BookWorm

Software Engineering Spring 2019

Group #16 Report 3

May 5, 2019

Group Members:
Vedanta Dhobley: vjd41@scarletmail.rutgers.edu
Avani Bhardwaj: ab1572@scarletmail.rutgers.edu
Shazidul Islam: si194@scarletmail.rutgers.edu
Akshat Shah: avs91@scarletmail.rutgers.edu
Kutay Kerimoglu: kk851@scarletmail.rutgers.edu
Alan Patel: akp122@scarletmail.rutgers.edu
Anthony Matos: amm720@scarletmail.rutgers.edu
Joel Cruz: jc2125@scarletmail.rutgers.edu

URL of our projects web-site:
Github Link: https://github.com/vedantadhobley/SoftwareEngineeringProject2019
We will be using this repository to commit all changes to our project. All form of communication will be through weekly group meetings, messages, and smaller group meetings where smaller groups of people will meet to finish mini-projects.

Team Profile:

a. **Individual Qualifications and Strengths:**

Avani: C++, data organization, documentation, presentation, management
Vedanta: Java, Python, AWS, design, presentation, management
Shazidul: C++, Java, Python, SQL, design, management
Alan: Java, Python, SQL, design, data analysis, documentation
Kutay: C++, some Java, programming error checking, documentation
Anthony: Java, C++, presentation, documentation
Joel: C++, presentation, documentation
Akshat: C++, Java, Python, SQL, design, documentation

b. **Team Leader:** Vedanta Dhobley

Table Of Contents

1. Changes and Contributions

1.1 Individual Contributions

Shazidul Islam (si194): Helped with the creation of the Demo 1 PowerPoint presentation. In the presentation, Shazidul was the one that created the road maps as well as put together the User Case Scenarios, Backend reasoning, and the general idea of why our program is useful. Helped with the coding in the backend, which was considered but then not used due to finding a much more effective way of implementing the idea using cosine algo. Learned Python fully to make sure that I would be able to understand my teams code as well as implement ideas around pythons coding rules.

Joel Cruz (jc2125): Once provided with the road map of the presentation, I presented it for the demo, as a way to ease into the actual presentation before we actually addressed the intricate details of the project. I also made sure to understand all the moving parts of the project in preparation for questions from professor and our fellow TA's. I also had to learn Python during spring break in order to work on the tier system of our project, but by the time the demo had arrived the tier system wasn't complete and was not a usable factor of the demonstration. Therefore, it will be a part of the second demo, and will be the piece of our project who will provide users with a more accurate result of our findings. Along with Shazidul, Alan, and Vedanta, i am a part of the backend unit of our group.

Avani Bhardwaj (ab1572): Head of documentation and bug testing with Kutay. Helped manage the team as a whole with scheduling meetings, deadlines, and rehearsals. Took all the notes from each meeting for the documentation files. Worked with Kutay to do the documentation formatting. We both also worked on debugging the lines of book and author names in the excel spreadsheet where our database was located. I am learning a bit of HTML and CSS to make the visuals of our project look better when it is time to integrate the frontend and backend parts. I made the brochure and did the powerpoint slides with Shazidul where we worked on the Use Cases and presentation structure. I continued to do the documentation and bug testing from then onwards. For the second demo I did the documentation, and powerpoint/brochure and we all practiced the demo together afterwards.

Alan Patel (akp122): Researched many different recommendation algorithms to finally come upon the cosine similarity algorithm to compare tags and authors, which I then had to tweak around to work with more than one input. It took some simple linear algebra and python programming. My other teammates were trying to use a tag weighting system, but when I found out about this one I completely created new code that replaced the old program, which made our program give more accurate recommendations. I also am the team member that programmed all

python code which contained the flask web framework to communicate to the web page, the recommendation algorithm, and the outputted list of books. I also worked with with Shaz and Joel to create a tier system from the list of books that I have found using the cosine algo. While I was programming I would also concurrently type in the technical and user documentations to make sure someone else other than me could understand what I was programming and how to use it. I also found the database of 10000 books from goodreads, which I formatted. Along with Avani, I cleaned the dataset to help remove any anomalies in the dataset. I have also helped Akshat and Anthony with any errors they received while doing the front end programming because I have experience in HTML and CSS as well. I have also contributed to documentation by creating UML diagrams and other diagrams that correspond to the code.

Akshat Shah (avs91): In order to set up the frontend, I learned to launch Bootstrap with Anthony to put up the basic search bar at first. Afterward, I worked on styling the page to add the search bar in the center and added a responsive search button, which directs to the output web page. I made it user-friendly by putting the application name above the search bar. For the result page, I worked with Alan on the Python code to add the segment that communicates that takes the input from frontend, sends to the backend and sends the output back to the frontend. After initializing the Flask in Python to generate the result webpage, I worked on converting the output from series to string and displayed as list form. Additionally, I implemented the local server on the Python code that is executable remotely on any browser.

Anthony Matos(amm720): For the frontend component, I was responsible for initiating the bootstrap that would yield a responsive and user-friendly search bar. Using HTML, I created the search bar with a drop down list of possible inputs that the user might have had in mind. The dropdown list was able to contain every book in the database by copying the books array from a csv file that was converted to a JSON file. This array was passed into an autocomplete method in JavaScript which basically searched through the array and suggested some books after a few keys were typed. I revamped the search bar so that it could take three books where each book that was selected from the list would be seperated in their own tiles with an ‘x’ button. This made it easier for users to delete some inputs while leaving others instead of just backspacing a lot. To add to that, I spent hours trying to append a search button that would stay together with the search bar whenever I tried to center both components together while also matching their sizes. Not only was it important to keep it simple for the user to operate, I came up with a navy blue and orange color scheme coded in CSS to give an appealing visual for all users with different vision capabilities. Additionally, the local server was implemented in the python code by other teammates, namely Alan and Akshat, but i connected it to this frontend page using a form in HTML that takes the users input of the books’ names and passes them to the algorithm in the python code.

Kutay Kerimoglu (kk851): I helped Avani with all of the documentation for every week. I learned a bit of Python in order to help the backend and frontend team members with the bug testing. So far most of my contributions were in the documentation area and the demo materials preparation. I helped with the brochure and the powerpoint. I helped with the explanations of the use cases for the demo. I have worked on the reports and more importantly, I relay information between frontend and backend and help Avani take notes as to what we need for the documentation. My biggest role is to do the final product testing multitudes of times in order to ensure that our algorithms work properly. While the other members are doing their development, I keep track of what is going on so that I can properly incorporate it into the documentation and I will know what to do for the later parts of the project.

