sentence reconstruction are the same because sentence reconstruction is done after obtaining matches. A warmup round was conducted before each 5 rounds of testing to obtain runtime average.

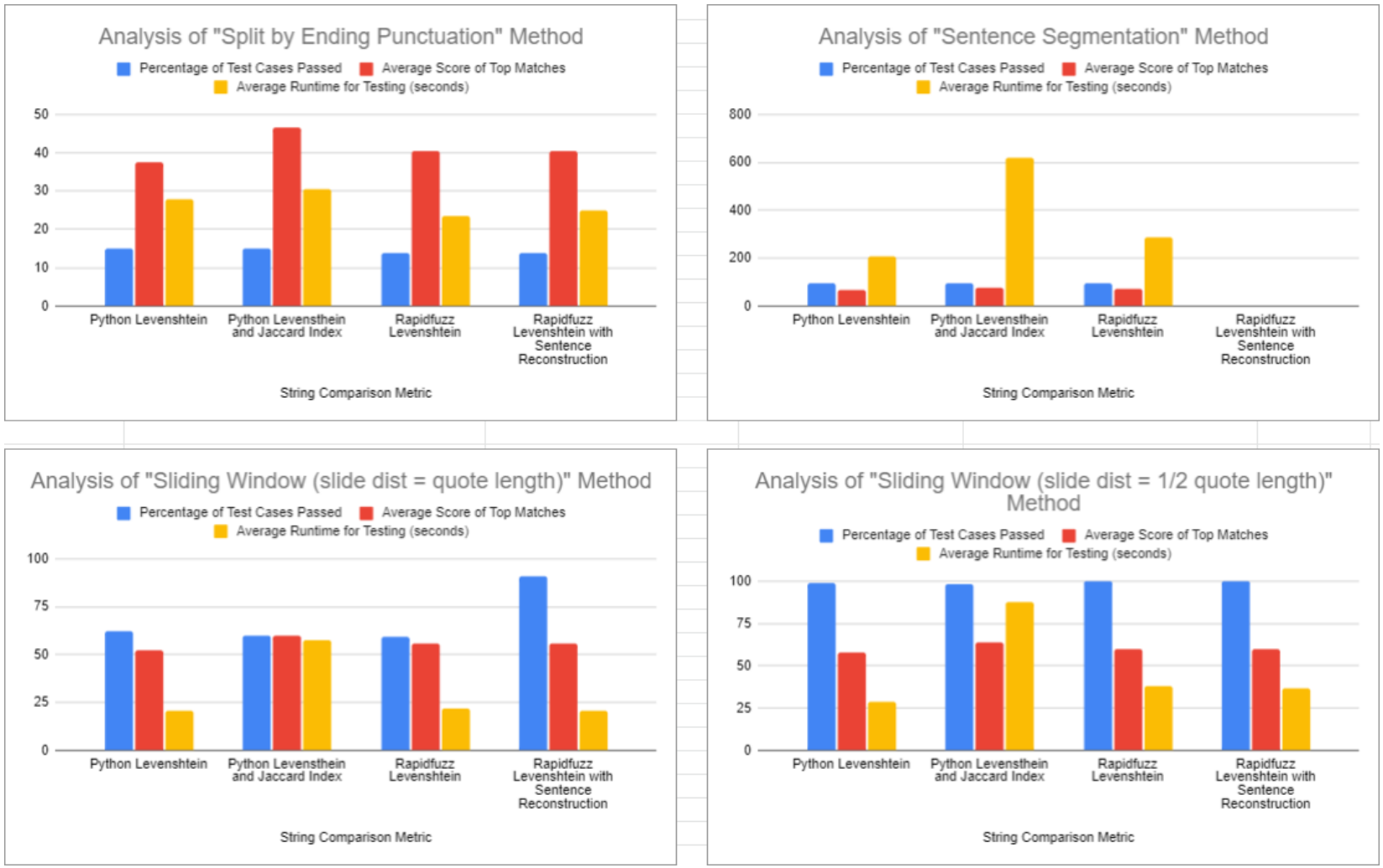


Figure 84: Bar graphs illustrating algorithm test results.

In our code, the main function is not directly tested, but it plays an important part in the overall process. Not only is it where the algorithm begins running, it is also where we connect to the development server and the database. It is also the home of the main loop that runs across all of the corpora and quotes. Equally importantly, it handles part of the edition-handling process.

To convert the quotes to an accessible format, we take the Excel spreadsheet and convert it to a JSON file using a Python script. From there, we can easily extract the JSON data directly into our code. Converting to JSON also adds a flag attribute to every quote, which is initially set to false. And when a quote is detected to be sufficiently similar across editions, rather than delete it, we simply add a flag attribute to it to the JSON for that quote. If the flag is set to true, the main function will ignore that quote and not process it. It will then continue processing normally.

