Project Alexandria: A Book Exploration Engine

Submitted To

Dr. Constantine Caramanis

Prepared By

Nikhil Garg
Joseph Lubars
Chris Roberts
Gavin Sellers
Sean Steel
Ankit Tandon

EE464 Senior Design Project
Electrical and Computer Engineering Department
University of Texas at Austin

Spring 2015

CONTENTS

CONTENTS (Continued)

TABLES

FIGURES

EXECUTIVE SUMMARY

Our team built Project Alexandria, a web-based book exploration engine that users can leverage to find similar books and to understand connections between them, much like Pandora creates playlists based on musical qualities that songs share. Project Alexandria is unique among book recommendation or exploration systems in both its graph-like interface and its deep analysis of metadata and book contents to generate connections between books. The project required a robust understanding of how books are related and how consumers choose what to read, data science and natural language processing skills, a careful design of an intuitive user interface, and strong competency with modern web technologies. We exceeded our goals and launched our final product for use at http://seniordesign.cloudapp.net.

Examining book relationships is a difficult problem. Our goal was to simplify this process for users by providing useful information about book similarities in an intuitive manner. We created specific requirements to meet these goals: provide users the ability to search by title or author, return results in under two seconds, and allow users to navigate the tool without training. We also specified requirements for the backend system and the features we use to compare books. These requirements were both quantitatively and qualitatively testable.

To meet these requirements, Project Alexandria is split into four parts: features that compare books, graph generation scripts that run those features efficiently on the more than one million books in the database, the website backend that retrieves the data from databases on our server and serves it to a user's browser, and the user interface (UI) with which users interact to browse books. The team developed seven different features used to compare books, using expert opinions and similarity between the genres, descriptions, authors, plot structure, word choice, and readability of books. The graph generation component uses an algorithm called Nearest Neighbor Descent (NND) to find, for each book, the ten most similar books with respect to each feature. The website backend component consists of Django and MongoDB. These tools query the databases generated in the graph generation component, process the resulting data to determine which subset of books to show to the users, and build the user interface. The UI, built using D3.js, jQuery, jQuery-UI, and Bootstrap, is a graph in which books are nodes that are connected with edges that represent the books' similarities. Information about the books is shown on a sidebar, and the user can expand the graph by clicking on nodes.

The design choices for each project component evolved throughout the development of the project. To create our feature set, we first brainstormed specific functions to capture relationships between books. We then spent several months prototyping and evaluating potential functions, finally converging on a final feature set. The graph generation code went through several iterations to improve its performance, and we developed a dynamic weighting algorithm to determine which books to show to the user while emphasizing variety. We also tested a number of website backend and UI technologies to find out what best suited our needs. The user interface design choices were made and modified throughout project development to more intuitively display information about the books based on user feedback.

We extensively tested Project Alexandria with a focus on overall performance and feature correctness, providing quantitative results for website response time, graph generation runtime,

and graph generation accuracy. For individual features we utilized unit tests and manual inspection of resulting book neighbors to verify feature correctness. We used a standard command line utility called "time" along with an accuracy metric called "recall" to compare the performance and accuracy of NND to the brute force graph generation algorithm. To test our website's performance, we relied on the Chrome web browser's developer tools and used a bash script to simulate multiple users connecting to the website. Our website UI's behavior was tested through each team member manually interacting with the graph interface, making sure to exercise all of its interactive functionality such as clicking on nodes and hovering on nodes and edges. We concluded that every component of Project Alexandria definitively met established requirements.

While completing Project Alexandria, the team spent under $20 and generally achieved the project milestones set down in earlier documents with a few exceptions. We temporarily fell behind on our feature engineering goals as creating suitable features ended up being more difficult than we expected. Regardless of hurdles along the way, however, our team met our final deadline of a live demonstration of Project Alexandria at Open House on April 29th.

After completing the project, we identified future improvements that would make Project Alexandria more engaging. We believe advanced search, such as the ability to search "Dystopian novels from the 1980s," would allow users to research much more specific connections than we currently provide. Personalized results and prompting for user feedback would also allow us to customize the site for each specific user. These next steps would make Project Alexandria a better exploration tool, and our team has set up a meeting with Zola Books to discuss our project's future.

Our team is proud of the work we did for Project Alexandria. We believe Project Alexandria achieves the mission we set out to accomplish, and our initial feedback has been incredible. The connections between books made sense to our users at Open House, and the UI is polished and responsive. Audience members at Open House almost uniformly indicated to us that they would use our product. After seeing Project Alexandria for the first time, a product manager at Zola even told us "that looks awesome. Serious congratulations." Our tests, personal experiences, and audience feedback all show that we have successfully developed a usable book exploration engine that provides recommendations on par with GoodReads and Amazon.

1.0 INTRODUCTION

This paper is the final report for our Senior Design project, named Project Alexandria. We set out to organize a large set of book data to allow users to explore similarities between books using an intuitive user interface (UI). Our system allows users to explore how books are connected, much as Pandora explains to users how songs are connected. In the Design Implementation Plan, we defined Project Alexandria as a queryable graph of book data and then described each component, detailing our plans to complete the entire exploration experience by Open House on April 29th. In the Test and Evaluation Plan, we described how we would test each subsystem including book features, graph generation, the backend, and the UI. We detailed both quantitative and qualitative requirements that would determine whether the features and UI work correctly and whether our project would ultimately be deemed a "success." We also described how we would test the end-to-end system connecting the website to the backend database.

In this report, we detail the final design of Project Alexandria, including each of the similarity measures that were used to analyze and compare books, the process of creating the graph of neighbors that is shown to users, the backend servers and processes, and the user interface. We then describe the process of implementing this design, including changes we made such as increasing the importance of searching on the database. Next, we review the tests we ran on each feature and then the final system, concluding that we met the feature quality and website responsiveness standards we set in the Test and Evaluation Plan. We then discuss the minimal costs to run the website, along with ethical concerns of keeping the data secure. Finally, we recommend extensions to the project, including a search functionality that can be implemented with access to more data and the possibility of taking human interaction into account to provide customized recommendations.

2.0 DESIGN PROBLEM STATEMENT

Project Alexandria is a large-scale queryable graph of book data that can satisfy a number of exploratory needs for a user. An end user of Project Alexandria will use the graph as a tool to explore books that have similar qualities. Instead of spending time searching various websites and disjoint catalogues, users will utilize Project Alexandria as a unified, friendly mechanism for book relationship exploration. In this section, we describe the goals, motivation, and

requirements of the book exploration problem that Project Alexandria tries to solve. We also restate the success criteria for our system.