Vedanta Dhobley (vjd41): Developed the original Python script to index and sort books, returning identification values and tag data as an object class for quick and easy manipulation. This is the code that was used as the basis for retrieving book information to pass to our recommendation algorithm. When deciding on project specifics, I decided that our design should be user-less in order to allow for a simple input-output interaction with our frontend, defining the simple philosophy of our UI design and input format. I also incepted the idea of using a tier system to display our results, allowing users to see a statistically sorted list of books placed in categories based on similarity to the input. Using tag similarity as our method of comparing books was also my idea, as I realized that using book summaries or reviews would be extremely subjective, whereas the tags assigned to describe the facets of a book would be objective. During the presentation, I defined our project and explained the ideas that lead us to the product we aim to create, going in detail about our user interface, frontend, and backend. I am currently working on improving our database management and book retrieval, with plans to use book index data directly from the user input to produce an O(1) time for each book and tag retrieval. I am also currently working on a statistical sorting algorithm, where the output list of books along with the "points" they've each been assigned can be analyzed and segregated based on standard deviations.

1.2 Summary of Changes

For this report, we had been given feedback from Demo 2 which we are implementing into our project. We were given recommendations for changing up the algorithm, which our team is currently working on improving. As we move forward we want to highlight the important details for this product.

- The main reason we chose this is because we want to implement a recommendation algorithm which does not require users to make accounts or register any information.
- Our product is different than other available products in that we are able to add (currently) up to 3 books at once to find similar books for.
- Our product will essentially figure out who the user essentially is based on their search results.
    - For example, if all three books searched are children's books, the search will return books meant for children. If a search contained three books on computer science, the results would return books based on a more advanced level.
- We are changing the breadth first search that we had originally planned on doing into a hash map for easier book retrieval within the algorithm.
- We are also moving away from the tag system and going toward a system that implements the Cosine Similarity Algorithm to do the recommendations
- For our use cases, we were told that one of the cases was more like a "Step" rather than a Use Case. So we are implementing that and changing it to two use cases with an extra step in the beginning.

2. Customer Statement of Requirements

2.1 Problem Statement

A current issue at hand is that most "Recommendation" programs for books simply search with only author or genre tags when looking for books at libraries or bookstores. This leads to titles being recommended that are not similar to the person's liking or all recommendations which have one very odd tag match but don't have any other similarities to the first.

a. Example: If someone were to say they read a disney book, what can happen is that the recommendation algorithm will search for similar genre or author. However a better search would be Disney tag. Disney would return more suitable recommendations for the person that read something that was Disney related.
b. Another Example: Many who put that they have read Harry Potter, seem to get back a recommendation of Twilight due to it falling under the fantasy genre. However a series such as Percy Jackson would be a better match due to its use of magic along with fantasy, which is a more precious and intuitive approach.
c. Another bad recommendation would be when differentiating between what categorizes as 'historical fiction' and a simple history documentary. For example, a student who is trying to write a history paper on the roaring twenties prior to the Great Depression, they would most likely want to find articles and books with facts about this time period, instead of receiving a search result of 'The Great Gatsby.' Although the Great Gatsby was written in this time period, the novel is not focused on the general historical value. It was written as a fictional novel during that time. A student doing a report on American history would probably not want to waste time reading 'The Great Gatsby' because of the fictional aspect. The implemented algorithm would be able to differentiate between 'history' and historical fiction' in order to give the user a better search result with books of the appropriate tags.

The target for this program would be to entice avid readers that have issues with finding books similar to their taste, or those that simply do, can't decide that their taste is. With this algorithm, these readers can narrow their searches and find material that is much more relatable to the genre of books they have interest in. There is no age target since everyone at any age can enjoy reading! Another implementation we want to add to our product is actually to remove the use of needing user accounts. After a lot of research, we realized that every book recommendation does searches for only one book and also requires users to create an account for the website. We want to eliminate this aspect and also allow users to input multiple books into the search bar to find recommendations that suit them.

2.2 Proposed Solution:

For the program, there will be two algorithms, which are 'Recommendation,' 'Analyze,' and a string search being implemented. In order to implement the first algorithm, this search must pass correctly. The string search does not assume that the user will type the book name in correctly. The data, which are the books, will be kept in a database. This search ensures that if the book is in the database, there will be a dropdown menu allowing the user to choose from the options that are in the database. If the book does not exist in the specific database, there will be no search results available. For the tags, the way the search function will work is by using a hash map, where all the tags and book are assigned to a hash map so we can quickly retrieve the book in our algorithm. There are many algorithms and libraries in python that can do this work for us so we can focus most of our time on the recommendation algorithm itself.

This also works well for people who are consistent at misspelling words. If words are misspelled, they will not be able to find the book that is actually available in the database. Therefore, having a dropdown menu that starts giving the user recommendations as they start typing the book will make it more convenient for the user to find the book rather than be unhappy when a search is unavailable due to a spelling error. Essentially, there will also be an assumption that the user is searching for books that are already in the database, for the books that are selected from the dropdown menu. For example. if someone were to try to find a book in a different language, this search would return no result because that specific book would not be able to be located in the database.

The program will allow multiple users. To allow this, the program will create a new spreadsheet for a new user. That spreadsheet will be used to store the information about the books entered as wells as the books that we pulled from the database that has the other books. This will help with sorting later down the line. As the user is inputting the names of the book, the system will being to predict what the use is trying to input. This will save the user time and thus increase satisfaction. Not only this but because the data is case sensitive, this will minimize errors and discrepancies.

For the second part, there will be a 'Recommendation' algorithm. This algorithm will work with the tags we put on the books. Each tag will have a set multiplier (1x, 5x, 10x, etc) added to the books. For example, the 'age' tag will have a x50 or so multiplier to ensure that the recommendations that are coming back are in the same age limit of the user/reader. 'The Lord of the Rings' would definitely not be a good recommendation for a 5- year old with a very limited vocabulary. Another example is a tag for books written during certain time periods. A modern millennial may find a book with a few similar tags, but when he or she sees the tag that determines what time period it was written in, he or she may not want to read a book that was written such a long time ago because he or she may not be familiar with the vernacular of the time. This is also a useful tool for parents who do not want their children to read books with a much higher maturity level than they can comprehend. A big concern for parents these days are whether or not a book is fit for a child, especially if they have no knowledge about what the book is about. A kid may want a book on monsters, but would probably not be to happy reading a book that would give them nightmares because of a book that was very explicit and too

gruesome for even the most avid young readers. This recommendation algorithm will recommend books to users without typing in specific keywords, which is essentially the whole point of the algorithm.