7.1.3 Performance Monitoring

The UCF development server that we ran the algorithm on had performance monitoring software installed on it, so we were able to obtain data of the algorithm’s performance. Below are three graphs displaying some of this data:

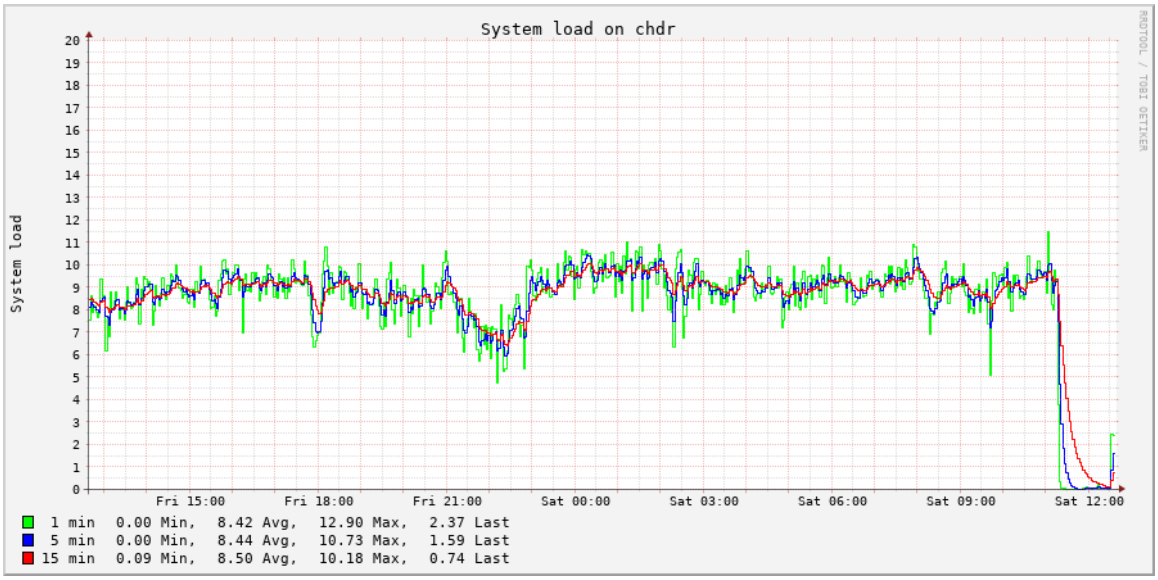


Figure 85: System load of last 24 of first round of quick lookup. Notice the sharp decline in CPU usage at the end once the algorithm finishes. Three rounds of

manual quick lookup were performed, and out of the three, the first round required the most time. It started running during the evening of Wednesday, March 10th, and finished around 11:30 AM on Saturday, March 13 (about 3 days).

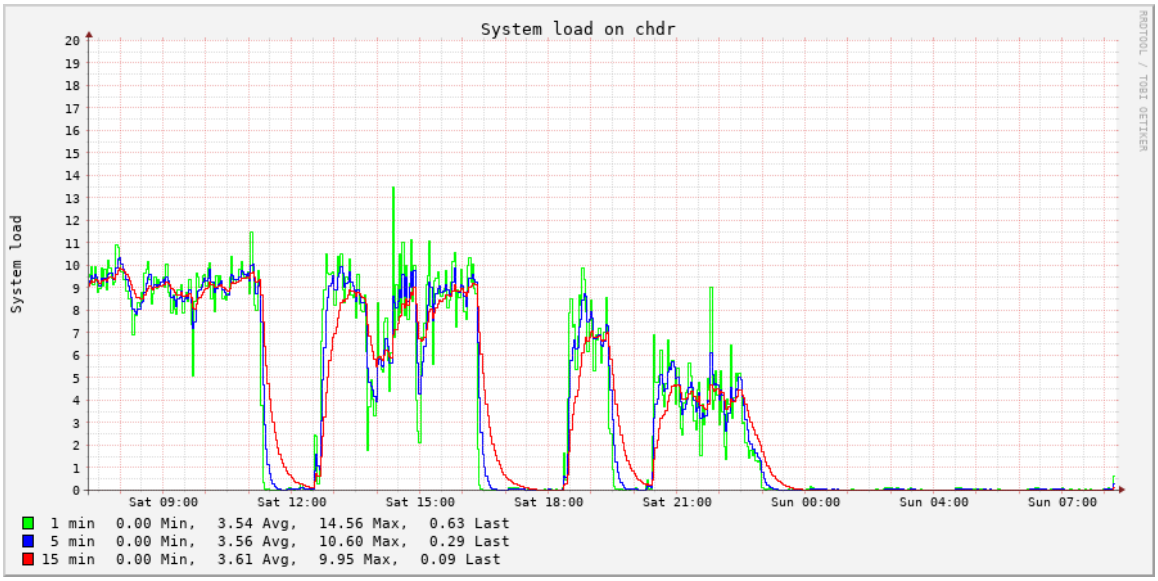


Figure 86: System load of last 24 hours of automatic quick lookup. The huge dips in system load are likely due to outliers – quotes that take a much longer time to search for than the rest. The algorithm currently waits until all the quotes in the current chunk that it is performing a multiprocessed search on are finished until moving onto the next quote, so that explains these dips.