2.1 Problem Definition

Examining the relationships between books can be a daunting task. Project Alexandria is a search tool for books that attempts to solve this problem. A user can leverage Project Alexandria as a tool to discover books with similar qualities and to explore the landscape of the book graph. The system will thus allow users to discover new books based on other books. After finishing a book, an avid reader often asks "What should I read next?" or "What are similar books?" The motivation behind Project Alexandria is to answer these questions for the user by easing the process of discovering new books.

2.2 Project Specifications

In the early stages of the project we specified certain criteria that Project Alexandria should meet in order for the project to be deemed a success. These criteria for success can be broken down into two categories: required user functionality on our site and the system's ability to scale to a heavy load. We quantify these requirements in this section.

2.2.1 *User Interaction Specifications*

We required that users must be able to interact with the system in the form of searching for book titles and clicking on nodes in our graph. The provided book title may not be exactly correct so the system must attempt to match the provided string with a book in the database. Furthermore, upon visualizing the graph, the user must be able to click on a graph node to expand the search results. Using only these simple inputs, the user should be able to interact with our web interface and explore connections between books. Through these interactions, the user should see a visual map with books as nodes and links connecting these nodes which describe why the books are similar as a user hovers over them. These outputs would be updated after every user input and the visual expands with new books as the user clicks to explore different areas of the graph. Our interface specifications can be verified through the web interface.

2.2.2 *Scalability Quantification*

In addition to qualitative, functional specifications, we set several quantitative performance specifications that Project Alexandria must meet. When researching common performance criteria for consumer-facing software systems, we found that the system response time to inputs and the site's scalability are the most important factors for a successful product [1]. We especially focus on those criteria when quantifying our performance specifications.

Specifying a bound on the time it takes to load and modify the graph makes our site more usable. The initial search results must be returned in under five seconds, and when a user clicks on one of the nodes of the graph, the graph must update with new connections in less than two seconds. Our specifications were chosen based on the complexity (and thus the feasibility) of the action and what would be reasonable for a user. Following these specifications will ensure that users do not grow impatient and leave the site while waiting for results. Maintaining a high level of responsiveness for our system ensures that end users will have an enjoyable experience.

In addition to being quick and responsive, the system must be scalable. In particular, for a given user, the UI must support up to twenty nodes displayed at once. Seeing this many nodes simultaneously both provides a better visual experience for the user and allows them to observe more complex relationships between books. Furthermore, the web interface must be scalable across users. The backend infrastructure must be able to support up to ten users and searches simultaneously. A web interface that cannot support simultaneous users would always crash as users compete to access the site. We also specified that graph generation must take less than 24 hours per feature to allow for a reasonable development and testing cycle. These specifications ensure both that we would be able to complete the project on time and that the end product would be usable.

3.0 DESIGN PROBLEM SOLUTION

The process required to generate similar books and display them to the user was split into four major components: features, which are functions that provide a descriptive and quantifiable similarity between pairs of books; graph generation, which is the process of identifying similar books and then efficiently storing those links; website backend, which serves the information

that will be presented to the user; and the user interface, which allows users to easily and intuitively browse our final book graph on a web application. Figure 1 is a block diagram that illustrates these components and highlights which ones happen in real-time when someone uses the website. Each of these components is separate but provides inputs to other components. As shown in Figure 2, the features component provides the similarity measures used by the graph generation component to find similar books. The graph generation component then provides the book graph that the website backend serves to the UI, which uses the information to display the landscape of similar books to a user. This pipeline is both efficient and modular, allowing us to easily improve any singular component.