Once the search is done, the results will be displayed in tiers. These tiers will be based on a statistical analysis of each book's tags. The more similar the tag, the more closely related the books will be. This would be considered as the second algorithm being implemented in the program. This 'Analyzing' algorithm uses the points allocated to different books based on our Search Function, using the data for statistical analysis of the results. Using this analysis, we can group our results into tiers (S, A, B, C from highest to lowest) based on standard deviations. This allows the user to see not only which books are recommended to them based on their inputs, but also how how closely related each result is, giving a more organized and focused list of results. The S tier books will contain books in the top 0.5% , A tier books will contain books in the next 2%, and so on and so forth. Using a standard deviation to split the tiers, we will decide a baseline score required for each book to earn a spot on the list of recommended books. This is the most optimal way for organizing the books for the user because it is easy to implement on a user interface and will be convenient for the user to see the tiers and maybe just have to decide between 5 or 10 books instead of 50+ books.

The most important feature in this program would be the recommendation algorithm. This algorithm needs to identify the tags on each book in the database in order to return the tiers. It will group the books that have the most tags in common, and list them after the user selects the book from the dropdown menu. The tags can include things such as genre, age level, "Disney", number of pages, difficulty of vocabulary, etc. The database is required to have these books with these tags in order for the search to go through. The database that will be used will be an open source library with about at least 50,000 books that will be searched through for their tags.

For the user interface, there will be an easily accessible website to use. So every input, whether it be from a input device or a smartphone screen, will be handled through features of the user interface and will essentially allow the program to perform its job through user demands. The list of books that have already been read will be displayed horizontally. Below this list of books that are already read, there will be a section called "Top Picks for You." This section will be based off all the recommended books that the user has displayed interest in previously. There will be a trending section which lists all the new books that are based off popularity and reviews. When scrolling through each book, there should be a brief summary of the books, the ratings, and the genres. Any additional lists will have other genres for easy access to a book that is **not** part of the user's preferred genre. For example, these lists may include: comedy, horror, romance, drama, etc. However, only a few genres should have their own list at this point because if not, users would have to keep scrolling to find their preferences which may end up ruining their experience on the platform. As previously mentioned in the search string implementation, the drop down menu can also be used to have users explore further. In any list, the image of each book will be its cover so that the user can easily identify the book without any difficulty. every input, whether it be from a input device or a smartphone screen, will be handled through our features of the user interface and will essentially allow the program to perform its job through user demands. There will need to be a way to check the algorithm consistently after it is

implemented. The database will constantly need to be refreshed as new books are added to the system. As the database is updated, the tier list will change for every input that is added in. If more books are added, more tags will also be added. This algorithm should be able to take this into account when the search is being done by the user.

The on-screen appearance aspect of this project is where most of the front-end design and development will be hosted. Along with the front-end work, this is also where the user interface aspects of this project will reside. The front-end design and development will drive how the visuals and website aesthetics will not only look, but also operate accordingly to the functional demands of the backend partition. All of the visuals of the website and user input will be held and driven through the on-screen requirements. The on-screen appearance aspect of the project will also be where the user-interface will strive. So every input, whether it be from a input device or a smartphone screen, will be handled through the features of the user interface and will essentially allow the program to perform its job through user demands.

There are a few main goals that need to be accomplished by this project. First, there needs to be a well made interactive website for the user to connect with the database. This database will need to be managed by a developer who updates the database every two weeks or so to make sure that the latest trending novels are included in the database. The website should be user friendly, and look the part. It should be easy to navigate through and easy to read. Next, there should be random checks to make sure that the tier list that was created is up to date with the database. There should not be books left out when more are added into the system, and each books' tags should be registered into the system as well. As more books are added, the searching string needs to make sure that it takes into account the added books. Adding more books into the database should not cause the algorithm to run slower. It should be optimized at the same speed that it was running on prior to the adding of new books into the database. The biggest challenge will be optimizing the runtime for this algorithm to run efficiently and quickly. Since there will be more than 50,000 entries, it might take a while to get a recommendation from the database which requires optimization. Getting the runtime to be efficient will require a lot of testing on a public database which will give good feedback on how efficient the algorithm is. Even after the program is launched, it should be checked on occasionally to make sure that the runtime is not affected badly at any point in time.

2.3 Glossary of Terms

Algorithm: A process or set of rules to be followed in calculations or other problem-solving operations.

Analyze: The algorithm that is used to separate the returned books into a specific tiers.

Books: The entries in the database.

Database: Hosted on the website, used to store the entries (books) and their tags.

Entries: Every single book in the database is an entry itself.

Feedback: Information about reactions to a product, a person's performance of a task which is used as a basis for improvement. For this project, the feedback would come from how well it works for users.

Tag: Words and short phrases that are used to describe a book.

Tier: Subsections which identify how close in relationship two books are. It includes a list of books that are related to the searched book.

Search: The algorithm that is used to find the user input from the database.

Recommendation: The algorithm that is used to recommend a book based on the tags that is associated with it.

User Interface: How the user and the computer system interact with each other. In this case, it will be a website that the user uses to search for their recommendations.

String Search:

Platform: The basic hardware (computer) and software (operating system) on which the applications are run.

Optimization: Bettering the algorithm so its runtime is faster and more efficient.

Multiplier: Associated with the tags. Each tag has a multiplier. They vary based on the importance of the tag.

Vernacular: The language or dialect that is spoken by people in a specific region or country

3. System Requirements

3.1 Enumerated Functional Requirements

Identifier	P.W.	Definition
REQ-1	**2**	Use hash map for optimized time
REQ-2	**5**	The tags will be organized in a link list type of fashion that allows for easier more organized traversing.
REQ-3	**4**	Account for all distinct tags that associate with all the books entered
REQ-4	**4**	Create a new table every time a new user enters the list of books (New user just means a new list. Users are not recorded)
REQ-5	**5**	Spreadsheet will hold a list of books that have tags that are matching the distinct tags list
REQ-6	**5**	The list will include all books that have at least Age and Genre matching.
REQ-7	**1**	(Optional) Delete the last sheet to save memory allocation
REQ-8	**4**	System will begin predicting user input
REQ-9	**2**	Update prediction with ever letter input
REQ-10	**4**	The spreadsheet list will be organized via point system to rank them
REQ-11	**4**	Ranking will be broken via percentage tile of the highest point in that list 95%+ =S, 90%-94%=A, 80%-89% =B, 70%-80%=C
REQ-12	**2**	System will have a short description about the tier in the box located next to the letter
REQ-13	**4**	List should hold tags about the books