Figure 87: System load of search for remaining quotes over all corpora – 24 hour period. The shape of this graph indicates that quotes should be grouped in larger chunks for multiprocessing.

7.2 More On Testing

Testing is an extremely important aspect of our development. Previously, we illustrated examples of unit tests that would be utilized to test individual functions. Unit testing is important, but we also recognize the importance of integration testing. One of our first tasks next semester, alongside actual implementation, will be to develop integration tests that will adequately check the capabilities and accuracy of our algorithm when everything is running together.

Our integration tests will be designed to test all the modules concurrently. The goal here is to make sure that everything is being returned in the right format, and that the proper information is being communicated between modules. If the algorithm fails for some reason, the integration tests will help us pinpoint where it failed and why.

We are leaning towards a “big-bang” approach with our test. This means we will not take a top-down or bottom-up testing strategy, where we evaluate modules sequentially in a tiered hierarchy, but rather couple everything together at the start and create one complete software system, and then test that. The main benefit of this is that it saves a lot of time in the testing process, time that we might not have, and if we can accurately record the test cases and results, we will obtain valuable results from the testing.

The final step in our testing process would be validation testing, or quality control. Basically, we will ask two questions: Did we build the product right? Did we build the right product? We need to compare against the specifications and requirements of the project to make sure we implemented them correctly. We also need to make sure we built the product right by evaluating the need of the user and making sure they have everything required to utilize the product effectively.

7.3 Role of Prototyping

Our project will not rely heavily on prototyping since its requirements are mostly static and because we are following the Waterfall Model for its development. That

being said, we will still use some forms of prototyping throughout the project at points in which they are most useful. For instance, during the planning stage of development we employed a form of prototyping by presenting the sponsor with mockups of our front-end designs. This provided the sponsor with deeper insight into our planned solution to their problem, so that they could clarify any misunderstandings that we may not have realized we had.

We again utilized prototyping in the design stage when constructing the ERD diagrams for the project database. This helped reveal discrepancies in different team members' comprehensions of the project requirements. This was critical to perform early on in project development for two reasons. First, since the algorithm team will be producing the data that the team member responsible for the database will be recording, the algorithm team needs a clear understanding of what to include in their output data. Second, because the team member assigned to frontend development will be consuming the data from the database via an API, that team member must know what to expect from that API's responses.

Lastly, we will intersperse the development stage with minor bouts of prototyping to ensure that we are on the right track with our solutions. Chris will need to prototype the corpora scrapers he makes for the various corpora sources he has identified to ensure that they work properly enough for testing, and Brent and Jacob may need to prototype new versions of the fuzzy search algorithm if testing proves that one implementation does not work effectively enough for the project. These latter forms of prototyping will be meant only for rapid, informal use, and will need to be backed by test cases. These informal prototypes will be especially useful for the algorithm team since some use cases will require a more heuristic testing approach.

7.4 End-to-End Testing

Though most of the *Quotations from Johnson's Dictionary Search Engine* project is tested with unit testing, the final testing will be done with End-to-End testing, also known as E2E testing.

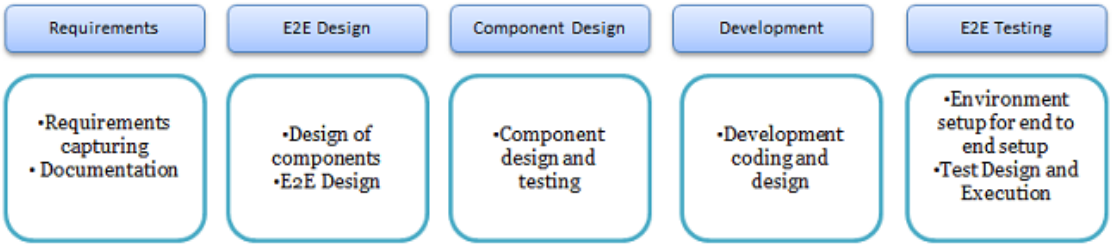


Figure 88: Development timeline for E2E testing, taken from https://www.guru99.com/end-to-end-testing.html

E2E testing is a software testing method that seeks to trial the entire software. As its name suggests, End to End testing validates everything about the software from the moment a user starts using it until task completion. This means that E2E testing encompasses all other aspects of the project, such as user interface display, data integrity, API communication, user inputs, database changes, and more [63]. Since this kind of testing can only be truly executed once the software pieces are assembled, it will be conducted after our other functional testing is complete. E2E testing is accomplished with production-quality data to ensure software efficacy. It is a very involved testing method and requires external interface checks to complete. E2E testing is thus almost always completed manually, and we will take this approach when testing the Search Engine.