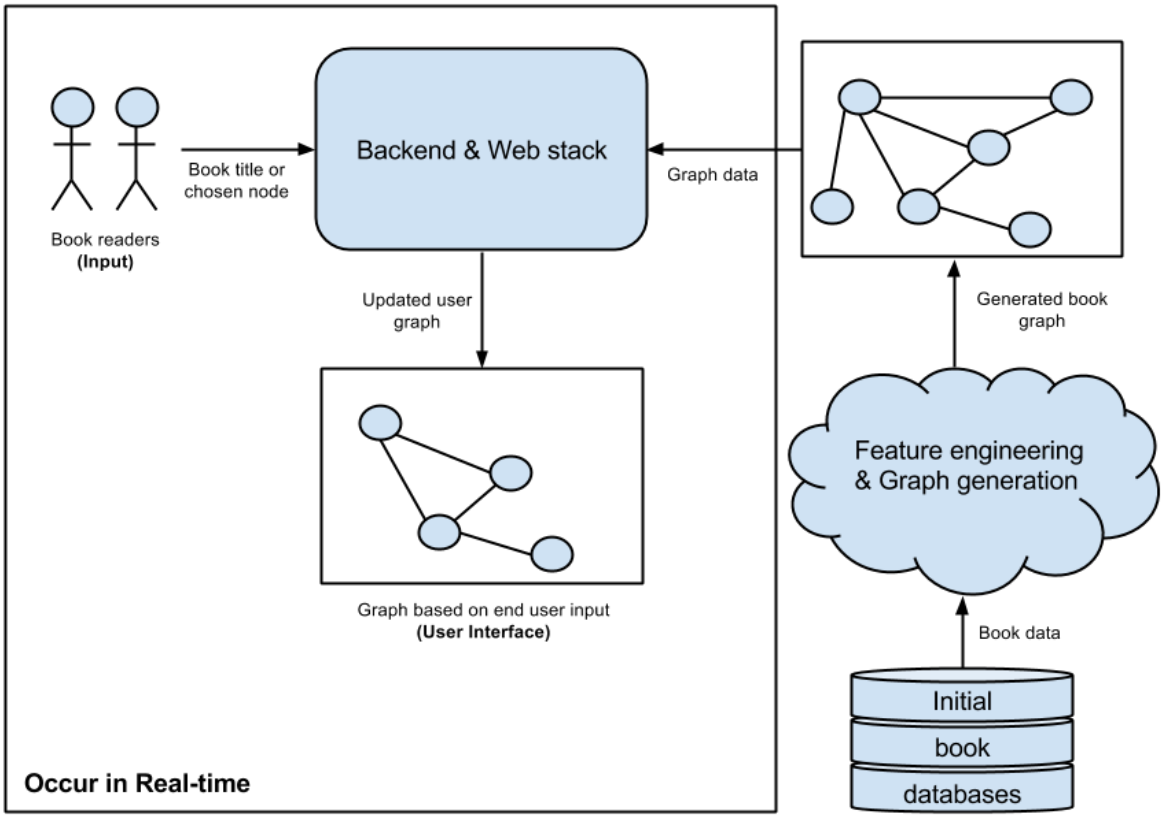

Figure 1. System Block Diagram

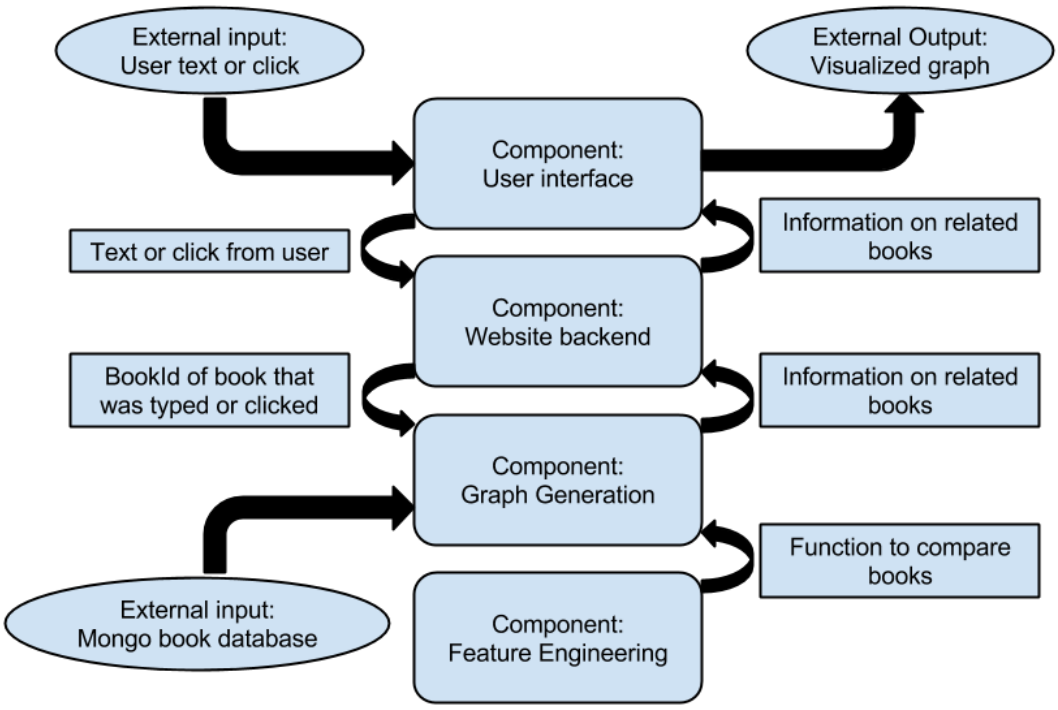

Figure 2. System and Component I/O Specification

3.1 Features

At the core of our project is a set of functions that compare two books and give a score representing their similarity. Each function compares books based on a different metric, and we designed various features that, when combined, capture many of the ways that books are related. Each feature complements the limitations of other features, providing a comprehensive analysis. In the following subsections, we discuss the features used in the system: genre, book descriptions, similarity of authors, expert opinions, sentiment analysis, readability, and word choice.

3.1.1 *Genre*

Our first feature compares the genres of two books. Genre is very important to readers; when people describe books, it is almost always one of the first things to be mentioned, and readers tend to prefer certain genres. For example, fans of one science fiction book will probably like other science fiction books. The genre feature makes use of BISAC codes given to us by Zola

Books. BISAC codes are nine-digit alphanumeric codes that describe one general genre and up to three sub-genres, and a given book may have multiple codes. To compare two books, the feature evaluates the similarity between the two sets of codes. The more overlap between the BISAC codes, the higher the score. As fewer BISAC codes overlap, the score goes down. By itself, this feature is effective in creating broad groups of related books. However, other features are needed to generate more detailed connections.

3.1.2 *Book Descriptions*

Book descriptions is another feature that we use to compare topics of books. Since we have book descriptions for over 90% of our books, this feature can be run on almost all of our data set and adds cross-genre similarities. While genre would match a historical fiction novel about a family moving to America with other historical fiction books, our book descriptions feature might match the historical fiction novel with fantasy novels or nonfiction books that discuss immigration or starting a new life. To implement this feature, we used a technique called Latent Semantic Indexing (LSI). LSI works by finding patterns of words that occur together among book descriptions, scoring books based on how well they match each pattern, and then giving a final number based on whether or not books have similar scores for each pattern. In general, this feature matches books about similar subjects. However, many books have generic descriptions such as "English Reprint," which can give meaningless matches and degrade this feature's effectiveness.

3.1.3 *Author Similarity*

One effective way to find similar books is by looking at the authors who wrote them. We quantify the similarity between two authors by comparing how much they have influenced or been influenced by other people. In order to calculate this measure, we used data from a site called DBpedia, which is an easy-to-parse extension of Wikipedia [2]. DBpedia has pages for a number of famous authors and includes fields like "Birth date," "Notable work," and "Family members." We used the "Influenced" and "Influenced by" fields to create a map of which authors were influenced by each other. Two books were connected if their authors were neighbors or shared first or second-hop neighbors on this map. For example, The Hobbit would be linked to A Game of Thrones because J. R. R. Tolkien and George R. R. Martin are both in a

neighborhood of fantasy authors influenced by each other. Because the data for this feature is hand-curated and reviewed by a number of Wikipedia editors, these links are very good. However, this feature is limited to books by well-known authors and cannot differentiate between books written by the same author.

3.1.4 *Expert Opinions*

Another feature that works well with popular books is our "expert opinion" feature. The data we received from Zola Books contained several hundred hand-generated lists of related books, connecting about fifty thousand thousand unique books by expert opinions. These books comprised only about 5% of our books. However, because these opinions are hand-generated, they contain some of the most popular books searched. Expert opinion connections thus provide a depth of knowledge that our other algorithms sometimes struggle with.

3.1.5 *Sentiment Analysis*

The features described above rely only on book metadata rather than book full text. When we had access to full book text we used it to create features that make deep connections between books. Though we are only able to run such features for books in the public domain, typically books older than 70 years, these features yield some of the strongest connections. Furthermore, an e-book company like Zola Books, which has full text for all of their books, could use these features on many more books.

Sentiment Analysis aims to graph the plots of novels and then compare the plots of different books. We create the graphs using tools created by a University of Nebraska professor named Matthew Jockers that use emotion as a basis for determining plot [3]. Using the tools, each book was first broken into sentences. Each sentence was then assigned an emotional value based off the number of positive words minus the number of negative words in the sentence, where the positive or negative categorizations came from values provided by Dr. Jockers. The emotional values were then plotted on a graph and passed through a low pass filter to smooth out small variations, resulting in a final graph of emotions over time similar to the one shown in Figure 3.