3.2 Enumerated NonFunctional Requirements

Identifier	P.W.	Description
REQ-1	**3**	Database should be updated once week to account for new books and other updates
REQ-2	**5**	System must be able to handle infinite amount of inputs and display finite amount of outputs for accurate representations. This should not take longer than 5 minutes
REQ-3	**4**	System must display suggestions for spelling mistakes or if the inputs are not written exactly as the actual book name (even if few words are written, or should read 50 as the word “Fifty” in the book and vice-versa)
REQ-4	**5**	System should display an error message for invalid inputs, even if there are few valid list of inputs followed by an invalid on (“Try again” button that refreshes the system.)
REQ-5	**5**	System should prompt an option to start anew after displaying output
REQ-6	**4**	System must be able output the recommended book using the algorithm in a quick fashion (Again limit is 5 min)
REQ-8	**3**	Program should be able to handle multiple clients accessing the program
REQ-9	**5**	Application must be straightforward to use
REQ-10	**5**	System code must be broken down into manageable and maintainable parts
REQ-11	**5**	Must be able to run on web browsers without error
REQ-12	**5**	Output should be displayed only after all desired inputs are submitted

3.3 On-Screen Appearance Requirements

Identifier	P.W.	Definition
REQ-1	**1**	A list of books already read should be displayed horizontally.
REQ-2	**5**	Below the list of books already read, a “Top Picks For You” section should be dedicated to all the recommended books based on the user’s interests
REQ-3	**2**	A trending section must list all books based on popularity and reviews
REQ-4	**5**	When scrolling through each book, there needs to be a brief summary of the book, the ratings, and it’s genre.
REQ-5	**4**	Additional lists should have other genres for easy access to a book that’s not part of the user’s preferred genre. For example, such lists are comedy, horror, romance, drama etc. However, only a few genres should have their own list at this point. Otherwise, the user would have to scroll further down to see all genres which can ruin the user experience
REQ-6	**2**	A browse drop down menu should list all possible genres for users to further explore.
REQ-7	**4**	In any list, the image of each book will be its cover so that the user can easily identify it.

4. Fully Dressed Description and Diagrams

4.1 Use Case Analysis

1. Case 1:

1. Step 1 (Entering the List of Books)

a. User→ Begins typing the book
b. System← starts using predictive text, begins to suggest book name
c. User→ select book which they are looking for.

The user then searches or finds books that they want by scrolling through the list and evaluating which books have the most similar features.

2. Case 2: (Find books according to their list of read books)

d. User→ Enter the name of book A into the search bar
e. System← begins to find the name of the book in the database for predictive text
f. User→ Selects the book from the predictive text
g. System← Creates table that stores all the books that the user input
h. System← Starts running the similar match algorithm, and then stores them in the same file with the books that were entered
i. System← Use Analysis Algorithm to rank the books form S→ C

4.2 Traceability Matrix

For Enumerated Functional Requirements

Requirements	Priority Weight (of Requirement)	UC-1	UC-2
REQ-1	2		✓
REQ-2	5	✓	✓
REQ-3	4		✓
REQ-4	4	✓	✓
REQ-5	5	✓	✓
REQ-6	5		✓
REQ-7	1		✓

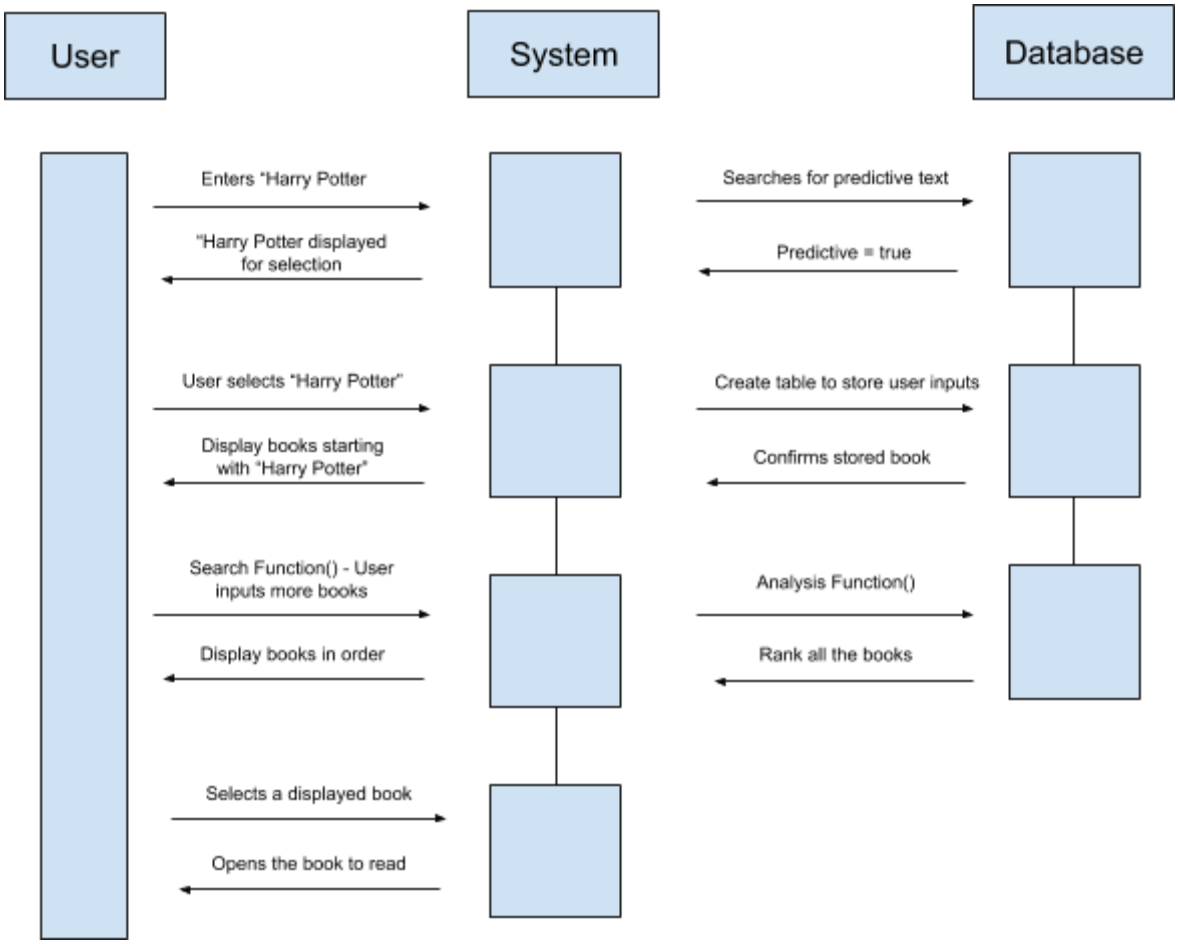
REQ-8	4	✓	
REQ-9	2	✓	
REQ-10	4		✓
REQ-11	4		✓
REQ-12	2	✓	
REQ-13	4	✓	✓

Max Priority Weight (of Use Case)	3	5

Use case 2 is most definitely what makes our software unique compared to similar software available today, hence why it is given the highest weight. The specific detail of use case 2 that so essential to our idea is that it involves ranking the found books into tiers, and then comparing all the tiers to all the books who were entered. It essentially creates the basis for which what our program will conclusively decide what will be recommended to the user. This specific tool is what our software does, that no other current software considers, hence why it is one of the crucial and innovative pieces of our project.