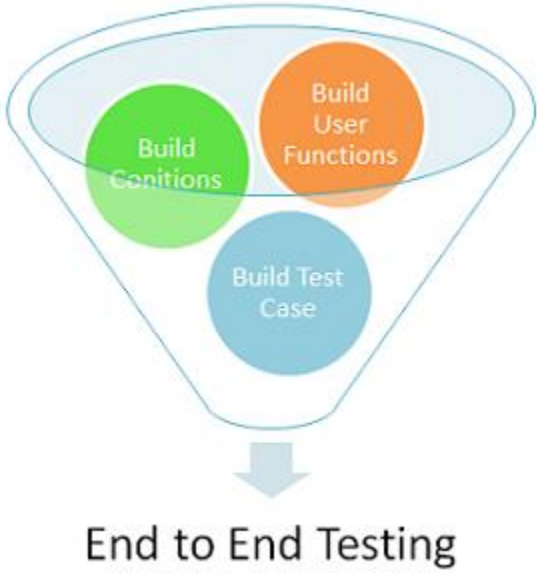


Figure 89: E2E testing component diagram, taken from https://www.guru99.com/end-to-end-testing.html

E2E testing requires an extensive amount of documentation and specifications that cannot be listed here. However, the general steps of creating E2E tests are not particular to the software in question, so they are outlined below.

7.4.1 User Functions

The first step in creation of E2E test cases is to establish user functions, which are similar to the user requirements as specified by the project sponsor. In this step, system features and interconnected components are listed with respect to the function of the software [64]. In this project, the users and data will not interact with the fuzzy search algorithm nor the corpora scraper, and so those project components are not included. Some user functions of the *Search Engine* could be:

- Select a quotation set to search the database by
- Retrieve a random selection of matches
- Search for all quotes from an inputted author
- View all desired quotation match results
- Navigate through search result pages
- Flag the best match out of returned quotation source matches of chosen quotation set
- Return to search screen

These user functions are by no means comprehensive, but they outline how a potential user will see the functions of the software.

7.4.2 Function Conditions

Once the user functions of the final software are identified, a set of conditions for each function is built. These conditions are the everyday scenarios as well as the edge cases, and function as the ‘stress’ in the stress tests [64]. Outlined function conditions should be thorough, as these conditions often account for edge cases individual unit testing may have missed.

- Select a quotation set to search the database by
    - Select a quotation set
    - Change between quotation sets
    - Change between quotation sets after entering a value into the search box (if applicable)
    - Change quotation set after returning from a previous search
- Retrieve a random selection of matches
    - Search again with a random selection search already open
- Search for all quotes from an inputted author
    - Searching without entering an author
    - Searching with search box filled to capacity
    - Searching for a valid author, but with a space before it (“ Homer”)
- View all desired quotation match results
    - No results found
    - Single quote’s matches take up more than one page
    - Results are less than a single page in length
    - Change number of results shown per page
- Navigate through search result pages
    - Navigate pages with navigation buttons

    - User enters a negative number, zero, or a number greater than the page count into the results navigator
    - Attempt to go to an out-of-bounds page with navigation buttons
- Flag the best match out of returned quotation source matches of chosen quotation set
    - Select one match from a quote as Best Match
    - Deselect a Best Match flag
    - Attempt to mark more than one match from a quote as Best Match
- Return to search screen
    - Use the Return to Search button
    - Return to search screen with no search results
    - Return to search screen after exporting search data

As with the user functions, these function conditions do not comprise all possible conditions these functions would need to account for.

7.4.3 Test Scenarios

With the user functions and their conditions asserted, test scenarios can be synthesized. These scenarios are step-by-step tests for the function conditions. Though test scenarios can be designed to test multiple different conditions at once, all conditions must be tested for a complete E2E test: the completed set of individual test scenarios may number in the dozens [63]. Many of these tests can be nearly identical, save for minor changes to account for equally minor condition differences. There is no login page in the front-end, so these tests will all begin implicitly at the main search screen with no selections made unless otherwise specified. An example of a test scenario could be:

1. Select Quotes by Author
2. Change selection to Quotes by Title
3. Enter “MacBeth” into the search box
4. Click Search
5. Scroll through all pages, changing pages with the navigation buttons
6. Select a match at random as Best Match
7. Deselect the match as Best Match
8. Click the Return to Search Button
9. Check that the “MacBeth” search term is saved
10. Click Search Results
11. Exit program