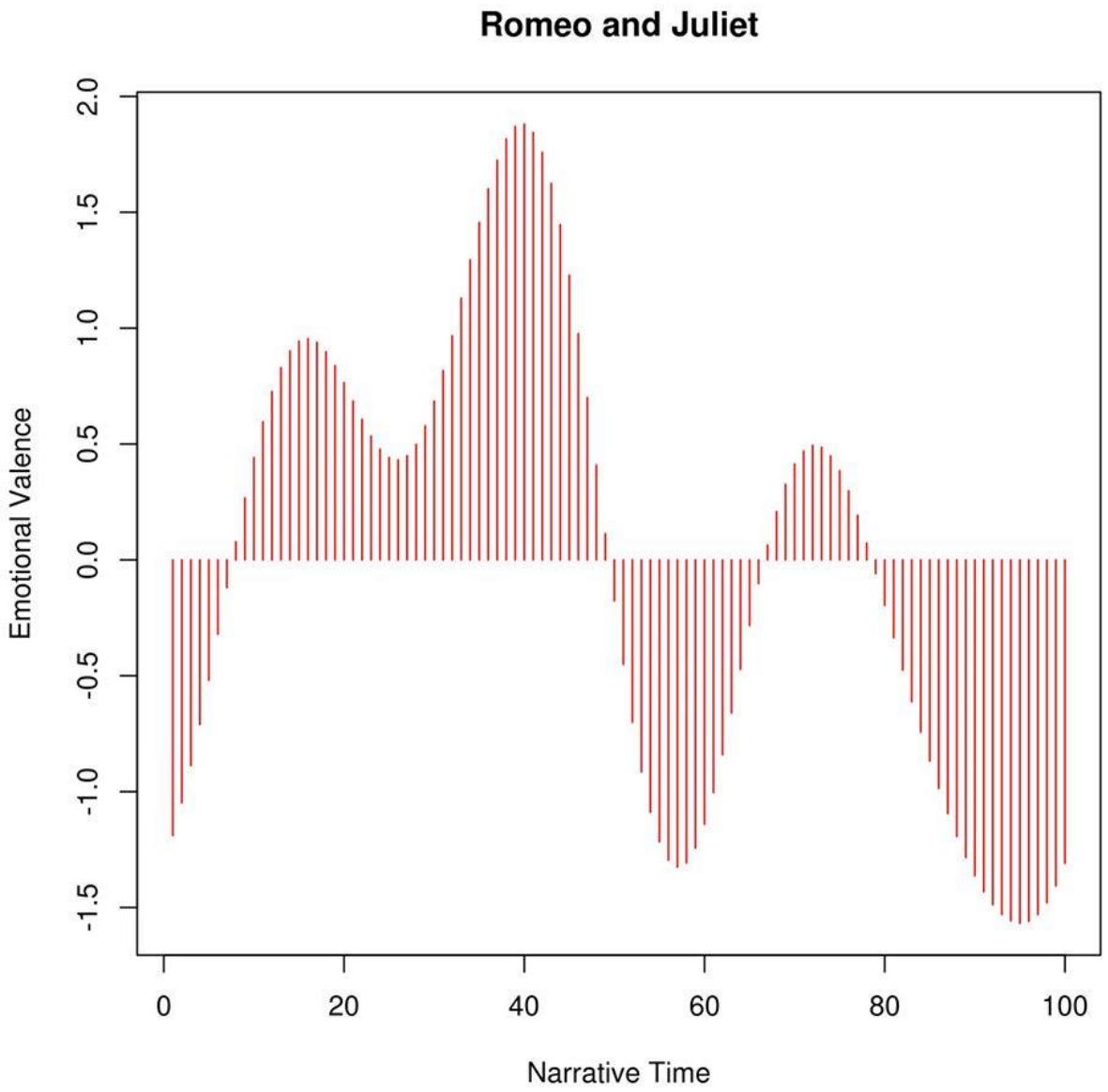

Figure 3. Example Sentiment Graph

Once the sentiment graph for each book was created, we created a function to compare books. We chose five characteristics of emotional graphs to use for graph comparisons. These characteristics are whether the book starts and ends happily or in tragedy; the number of transitions from positive to negative emotions, which indicates the pace of the book, because fast-paced page-turners have more emotional transitions than slow paced novels; the mean value of the emotions, which determines whether the book's overall mood is sad and dark or light and cheery; and the amplitude of the emotional transitions. After these characteristics are calculated

for each book, we can compare these values in the similarity function.With these five characteristics, we have an extremely strong metric to rate similarities in the plots of two books for which we have full text.

3.1.6 *Readability*

Another feature that uses the full text of books is readability analysis. Readability is an important criteria because people will often not read a book that is either too simple or difficult for them. To calculate reading level, we pass each book to a set of standard readability functions such as the Flesch-Kincaid algorithm and then average the results together. This feature does well at differentiating children's books from adult books.

3.1.7 *Word Choice*

If we do not have full text for a book, we can gain insight about the book using n-gram data. N-gram data, provided by Zola Books, contains the frequency of each two-or-three-word phrase (called an n-gram) that appears in each book. By splitting n-grams into words, we determined the frequency that each word was used in each book. Each word was given a weight proportional to the frequency of the word in the book and inversely proportional to the percentage of all books that contain the word. This weighting finds unique, commonly-used words for each book while penalizing commonly used words like "the". The n-grams feature calculates the percentage overlap of the fifty highest-weighted words between books. This feature supplements the genre and book description features well by giving a sense of the language that an author uses.

3.2 Graph Generation

Creating features to compare books is only the first step; we must then use each feature to find the most similar neighbors for each book. First, we found, for each book, the most similar books with regards to each feature. Next, we merged those nearest neighbor graphs into a combined graph with scores from each feature for each neighbor that was connected by at least one feature. Finally, these scores were appropriately weighted to find a diverse set of similar books to show to the user.

3.2.1 *Creating the Nearest Neighbor Graphs*

The step of constructing the nearest neighbor graphs for each feature was solved by implementing a K-nearest neighbor graph (KNN graph) algorithm called Nearest Neighbor Descent (NND). This algorithm, when given a similarity function from a feature along with a list of books, produces a graph with the most similar books to a given book as neighbors. While many other algorithms exist to solve the KNN graph problem, we chose NND due to its ability to be run on all of our features and its high speed [4].

Once a KNN graph was computed for each feature, we merged the separate graphs into one combined nearest neighbor graph. This combined graph contained all neighbors from all KNN graphs for a given book. Since each KNN graph only contained similarity scores for the graph's respective feature, we then filled in missing similarity scores in the combined graph. The tools developed to create these book graphs are general enough to run on any feature with just one function call.