Use case 1 is the same type of implementation that can be found in practically all web searches, making it not too exclusive to our design, hence why it was given a ⅗ priority weight. There is a slight step within this case which leads to the completion of the use case. Since the user must select his or her book from the drop down menu, this ensures that there will be no type of spelling error and makes sure that the book is located within the database. So the entirety of the case includes 1. Typing the name in with at least three letters and 2. Having the aalgorithm run to dropdown the list of books.

4.3 System Sequence Diagrams



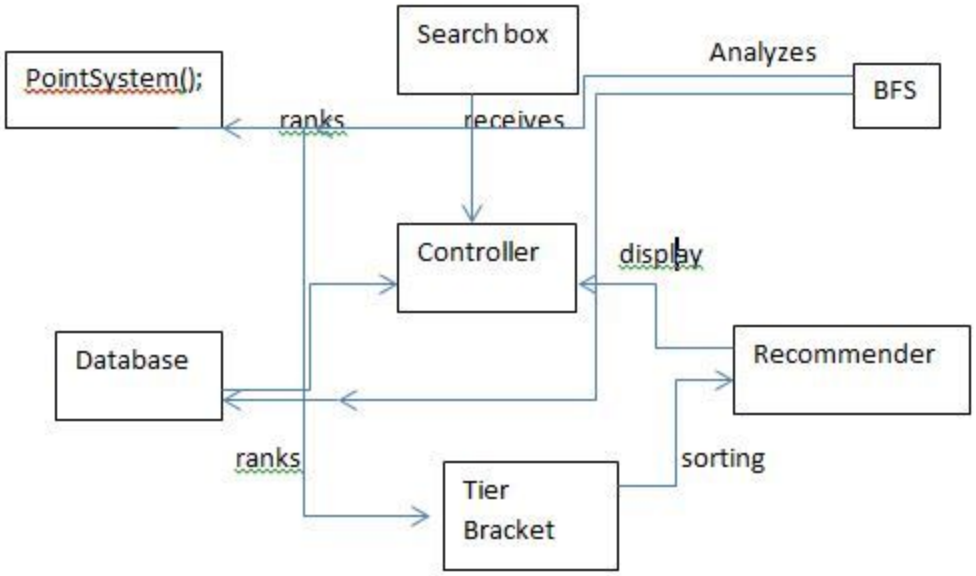
5. Effort Estimation

The scenario with the most usage is likely someone looking for books that explore the same themes and share the same qualities (author, genre, age, etc) as books they have enjoyed reading previously. In order for this scenario to be present, the user would have to enter the name of each book. Because we don't want them to enter the name of a book which we do not have in our database, the application will have a query completion dropdown menu from which they will have to select the book they wish to enter. This will decrease the number of keystrokes required to enter each book's title, and although it increases the number of mouse clicks (1 per book) it will decrease the number of inputs required overall.

After inputting all the books, one mouse click is required to execute the recommendation algorithm on the selected books. Therefore, between 5-10 keystrokes (estimate) and a mouse click may be required per book, and another mouse click to generate results. With 5 books being used to search for results, a total of 25-50 keystrokes and 6 mouse clicks are required. In the results page, after the search is completed, 1 mouse click is required to navigate to each book for information that the user wishes to view (not on our website). Of all of these inputs, none are required for user-interface navigation as the website has a search page and results page; rather all of the inputs are clerical data entry that is required for the algorithm to function.

6. Domain Analysis

Domain Model



Domain Model Derivation

In simplest terms, the domain model ‘s purpose is to take all the characteristics and operations of the requirements and use cases, then proceed to describe and illustrate how they interact with each other to make the program operate. The domain model allows for an in depth look at these interactions and gives a clearer look as to what is happening in the “backend” portions of the program.

For our particular program, the main components of the program exist in the user interface, database, the manipulation of the data in the database, and then finally the end result outputted to the user. In essence, besides the input partition of the program, every piece of this project interacts with each other and depends on each output from the previous step.

The user interface allows the user to input the books they’d like to search, and our particular search box will then interact with our current database of existing books in order to try and predict to the user the entire title of the book.

The database in itself will be the host of where all the inputted books will reside (provided to the database from the user interface), where all the current books in the database resides, and where the inputted books along with the books for recommendations. So the database interacts with the user interface (search box), and also essentially plays a role with the data that will conclusively be communicated to the user.

The manipulation of the data in the database will use the information gathered in the database, and begin to breakdown the significance of certain books in order to build possible recommendations to the user. So it will interact with the database's given outputs, and proceed to filter out what will/will not be communicated to the user.

The output will interact with the most updated version of the database (what's left after all the manipulation), and output it chronologically to the user. So this will have heavy interactions will practically all aspects of the program except for the initial search box.