This test scenario is a somewhat realistic workflow that the software will encounter. This test scenario validates a number of the function conditions specified above:

- Select a quotation set to search the database by
    - Select a quotation set
    - Change between quotation sets
- View all desired quotation match results
    - Single quote’s matches take up more than one page
- Navigate through search result pages

    - Navigate pages with navigation buttons

- Flag the best match out of returned quotation source matches of chosen quotation set

    - Select one match from a quote as Best Match

    - Deselect a Best Match flag

- Return to search screen

    - Use the Return to Search button

When finished, with any found bugs fixed, End to End testing will mark the end of our development of the *Quotations from Johnson's Dictionary Search Engine*. Passing E2E testing is the final, crucial step before delivering the project.

8 Facilities & Equipment

The project does not require any special facilities or equipment outside of the hardware currently in use for the *Johnson's Dictionary Online* parent project.

The *Johnson's Dictionary Online* server is currently being hosted under the University of Central Florida's VPN, who also owns the server. These technologies are free for our project group to utilize at no cost.

9 Budget and Financing

The finances of this project are modest in size and scope. The sponsor has generously provided a development server with 67 gigabytes of storage, which should be more than enough for our needs. Also, since the user interface is only meant to be used internally by specific employees, its only users will be individuals associated with the project. This means we do not need to host our code on a live server with an active domain. It will remain only on the development server for internal use. The database itself will also be hosted on the development server. Also, since the server is “ours” rather than one we rented online, we have full access to its filesystem, which can be used to store written works from the corpora. However, should the storage become insufficient for our purposes, we can easily incorporate a Digital Ocean droplet to extend our storage space. For an additional 25 gigabytes, it would cost $5/month. For 50 gigabytes, the cost would be $10/month.

Another situation that might arise is a corpus might require users to pay for access to its content. If we come across this situation, we might decide that the data in the corpora is not necessary and we can live without it. If, however, we still want access, the grant money for the overall project (of which we are one part) will go towards covering that cost.

Since the project was to be hosted on UCF’s private development server, our team did not have to purchase web hosting for the project web application. We did, however, purchase additional server hosting to run the fuzzy search algorithm on in parallel. We purchased four VPS XL SSD instances from Contabo for a total of $167.92. We divided this cost among the five us so that we only paid $33.58 each.

10 Project Milestones

- September 16 - Team introductions. Schedule weekly team meetings and bi-weekly sponsor meetings. (completed)
- September 28 - First TA check-in. (completed)
- September 30 - Finalize and submit initial design document. (completed)
- October 2 - Set up Trello for task management. (completed)
- October 11 - Finish algorithm diagram. Finish entity relationship diagram. Finish corpora scraper diagram. (completed)
- October 12 - Check-in meeting presentation. (completed)
- October 25 - Finish corpora scraper draft for proof of concept. (completed)
- October 31 - Finish designing frontend prototypes. Finish designing database structure. Finish Gantt chart. (completed)
- November 6 - Second TA check-in. Create database and implement database structures (tables, etc.). (completed)
- November 13 - Decide on using pipenv to handle dependency management. (completed)
- November 16 - Extract the quotations from the spreadsheet into a JSON format. (completed)
- November 19 - Finish overall project sequence diagram. (completed)
- December 4 - Finalize algorithm research. (completed)
- December 7 - Finish project design document. (completed)
- January 31 - Finish implementing backend API.

- February 17 - Finish implementing all corpora scrapers.
- February 19 - Finish integrating UI framework into the frontend of the web application.
- February 24 - Finish extracting all corpora data and storing it in database
- February 28 - Finish implementing algorithm.
- March 7 - Finish running the algorithm and storing the results in the database.
- March 26 - Finish programming frontend UI and functionality.

11 Project Summary and Conclusion

Our project was composed of various components to serve the express purpose of assisting researchers in correctly identifying the sources of each of the quotes in *Johnson's Dictionary*. While our overall project may achieve a specific goal, we hope that each of its components may be sufficiently extensible to help researchers accomplish scholarly objectives in other fields as well. Future teams could use this project's corpora scraper as a template for creating their own web scrapers to obtain domain-specific data from the Internet. Perhaps the easiest to imagine application of this would be in the case of historians using it to obtain information about a specific historical event en masse (a task quite like our own project's). Another scenario would be in the case of seismologists scouring numerous government websites for data on the frequency of earthquakes in locations across the globe for the last five decades. Other researchers may find the project's front-end design as a suitable template for their own projects after only a few modifications. For instance, biologists may use a very similar design for a project to facilitate the ranking of comparisons of DNA sequences. All they would have to change is the number of columns and column titles in the table, and the front-end design of the application would be nearly finished. Finally, the fuzzy search algorithm could be applied to various domains. A chemist could use it to identify chemical formulas similar to a given formula, and forensic analysts could use it to match a suspect's DNA sequence against those found on evidence at a crime scene. We wish that later teams may be able to use our project as a foundation for realizing diverse goals such as these.