3.2.2 *Weighting Scores to Find Neighbors*

Once we created the combined graph of neighbors, the next step was to use those neighbors' scores to find the five books to display to the user. In order to maximize the variety of our results, we designed an algorithm which dynamically changes the importance of various features to ensure that a specific feature does not dominate the results. First, the top neighbor is determined by an initial weighted sum of feature scores. Next, the weights that contributed most heavily to this top neighbor are decreased, and the next neighbor is chosen with the new highest weighted sum. The process continues until the five most similar books are found. This process was not precomputed for each book but instead computed at runtime on the website to allow greater flexibility and extensibility.

3.3 Website Backend

In order to build a website that made use of our generated graphs, we needed several web technologies. Our two main needs were a database that could store our book graphs and a web framework that could serve as the middleware between our UI and our database.

3.3.1 *MongoDB*

Project Alexandria required a number of databases to store our data sets and save our book similarities. To organize this data, Project Alexandria uses MongoDB, a flat database that works as a jack of all trades. MongoDB stores the data we received from Zola Books, DBpedia, and Project Gutenberg. We also use it to store our final graph data structure. Finally, we created a separate database specifically for book search and autocomplete. This data, which must be optimized for speedy lookups, is also managed by MongoDB.

3.3.2 *Django*

To act as a bridge between our MongoDB database and our user interface, we needed a framework that could manage communication between the two. To perform this task we use Django, a Python server-side framework that also renders the website. This framework manages the functionality of the URL routes for our website. Django takes requests in the form of a URL and returns a response by querying our graph database for data, processing that data, and then serving it to the front end UI. The main routes that Django manages are the main website page route, the autocomplete route, and our book ID query route. The main website page route renders the page a user sees when they navigate to our website. The autocomplete route returns results when a user tries to search for a book, and the book ID query route returns a book and its neighbors so that they can be rendered on the book graph UI.

3.4 User Interface

In order to build a fully functional website, we needed to create a front end UI that was visible to a user. Our main priority was to design an interface that allowed for easy exploration of a visual book graph. To do this, we first defined the actions a user can take on our website, and the information that must be presented to them. We then needed a graphing library that could represent our book data, a styling library to make our UI look visually appealing, and a more comprehensive library to handle events and requests to our server on the back end.

3.4.1 *User Actions and Capabilities*

The key to our user interface is simplicity. When a user navigates to our website, the first thing he or she sees is a blank graph and a sidebar with a search box. The user can then begin typing a

title or author to search for a book and an autocomplete menu will drop down to assist with the search. When a user selects a book, the graph is populated with the selected book and its neighbors and the book's information shows up under the "main result" section of the sidebar. The book information includes a cover photo, title, author, and a description. This info changes when a user searches for a new book or clicks a node in the graph. If a user hovers over a node on the graph, its information will appear in the sidebar under the "secondary result" section. A user may also hover over an edge to see why two books are related. In order to facilitate easy exploration of the book graph, we added three features: zooming, panning, and auto-centering. When a user clicks on a node in the graph, its neighbors will be shown, and the graph will automatically re-center on that node. Also, if the graph becomes crowded, a user may zoom out and pan around the graph to find different sections of it they may want to explore. One of the final features we added to our site was a helpful how-to menu and an about menu so that users may learn more about the project. We feel that this approach is simple and easy to understand, but our non-intrusive hints provide an effective safety net in case of confusion.

3.4.2 *D3.js*

To create the graph interface on our website, we used D3.js, a JavaScript library. In order to render the graph, we have written a number of functions that use D3.js to perform various actions such as adding or removing nodes from the graph. The primary functions we incorporated are zooming, panning, and expansion of the graph by clicking on nodes, and creating an entirely new graph. We also created auto centering functionality for when a user expands the graph by clicking on a new node. This makes it easier to explore the graph without having to manually zoom and pan around. Towards the final weeks of our project, we did a lot of work to tweak the physics engine in D3.js and make the nodes look spaced out and to ensure that edges in the graph linked the correct books together.

3.4.3 *jQuery and jQuery-UI*

We used jQuery and jQuery-UI, both of which are JavaScript libraries, to facilitate communication between our Django server and our UI, handle effects, and render elements in the sidebar of our UI. We have also written a number of functions that use jQuery to handle certain events on our website such as searching for a book or clicking a node as well as sending queries

to our Django server. The queries sent to our Django server include autocomplete queries to suggest books a user might be trying to find and book id queries to find a book and its neighbors for rendering on our visual book graph with D3.js. jQuery also handles events such as clicking or hovering on a node, hitting enter to complete a search, or clicking read more to expand a book description. jQuery works with Bootstrap to launch the "about" and "how to" modals. jQuery-UI is used to render our autocomplete menu for searches and to create popups for when a user hovers over an edge in the graph.

3.4.4 *Bootstrap*

For styling and UI effects, we used Bootstrap, a CSS and JavaScript library. The library comes with a variety of icons, CSS classes, and various JavaScript functionality such as tooltips and popovers. The main JavaScript functionality of this library is for rendering the about and how to modals. We largely use Bootstrap for styling purposes, as it includes a large base of CSS code that helps with the spacing of elements, font styling, page layout, and form elements. In order to supplement Bootstrap, we added our own CSS classes. For instance, instead of using Bootstrap's responsive grid layout, we instead chose to use a fixed width sidebar and graph. By design, our site was never meant to be mobile-friendly, and we made our styling decisions to optimize the site for larger displays. We also chose to add a few fonts from Google WebFonts and we chose the color palette from Google's Material Design colors. These additions round out our simple and clean UI design.

4.0 DESIGN IMPLEMENTATION

The design described in the previous section was not static; numerous changes were made throughout the project as we learned more about available data and web development. Due to the subjective nature of developing features to compare books, much of the time developing features was spent in brainstorming and prototyping potential features. This section details our general process for building new features and explains the intuition behind each completed feature. The graph generation problem was more defined, and so the main hurdle was optimizing the algorithm and the code as much as possible so that it could run each feature on the full set of the data in a reasonable time-frame. The website backend remained the same as the initial design because of data format constraints. However, the technologies we used for the user interface

changed because we found libraries that performed some of the necessary (and complicated) functionality for the dynamic graph user interface. These design changes were all motivated by system performance and implementation feasibility rather than cost, as the project cost was negligible and unaffected by design decisions.