6.1 Traceability Matrix

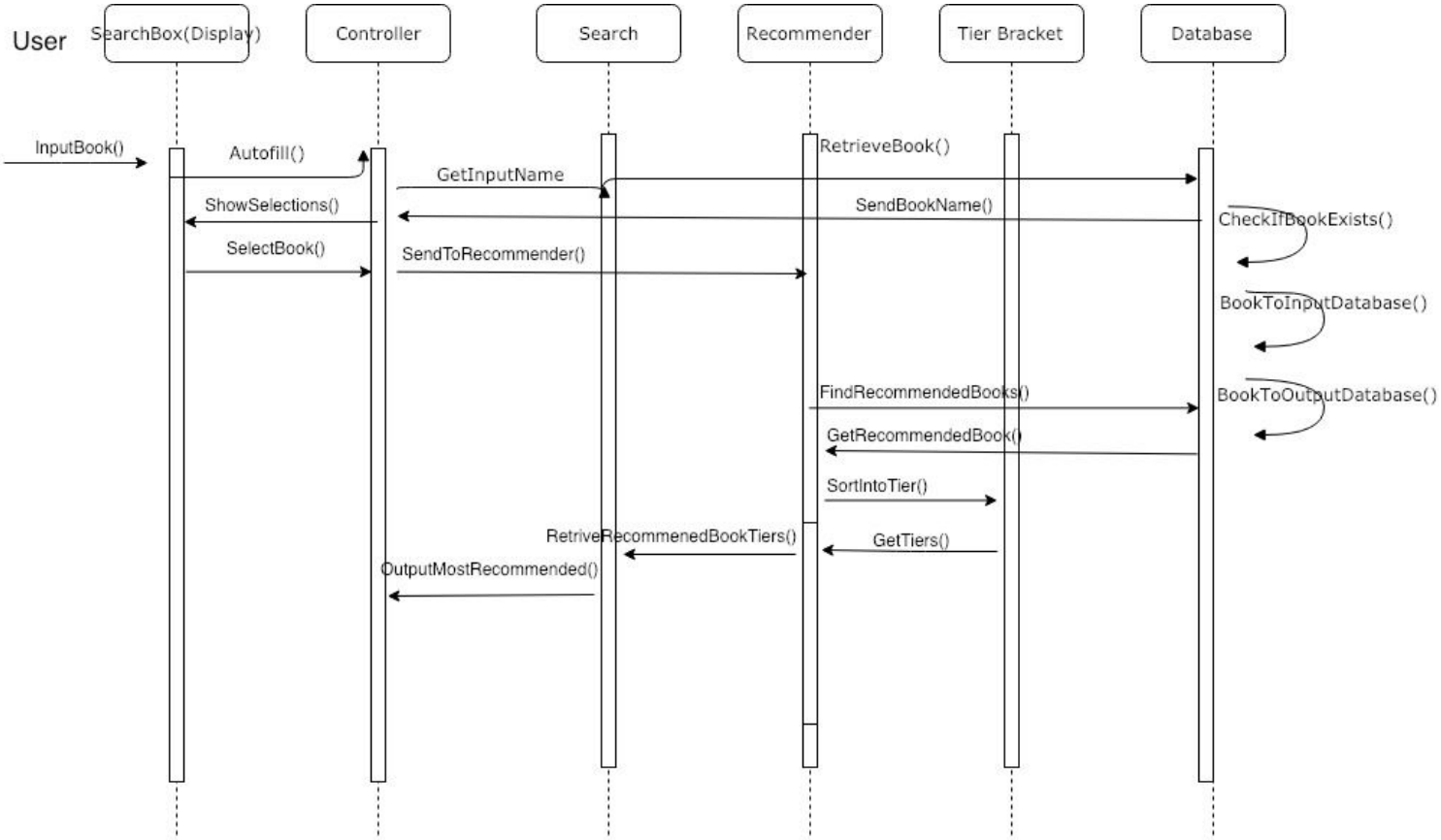
The domain concepts here are elaborated on and described further in the sections of "chapter" 8 of our report, so i left the description out of this table although the instructions said not to

Domain Concept	Use Case 1	Use Case 2
book	✓	✓
book_tags	✓	✓
books_with_tags	✓	✓
cosine_sim		✓
img		✓
indices		✓
ratings		✓
tags	✓	✓
tags_join_DF	✓	✓
titles	✓	✓
recommendation(book1,book2,book3)		✓
read_csv([Book])		✓
listS		✓
book_indicesS		✓

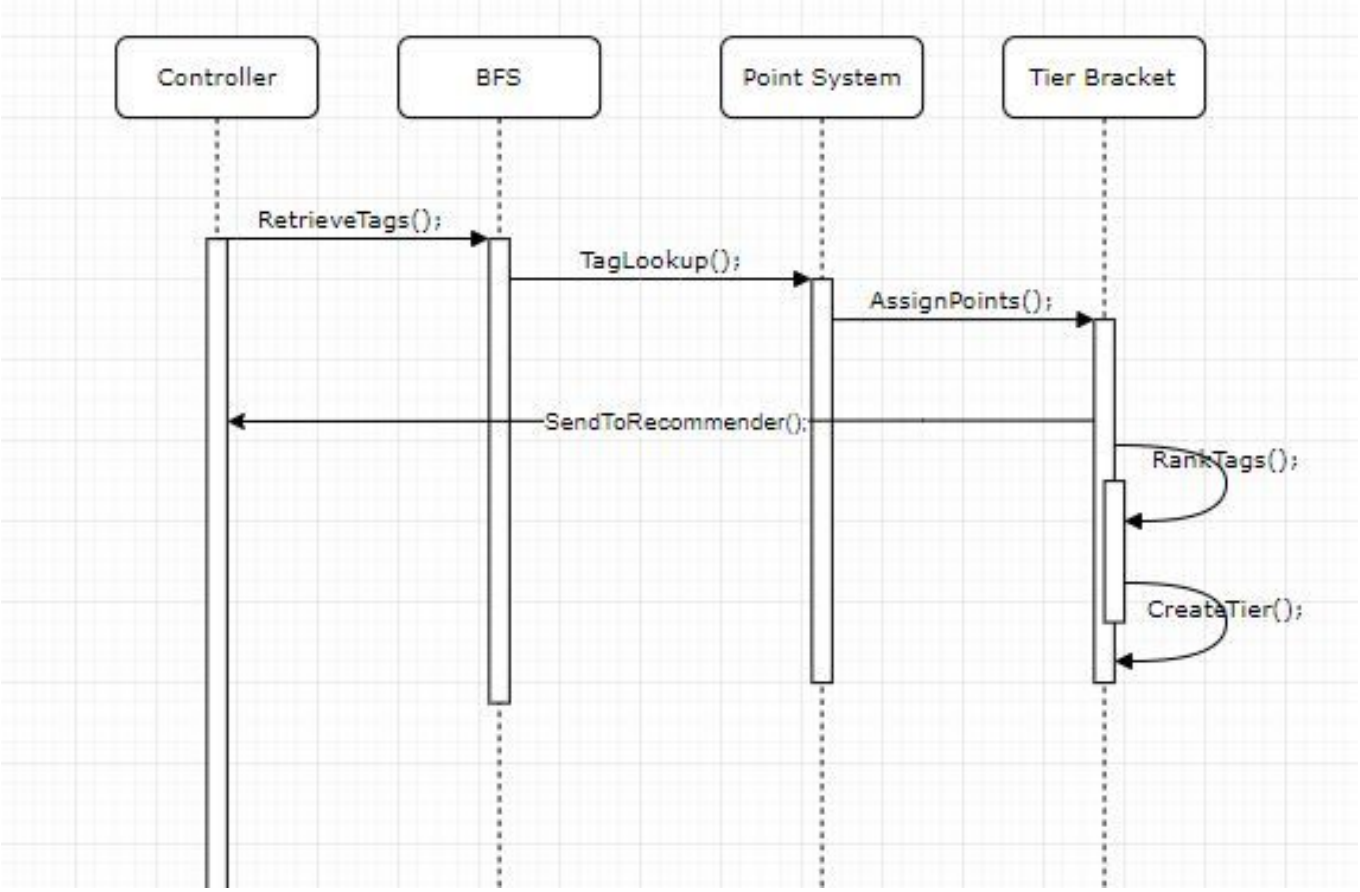
dataString(titles)	✓	
get()	✓	
post()	✓	

7. Interaction Diagrams

Use Case 3



Use Case 4



Interaction diagrams are most effective when they're manufactured off the basis of a design system sequence diagrams. What a design system sequence diagram does is it looks at all of the objects that help drive a system, and breaks down how exactly each object interacts with other objects. These interactions conclusively defines how each object contributes to execute what the system is intended to do. They are very in depth and much more detailed visual representations of how each object plays a certain role in getting a task completed.