During the Fall semester of 2020, our team members convened regularly with our sponsors and each other to ensure that our project's objectives were clear, and that a plan for reaching those goals was always present. Each team member needed to focus their efforts on a specific part of the project while simultaneously collaborating with the other team members to ensure that their assigned portion of the project would fit with those of other team members once finished. This process has proven to be efficient so far in achieving the project's goals, and we will continue to use it while implementing the designs listed in this document in the following semester. We anticipate the emergence of novel problems and unforeseen obstacles while creating this system and are confident in our ability to solve them through cooperation. If we encounter any problems that appear intractable, we can rely on this document to guide us on how to resolve them. Moreover, this document's comprehensive analysis of each individual component of the project may hopefully benefit the research community as well.

12 References

1. N. Scialli, “What languages, frameworks, and libraries would you put on a front-end developer's 2020 ‘must learn’ list?,” DEV Community, 27-Apr-2020. [Online]. Available: https://dev.to/nas5w/what-languages-frameworks-and-libraries-would-you-put-on-a-front-end-developer-s-2020-must-learn-list-345h. [Accessed: 05-Dec-2020].
2. I. Gerchev, “The 5 Most Popular Front-end Frameworks Compared,” SitePoint, 19-Aug-2020. [Online]. Available: https://www.sitepoint.com/most-popular-frontend-frameworks-compared/. [Accessed: 05-Dec-2020].
3. M. Pekarsky, “Does your web app need a front-end framework?,” Stack Overflow Blog, 06-Feb-2020. [Online]. Available: https://stackoverflow.blog/2020/02/03/is-it-time-for-a-front-end-framework/. [Accessed: 05-Dec-2020].
4. J. babu, “All possible ways of making an API call in JavaScript,” Medium, 07-Jul-2020. [Online]. Available: https://levelup.gitconnected.com/all-possible-ways-of-making-an-api-call-in-plain-javascript-c0dee3c11b8b. [Accessed: 05-Dec-2020].
5. jQuery Introduction. [Online]. Available: https://www.w3schools.com/jquery/jquery_intro.asp. [Accessed: 05-Dec-2020].
6. G. S. Ingersoll, T. S. Morton, and A. L. Farris, Taming text : how to find, organize, and manipulate it. Shelter Island, Ny: Manning, 2013.
7. D. Jurafsky, “Minimum Edit Distance Definition of Minimum Edit Distance,” Oct. 2018. Accessed: Dec. 04, 2020. [Online]. Available: https://web.stanford.edu/class/cs124/lec/med.pdf.
8. A. Haapala, “python-Levenshtein: Python extension for computing string edit distances and similarities.,” PyPI, Feb. 01, 2021. https://pypi.org/project/python-Levenshtein/.
9. “Definition of TRANSPOSITION,” www.merriam-webster.com. https://www.merriam-webster.com/dictionary/transposition (accessed Oct. 19, 2020).
10. G. Fairchild, “pyxDamerauLevenshtein: pyxDamerauLevenshtein implements the Damerau-Levenshtein (DL) edit distance algorithm for Python in Cython for high performance.,” PyPI, Jul. 27, 2020. https://pypi.org/project/pyxDamerauLevenshtein/ (accessed Oct. 19, 2020).