4.1 Features

One of the large hurdles in the project was deciding which features to develop and then which features to include in the final product. From the very beginning, we realized that we would run into three issues with comparing books: the subjective nature of such comparisons, technical challenges, and limited data. Subjective issues mainly involved understanding how people choose what books to read and how books can be compared. Technical challenges comprised of figuring out how to implement a feature once we knew what we wanted to implement. Finally our data limitations came from incomplete and non-comprehensive information from our Zola Books dataset. These challenges influenced how we chose to proceed with developing features.

In order to decide which features to develop, we first tried to identify gaps in our feature set to understand how a human might compare books in a way our existing features could not. We then came up with various features that could fill those gaps. After brainstorming several ideas, we informally evaluated how well each potential feature would solve our 3 issues: the subjective problem, how much effort it would take to overcome the technical challenge, and whether it would be possible to deal with our data limitations. We then spent time prototyping our most promising features. If the prototype progressed easily and produced reasonable results, we polished and completed the feature. If not, we abandoned that feature and returned to step one. This development process occurred for multiple features in parallel, as at least two members of the team were devoted to feature engineering at all times.

The first feature we completed was our genre feature, which was useful to readers, technically simple to implement, and free from data issues. Our book description feature followed shortly after, because while we had good data and knew it would be useful, we had to spend significant effort researching, understanding, and implementing language processing techniques to process book descriptions. We tried two similar methods, Latent Dirichlet Allocation (LDA) and LSI.

LSI was chosen because its results were qualitatively better. Next, we finished developing the expert opinions feature, since there was a straightforward technical implementation, but it had limited use because we only had data for about five percent of our books.

After implementing these features, we realized that the current feature set was missing any reference to authors. We also realized that the author of a book heavily influenced whether a person will read a given book. We built the author similarity function after overcoming a technical challenge by collecting more data. The data given to us by Zola Books contained author biographical information, but we could not figure out an effective way to compare authors using such information. By finding external sources of data (DBpedia, in this case), we could get the data needed for this feature, and we built the author influence feature.

After building the genre, authors, and description features, we observed that none of the features took into account the content of books. We thought that developing such features was not only useful but was also interesting technically, and going through the process would be a good educational exercise. We eventually completed three features: sentiment analysis, readability, and word choice, to fill this gap. We faced multiple difficulties with each function. Sentiment Analysis took a lot of effort to research and implement and only worked for full text books. Readability was quick to implement but we felt it was only moderately consequential and once again needed full text. Our word choice feature did not need full text, but it was challenging to implement and we were initially unsure of how much sense the results would make. However these features all had benefits that made them higher priority than other options, and we successfully completed these features.

Not all features were as successful as our final features, and we abandoned several features that were either too difficult to implement or did not produce qualitatively good results. We spent significant time on failed features that would compare books' writing styles. One such feature was one internally named "Text Rarity" that matched books in a way we thought would represent writing style. Unfortunately, it was difficult to determine whether or not the results were reasonable because writing style is very difficult to judge, and we could only run the feature for books for which we have full text. Group members had not recently read many of the

books in the public domain, and we could not make informed decisions about the prototype's reasonability. The feature was abandoned in favor of other features that process the full text of books. We also abandoned a feature built around information such publication location and date. Future extensions of this project may attempt to finish and integrate these features.

4.2 Graph Generation

The central fixture of the graph generation component was the NND algorithm, which was chosen due to its speed and accuracy. Because we needed to process over one million books, our KNN graph algorithm was required to efficiently process data sets of that size. While the brute force approach to the KNN graph problem produces optimal results, its time complexity is very high, with a predicted running time of over six months on each feature as shown in Figure X. This time constraint prevented us from using the brute force algorithm on our full set of books. NND was able to create the KNN graph for each feature in a matter of hours without differing from brute force results by more than 3%.

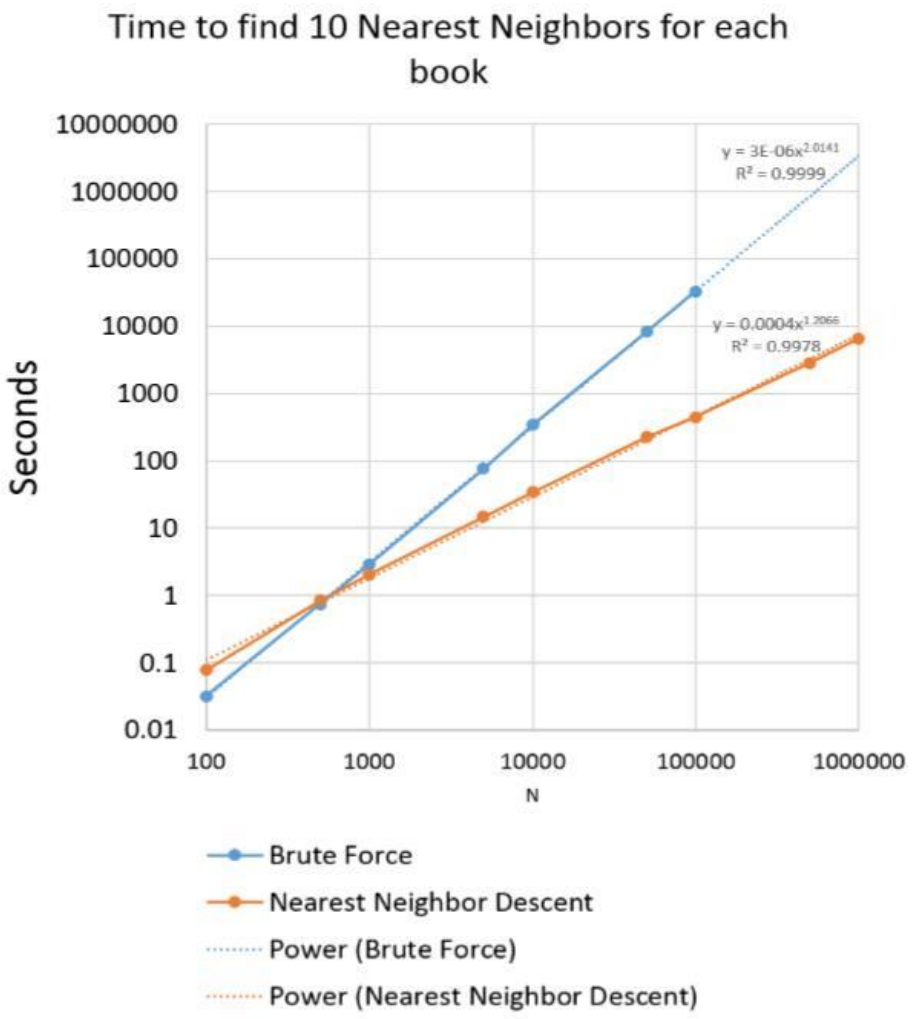

Figure 4. Comparison of NND vs Brute Force

Our implementation of the NND algorithm went through many iterations to improve its speed and accuracy to the point where it could be run on our data. The first iteration of NND naively implemented the algorithm from the paper by Dong et al. [4]. However, this implementation could result in extremely high running time on certain features due to the unbounded number of reverse edges that could point to any given book. An attempt to resolve this issue by only considering the most similar reverse edge resulted in a drastic loss of accuracy. Finally, a more moderate solution that randomly sampled reverse edges when there were a high number of them allowed us to reach our target parameters for both speed and accuracy for NND.