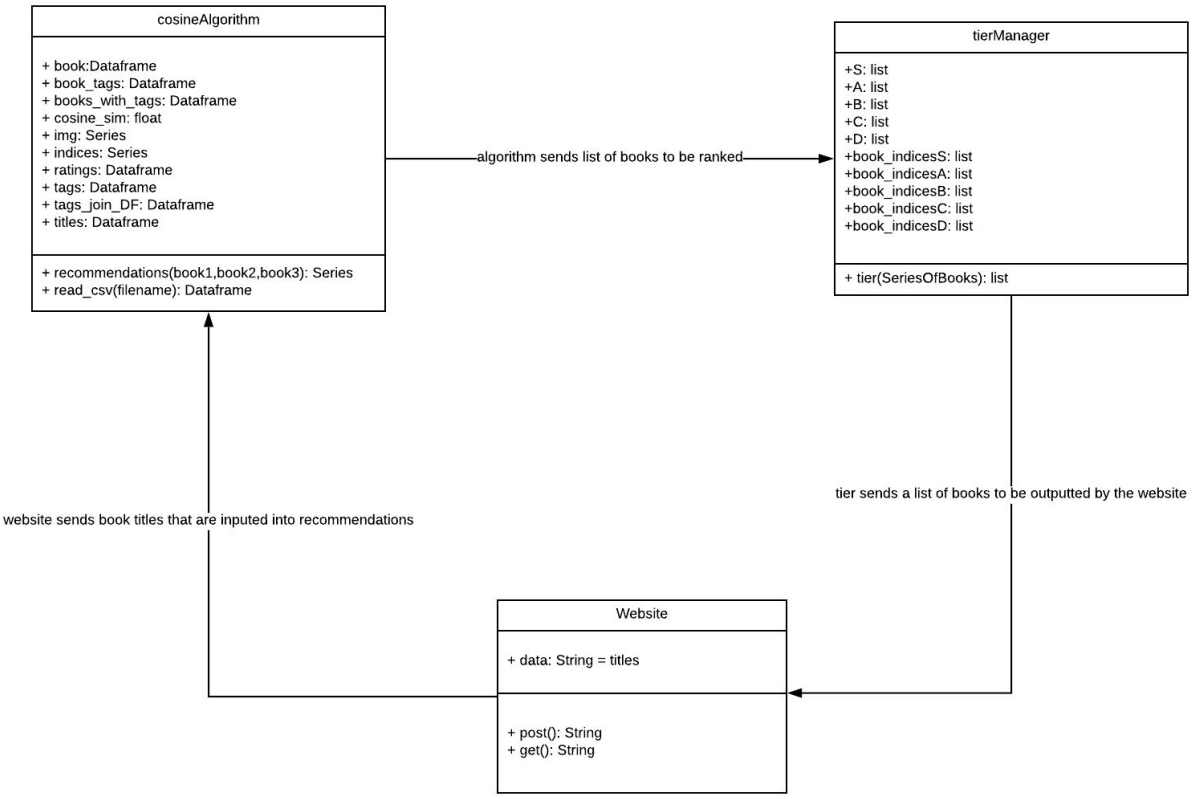
Objects are the different functions inside of a class (programming definition of class), which make the class fulfill its job. So we had to look at how each object interacts with one another, and break down a task into pieces and make sure each piece is given attention from objects.

When assigning our objects to responsibilities, we used certain design principles in order to make sure that each object had a fair share of responsibilities and that the work between the system was fairly spread apart so no single object had too many tasks. We also used some of the techniques provided through the SOLID design principle to make sure that we were forming a diagram that does necessary tasks, and nothing was redundant or irrelevant.

For use case 3, which is the most dense use case in regards to tasks that need to be done, the main focus was making many distinct objects who allows for use case 3's tasks to be broken down into as many singular tasks as possible. The main design principle that allowed for us to assign responsibilities to all the objects was the high cohesion principle, since our use case 3 involved the most computational and data manipulation out of all the use cases. The single responsibility principle from SOLID also helped us in breaking down the task into classes, followed by the high cohesion principle assisting in helping break down responsibilities of the concepts in each class. The goal was to make sure not a single class had objects responsible for too much at once, by creating several objects interacting with each other in order to make up that class.

Use case 4 had a lot more to do with the user interface, and communicating the correct results to the user. Given the task of use case 4, it was a dead giveaway to us to look into the low coupling principle when assigning responsibilities to objects because the low coupling principle mostly deals with communications in regards to one object being the primary one to delegate responsibilities to other objects. The goal here was to make sure that any object isnt responsible of communicating too many things independently, and that we made sure different things that needed to be communicated were each handled by single objects. Liskov's Substitution Principle also assisted with this use case because it allowed us to make sure that each object should operate independently, but should still interact with other objects. No changes in our software, system, or objects around a certain object should directly affect any object, only changes directly made to an object should affect its performance and outcome.

8. Class Diagrams and Interface Specification



8.1 Data Types and Operation Signatures

1. cosineAlgorithm
    a. Attributes
        - Dataframe book: holds all the basic information about books such as id, author, ratings, image links, title
        - Dataframe book_tags: holds the id the book with corresponding tags
        - Dataframe books_with_tags: only holds books that has tags
        - Float cosine_sim: holds a matrix of cosine similarity scores
        - Series img: contains all the image links for each book

- Series indices: contains all the indices for each book
- Dataframe ratings: contains ratings for each books
- Dataframe tags: holds the correspond tag string for each tag id
- Dataframe tags_join_DF: combines the titles and tags
- Dataframe titles: holds the ids and titles

b. Operations

- recommendation(book1,book2,book3): uses our modified cosine similarity algorithm to take in the cosine scores of each of the books and output a single list containing the best books to read in order from best to worst
- read_csv(): used to help convert all the csv files to dataframes

2. tierManager
   a. Attributes
      - List S: contains all the books places in tier S. same logic applies for the other Lists
      - List book_indicesS: contains all the indices for tier S used to look up the books

8.2 Design Patterns

For our project, we used the Publisher-Subscriber design pattern. The subscriber, the website, sends the book titles to the cosine algorithm in order to get similar recommendations to these books. The algorithm does inventory of the books requested by the website and accesses the database to provide all the detailed information such as authors, ratings, and tags to the tier manager. The tier manager is the publisher in that it categorizes the requested books based on the similarities between the books requested and the books available with the help of the cosine algorithm.