11. Kenneth Tay, “What is Jaro/Jaro-Winkler similarity?,” Statistical Odds & Ends, Sep. 11, 2019. https://statisticaloddsandends.wordpress.com/2019/09/11/what-is-jaro-jaro-winkler-similarity/ (accessed Oct. 19, 2020).
12. J. Turk and M. Stephens, “jellyfish: a library for doing approximate and phonetic matching of strings.,” PyPI, May 21, 2020. https://pypi.org/project/jellyfish/ (accessed Oct. 19, 2020).
13. K. Dreßler and Ngomo, Axel-Cyrille Ngonga, “Time-efficient Execution of Bounded Jaro-Winkler Distances,” in OM’14: Proceedings of the 9th International Conference on Ontology Matching, 2014, pp. 37–48.
14. A. Abramov, “A deep and fuzzy dive into search,” ArangoDB, Jul. 16, 2020. https://www.arangodb.com/2020/07/deep-and-fuzzy-dive-into-search/ (accessed Oct. 19, 2020).
15. G. Kondrak, “N-gram similarity and distance,” in Proceedings of the 12th Conference on String Processing and Information Retrieval Buenos Aires, 2-4 November, Oct. 2005, pp. 115–126, doi: 10.1007/11575832_13.
16. Wikimedia Commons [https://wikimedia.org/api/rest_v1/media/math/render/svg/eaef5aa86949f49e7dc6b9c8c3dd8b233332c9e7]
17. Python Software Foundation, “Built-in Types — Python 3.9.0 Documentation,” Python.org, Nov. 11, 2020. https://docs.python.org/3/library/stdtypes.html (accessed Nov. 11, 2020).
18. National Archives and Records Administration, “Soundex System,” National Archives, May 30, 2007. https://www.archives.gov/research/census/soundex (accessed Oct. 19, 2020).
19. A. Patterson, “Why Writing Your Own Search Engine is Hard,” Queue, vol. 2, no. 2, Art. no. 2, Apr. 2004, doi: 10.1145/988392.988407.
20. S. Ji, G. Li, C. Li, and J. Feng, “Efficient Interactive Fuzzy Keyword Search,” in WWW ’09: Proceedings of the 18th International Conference on World Wide Web, 2009, pp. 371–380, doi: 10.1145/1526709.1526760.
21. R. E. Pattis, “Complexity of Python Operations,” Donald Bren School of Information and Computer Sciences at University of California, Irvine, Mar. 02, 2019. https://www.ics.uci.edu/~pattis/ICS-33/lectures/complexitypython.txt (accessed Oct. 29, 2020).
22. Python Software Foundation, “TimeComplexity - Python Wiki,” Python.org, Aug. 18, 2020. https://wiki.python.org/moin/TimeComplexity (accessed Oct. 29, 2020).
23. K. Reitz, "Requests: HTTP for Humans(TM)," 2016. [Online]. Available: https://requests.readthedocs.io/en/master/.

24. L. Richardson, "Beautiful Soup Documentation," 3 October 2020. [Online]. Available: https://www.crummy.com/software/BeautifulSoup/bs4/doc/.
25. "Making data on the web useful: scraping," School of Data, 2 September 2013. [Online]. Available: https://schoolofdata.org/handbook/courses/scraping/.
26. "Project Gutenberg," Project Gutenberg, 26 August 2020. [Online]. Available: https://www.gutenberg.org/.
27. "Liberty Fund," Liberty Fund, 2020. [Online]. Available: https://www.libertyfund.org/.
28. "Online Library of Liberty," Liberty Fund, Inc., 2020. [Online]. Available: https://oll.libertyfund.org/.
29. "About the Internet Archive," Internet Archive, [Online]. Available: https://archive.org/about/.
30. E. Hardy, "Dhammapala;s Paramattha-Dipani," 9199. [Online]. Available: https://archive.org/details/in.gov.ignca.9199/mode/2up.
31. "Our Research Center," HathiTrust, [Online]. Available: https://www.hathitrust.org/htrc.
32. "About the Library," Library of Congress, [Online]. Available: https://www.loc.gov/about/.
33. H. Krekel, "Usage and Invocations," pytest, 2020. [Online]. Available: https://docs.pytest.org/en/stable/usage.html.
34. "unittest - Unit testing framework," Python Software Foundation, 18 October 2020. [Online]. Available: https://docs.python.org/3/library/unittest.html.
35. "doctest - Test interactive Python examples," Python Software Foundation, 18 October 2020. [Online]. Available: https://docs.python.org/3/library/doctest.html.
36. A. Ajitsaria, "Logging in Python," Real Python, [Online]. Available: https://realpython.com/python-logging/.
37. "Python MySQL - Introduction," TutorialsPoint, [Online]. Available: https://www.tutorialspoint.com/python_data_access/python_mysql_introduction.htm.
38. “ACID Properties & Normalization - SQL Tutorial,” *Intellipaat Blog*. [Online]. Available: https://intellipaat.com/blog/tutorial/sql-tutorial/acid-properties-normalization/.
39. “What's the Difference? Relational vs Non-Relational Databases,” *Izenda*, 06-Oct-2020. [Online]. Available: https://www.izenda.com/relational-vs-non-relational-databases/. [Accessed: 06-Dec-2020].