While we initially planned to use a set of static weights for the final feature weighting required to determine the best neighbors for a given book, we instead switched to the dynamic algorithm described in the problem solution section. We decided that a static weighting system for features would inherently bias the system towards preferring certain features, and it was unclear how to handle cases where multiple sets of static weights preferred the same books. The algorithm we designed avoids both issues by encouraging variety no matter which features are present for books and providing one unified set of top neighbors instead of one for each set of weights.

4.3 Website Backend

The backend web technologies remained the same throughout the project. We chose to use Django for our website's framework for a few reasons. The fact that Django uses Python, making it easy for most of our team to work with it because most of us are familiar with the language. Python is faster than other web frameworks such as those written in PHP but slower than newer server side frameworks written in JavaScript or Java. We found that Django was fast enough for our needs while being more accessible and better documented than other web frameworks. We chose to use MongoDB because we were given our book data in a format that was easily transferred to a MongoDB database. MongoDB also met our needs for clear documentation and fast read and write times for our data. The suite of technologies we chose performed as well as we expected and allowed many of our team members to contribute to the website code, which made completing our project on time feasible.

4.4 User Interface

One of our initial plans for our website was to create the visual book graph from the ground up. We first attempted to use HTML5 Canvas and a library called KineticJS to code our own node and edge animations and draw all of the shapes ourselves using a library of functions. This proved to be much too involved, and we looked elsewhere for more comprehensive libraries that we could use. We eventually found Sigma.js and D3.js and decided to go with D3.js for a few reasons. D3.js was a much more developed library and seemed to have more capabilities. D3.js also had more documentation since it has been around longer than Sigma.js. We believe we made the right choice in choosing D3.js as our graphing library because it has allowed us to build all the features we needed with relative ease. jQuery and jQuery-UI were the obvious JavaScript

library choices because they are the most widely used and documented libraries for handling events, ajax queries, and UI effects. Bootstrap is also the most widely used CSS framework and created a good foundation for the styling of our website. We believe our UI allows for simple and intuitive exploration of our book data through the use of a visual graph and a sidebar of relevant information.

5.0 TEST AND EVALUATION

We separately tested the primary components of our project: features, graph generation, and the website, and then conducted system-wide tests. In this section, we detail these tests. Features were largely tested qualitatively through manual inspection, but we developed scripts to prevent sample bias in our qualitative inspections. The graph generation component was tested for both accuracy and code performance. The UI and website backend were tested together, both qualitatively and quantitatively. We've concluded that each sub-system and the overall project met all the specifications and requirements we set in the Design Implementation Plan.

5.1 Features

Two questions must be answered to evaluate each similarity function. First: did the function itself behave as described? Second: did the function produce reasonable related books? To answer the first question, we wrote unit tests for each similarity function. The tests provided inputs to the function and then determined whether the resulting output matched an expected value that we had already computed. These unit tests were also run after any modifications to a similarity function to ensure that those modifications had not unintentionally changed its underlying behavior. To address the second question, we created a script capable of running a similarity function on a small, random subset of our books and produce the top related neighbors for those books. We then manually inspected those neighbors and made a qualitative assessment as to whether the neighbors could be considered closely related to each book. Performing these tests on a small, random subset of books allowed us to draw reasonable conclusions about the overall dataset while still being able to test our results quickly and make changes to the similarity function as necessary.

5.2 Graph Generation

For graph generation, the team had to ensure that the algorithm we used to generate the KNN graph produced results that were similar to the results that a brute-force algorithm would produce, along with whether the algorithm could be run in under 24 hours. These specifications were quantified in the Test and Evaluation Plan. Testing the correctness of the NND algorithm involved using a metric termed "recall" by the authors of NND. "Recall" is the percent overlap of the graph produced by NND on a given dataset when compared to the graph produced by a brute-force algorithm on the same dataset. A recall of 1.0 indicates that the two graphs are identical, which is ideal. Because calculating recall requires running both the NND and brute-force algorithms, it is unfeasible to determine the recall for our entire set of books. Thus, we again tested a small, random subset of books for each feature in order to determine the recall of our implementation of NND. In our specifications for the project, we established a recall of 0.9 as the minimum required for each feature. As can be seen in Table 1, we achieved that goal for every similarity function, with one even achieving a recall of 1.0.

Table 1. Recall for Each Feature

Feature	Recall
Genre	0.998
Book Descriptions	0.977
Author Similarity	0.995
Expert Opinions	1.000
Word Choice	0.975

Testing the performance of the graph generation was more straightforward. We simply used the Linux "time" utility when executing the algorithm for each feature to track how much time the algorithm took to run on the full book dataset. Almost all of our features required less than four hours to generate their KNN graph, significantly less than the constraint of 24 hours that we set for ourselves in our test and evaluation plan. One notable exception was the "book description" feature which required roughly sixteen hours to run; however, this time was still well within our time constraint. Table 2 contains the time it took to run graph generation for each feature.

Table 2. Time Required to Run Each Feature

Feature	Time (hours)
Genre	3
Book Descriptions	16
Author Similarity	0.8
Expert Opinions	1
Word Choice	2.2

5.3 Website Backend and UI

Lastly, we quantitatively tested the performance of the website. In our Test and Evaluation Plan we established specifications for how quickly the website would respond to different types of user interactions, specifically searching for books and clicking nodes on the graph interface. To measure the response time of the website when performing both of those actions, we used the "Network" utility in the Chrome web browser's developer tools. The utility lists the time each query to a website takes to complete. To test searching, we searched for ten simple words and calculated the average of the resulting response times, shown in Table 3 below. We used a similar approach for testing node clicks, using ten random book IDs as input. The results for both metrics were nearly instantaneous, well below our goals of five seconds and two seconds for searching and clicking nodes respectively. We also ensured that interacting with the graph interface resulted in correct behavior: clicking a node added neighboring nodes to it, hovering over a node caused its information to appear in the sidebar, and hovering over an edge showed an explanation for that edge's similarity in a tooltip.