40. K. Wenzel, “Database Indexes Explained,” *Essential SQL*, 01-May-2020. [Online]. Available: https://www.essentialsql.com/what-is-a-database-index/. [Accessed: 06-Dec-2020].
41. “Flask,” *Full Stack Python*. [Online]. Available: https://www.fullstackpython.com/flask.html. [Accessed: 06-Dec-2020].
42. J. Milton, J. Leonard, and Penguin, Paradise lost. London: Penguin Books, 2003.
43. Python Software Foundation, “sys — System-specific parameters and functions — Python 3.9.1rc1 documentation,” Python.org, Dec. 05, 2020. https://docs.python.org/3/library/sys.html (accessed Dec. 05, 2020).
44. Python Software Foundation, “argparse — Parser for command-line options, arguments and sub-commands — Python 3.9.1rc1 documentation,” Python.org, Dec. 05, 2020. https://docs.python.org/3/library/argparse.html (accessed Dec. 05, 2020).
45. Pallets Projects, “Welcome to Click — Click Documentation (7.x),” click.palletsprojects.com, Jun. 23, 2020. https://click.palletsprojects.com/en/7.x/ (accessed Dec. 05, 2020).
46. Real Python, “Primer on Python Decorators,” Realpython.com, Aug. 22, 2018. https://realpython.com/primer-on-python-decorators/ (accessed Dec. 05, 2019).
47. A. Iles, “begins: Command line programs for busy developers,” PyPI, Aug. 01, 2014. https://pypi.org/project/begins/ (accessed Dec. 05, 2020).
48. Python Software Foundation, “Typing — Support for Type Hints — Python 3.9.1rc1 Documentation,” Python.org, Dec. 05, 2020. https://docs.python.org/3/library/typing.html (accessed Dec. 05, 2020).
49. J. S. F.- js.foundation, “jQuery.getJSON(),” jQuery.getJSON() | jQuery API Documentation. [Online]. Available: https://api.jquery.com/jquery.getjson/. [Accessed: 05-Dec-2020].
50. NumFOCUS, “Python Data Analysis Library — pandas: Python Data Analysis Library,” Pydata.org, Nov. 09, 2020. https://pandas.pydata.org/ (accessed Nov. 09, 2020).
51. Pandas Development Team, “pandas.read_excel — pandas 1.1.4 documentation,” pandas.pydata.org, Oct. 30, 2020. https://pandas.pydata.org/docs/reference/api/pandas.read_excel.html (accessed Nov. 09, 2020).
52. Pandas Development Team, “pandas.DataFrame.iterrows — pandas 1.1.4 documentation,” pandas.pydata.org, Oct. 30, 2020. https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.iterrows.html (accessed Nov. 09, 2020).

53. Pandas Development Team, “pandas.DataFrame.isnull — pandas 1.1.4 documentation,” pandas.pydata.org, Oct. 30, 2020. https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.isnull.html (accessed Nov. 09, 2020).
54. Python Software Foundation, “Built-in Functions — Python 3.9.0 documentation,” docs.python.org, Nov. 18, 2020. https://docs.python.org/3/library/functions.html (accessed Nov. 18, 2020).
55. Python Software Foundation, “json — JSON encoder and decoder — Python 3.9.0 documentation,” docs.python.org, Nov. 18, 2020. https://docs.python.org/3/library/json.html (accessed Nov. 18, 2020).
56. Pipenv Maintainer Team, “pipenv: Python Development Workflow for Humans.,” PyPI, Nov. 04, 2020. https://pypi.org/project/pipenv/ (accessed Nov. 12, 2020).
57. Python Software Foundation, “Installing Packages — Python Packaging User Guide,” packaging.python.org, Nov. 09, 2020. https://packaging.python.org/tutorials/installing-packages (accessed Nov. 12, 2020).
58. Python Software Foundation, “PyPI · The Python Package Index,” PyPI, 2020. https://pypi.org/.
59. Python Software Foundation, “12. Virtual Environments and Packages — Python 3.9.0 Documentation,” Python.org, Nov. 24, 2020. https://docs.python.org/3/tutorial/venv.html (accessed Nov. 25, 2020).
60. Python Software Foundation, “os — Miscellaneous operating system interfaces — Python 3.8.0 documentation,” Python.org, Dec. 05, 2020. https://docs.python.org/3/library/os.html (accessed Dec. 05, 2020).
61. S. Kumar, “python-dotenv · PyPI,” PyPI, Oct. 28, 2020. https://pypi.org/project/python-dotenv/ (accessed Dec. 05, 2020).
62. Contabo, “About Contabo, Your Quality VPS Provider - Contabo,” Contabo.com, 2013. https://contabo.com/en/about-us/.
63. “END-To-END Testing Tutorial: What is E2E Testing with Example,” Guru99. [Online]. Available: https://www.guru99.com/end-to-end-testing.html. [Accessed: 05-Dec-2020].
64. “End To End Testing: A Detailed Guide,” BrowserStack, 05-Feb-2020. [Online]. Available: https://www.browserstack.com/guide/end-to-end-testing. [Accessed: 05-Dec-2020].
65. Bird, Steven, Edward Loper and Ewan Klein (2009), Natural Language Processing with Python. O’Reilly Media Inc.
66. M. Bachmann, “rapidfuzz 1.4.1,” *PyPI*, Mar. 29, 2021. https://pypi.org/project/rapidfuzz/.