Table 3. Response Times

Query Type	Average Response Time (ms)
Book ID	175
Search	261

6.0 TIME AND COST CONSIDERATIONS

Due to the strict final deadline that our project had to adhere to (in the form of Open House), time was a factor that the team had to constantly consider when planning the direction of Project Alexandria. We hit a number of challenges and delays, especially when developing features, but ultimately ended up finishing before Open House. Costs, on the other hand, were always minimal and stayed well within budget throughout the semester.

Project Alexandria met the primary schedule constraint of meeting all design requirements by Open House on April 29th. However, some mid-semester milestones were not fully met due to unforeseen complications. Our first end-to-end demonstration milestone, scheduled for February 11th, was successfully met with a bare-bones UI demonstration using data from a limited set of books. Our second milestone, consisting of a fully-functional UI with search functionality and clickable nodes, was met on time on March 4th. However, we continued to refine and improve the functionality of the UI until the final demonstration at the open house. Our third milestone required a feature set of at least five features, run on at least 1 million books, by March 18th. While we met the goal of running features on 1 million books by this date, only the genre and description features were complete because building effective features took longer than we predicted. To get back on track, we dedicated an extra team member over the following month to completing and testing five additional features. Fortunately, in Project Alexandria's original timeline, the month leading up to open house on April 29th was allocated to any necessary remaining work. Our team was able to use this time to complete all remaining features and polish the UI by Monday, April 27th, two days before the deadline.

Project Alexandria is an entirely software-based project, so we were able to keep our costs low by exclusively using free open source software like Django and MongoDB. The only cost that we incurred while working on Project Alexandria was the cost of the Microsoft Azure cloud-based server on which we hosted the website and databases. This service usually costs about $100 per month. Fortunately, one of our group members received, for free, a year of Azure credit worth $150 per month due to an internship. The cost of the server only rose above our monthly credit budget of $150 once: in mid-April we mistakenly kept our server in a higher performing hardware configuration for longer than we intended, resulting in the server being automatically shut off once we prematurely reached our spending cap of $150 that month. Fortunately, this occurred just two days before the end of the billing period, so we only needed to pay $11.84 to run the server for those two days before our credit was replenished. The project did not have any other costs.

7.0 SAFETY AND ETHICAL ASPECTS OF DESIGN

Although our project did not have any significant physical safety considerations due to its purely software nature, we still had to consider data protection while developing Project Alexandria. Much of the book data provided to us by Zola Books was confidential and protected under a non-disclosure agreement, and accidentally leaking the data publicly would have led to significant financial costs and legal troubles. We took great care in ensuring that the data and the server on which it was hosted were secure. We designed our web interface such that it did not expose any confidential information to users of Project Alexandria, and we kept all of our server's software up to date to eliminate any security vulnerabilities.

As for our project's contribution to the public interest, we feel that creating a product that enables and encourages others to read is a noble pursuit that will hopefully have a positive impact on people around the world.

8.0 RECOMMENDATIONS

We hope to see Project Alexandria become a powerful research tool that any avid book reader, researcher, student, or teacher could use. Our team has identified two pieces of work that would

improve the site's functionality and user experience. The first potential improvement would be to add more focused search functionality. We would like a user to be able to search for "Dystopian novels from the 1980s" or "Historical fiction from the 2000s." This kind of search functionality could make Project Alexandria an extremely powerful book search and research tool. Enabling this kind of search involves indexing all the book metadata to make it searchable and adding a button to switch between the existing book search mode and the new focused search mode.

Another potential improvement that could drastically improve the user experience on the site is to allow for more personalized graphs to be displayed. Fundamentally, similarity between books is a very subjective measure so creating a system that learns what kinds of links a user prefers and offering him or her a more tailored graph would better suit the user's needs. The current system displays the same neighbors for each book regardless of a user's history. As a user searches for different books and clicks on different nodes, our system could learn what kinds of links dominate the user's exploration. Similarly, the existing system has no form of user feedback. Asking the user "Was this a good connection?" would give the system explicit feedback from the user that could help improve connections or personalize the graph being displayed.

Ultimately, the future of Project Alexandria may lay in the hands of Zola Books. Our team is meeting with them later this week to give them a demonstration and see if they may be interested in continuing the project. Fortunately, our email exchanges with Zola Books have elicited enthusiastic responses, so our team is looking forward to seeing how the meeting will go.

9.0 CONCLUSION

In this paper, we discussed each of the features we used, how we generated neighbors for all million books in the database, the backend web technologies, and the intuitive graph interface of Project Alexandria. These subsystems were not static, and in this paper we described the process of prototyping potential features, optimizing graph generation, and choosing the appropriate web technologies. We also detailed in this paper the the unit and system level tests on each sub-system to ensure quality of connections and responsiveness of the site. The team completed the project with a negligible monetary cost and and successfully managed to secure the data and

honor the non-disclosure agreement we signed. Our paper also identified future work for Project Alexandria. In particular, we indicated that an advanced search functionality could improve the usefulness of the site for specialized purposes. We also highlighted the potential of taking into account human interaction to customize and improve connections. We will meet with Zola Books to discuss these extensions and Project Alexandria's future.

Project Alexandria has been an unequivocal success. The connections that it generates are both non-trivial and reasonable, and we would be comfortable launching our product for the public. We exceeded all project parameters, including the number and quality of features used to connect books and the performance of the website. Our tests, personal experiences, and audience feedback all show that we have successfully developed a usable book exploration engine that provides recommendations on par with GoodReads and Amazon. Our website is live and available at http://seniordesign.cloudapp.net.

REFERENCES

[1] P. Broadwell, "Response Time as a Performability Metric for Online Services," 2004.[Online]. Available: http://roc.cs.berkeley.edu/papers/csd-04-1324.pdf.

[2] Jens Lehmann, Robert Isele, Max Jakob, Anja Jentzsch, Dimitris Kontokostas, Pablo N. Mendes, Sebastian Hellmann, Mohamed Morsey, Patrick van Kleef, Sören Auer, Christian Bizer: DBpedia – A Large-scale, Multilingual Knowledge Base Extracted from Wikipedia. Semantic Web Journal, Vol. 6 No. 2, pp 167–195, 2015.

[3] M. Jockers, Syuzhet: Extract Sentiment and Plot Arcs from Text. 2015.

[4] Wei Dong, Charikar Moses, and Kai Li, “Efficient k-nearest neighbor graph construction for generic similarity measures,” In Proceedings of the 20th international conference on World wide web (WWW '11). ACM, New York, NY, USA, 577-586.