8.3 Object Constraint Language

There are some preconditions and postconditions for our classes. The website has the preconditions of having three books typed in the search bar and that the user clicks on the search button. The postcondition is that the website sends list of books for which the user needs similar books related to them. The cosine algorithm needs to receive three books from the website and needs to be able to access a data frame comprised of books, their corresponding tags, and cosine similarity scores. The tier manager has the precondition of receiving the similarity scores produced based on the cosine algorithm's assessment of the similarity between the books requested and the books available in the database. The tier manager has preconditions of having all the recommended books in tiers/categories and contains all the indices for those tiers used to look up the books. This class has a postcondition of sending all the similar books in their appropriate tiers to the website so that it can be outputted.

8.4 Traceability Matrix

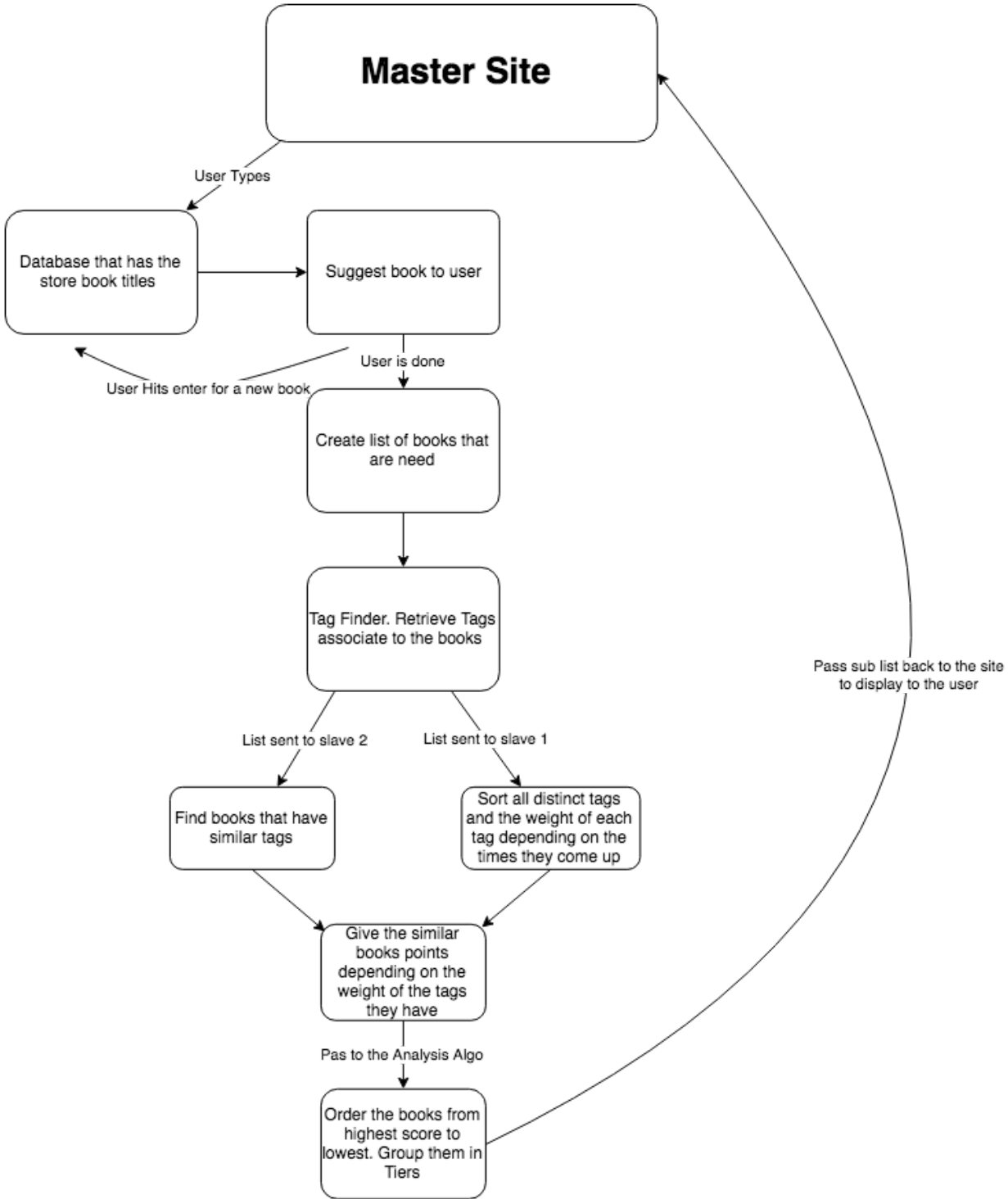
Class	Domain Concept	Definition/Operation
cosineAlgorithm	book	holds all the basic information about books such as id, author, ratings, image links, title
cosineAlgorithm	book_tags	holds the id the book with corresponding tags
cosineAlgorithm	books_with_tags	only holds books that has tags
cosineAlgorithm	cosine_sim	holds a matrix of cosine similarity scores
cosineAlgorithm	img	contains all the image links for each book
cosineAlgorithm	indices	contains all the indices for each book
cosineAlgorithm	ratings	contains ratings for each books
cosineAlgorithm	tags	holds the correspond tag string for each tag id
cosineAlgorithm	tags_join_DF	combines the titles and tags
cosineAlgorithm	titles	holds the ids and titles
cosineAlgorithm	recommendations(book1,book2,book3)	uses our modified cosine similarity algorithm to take in the cosine scores of each of the books and output a single list containing the best books to read in order from best to worst
cosineAlgorithm	read_csv([Book])	used to help convert all the csv files to dataframes
tierManager	listS	contains all the books placed in tier S. same logic applies for lists S, A, B, and C
tierManager	book_indicesS	contains all the indices for tier S used to look up the books
Website	dataString(titles)	Uses predictive text algorithm in order to predict, and input book titles into the search bar

Website	get()	Sends books to the cosineAlgorithm class in order to create the books that will be recommened
Website	post()	Recieves books from the tierManager class, then provides the output (in the form of tiers and images) to the user

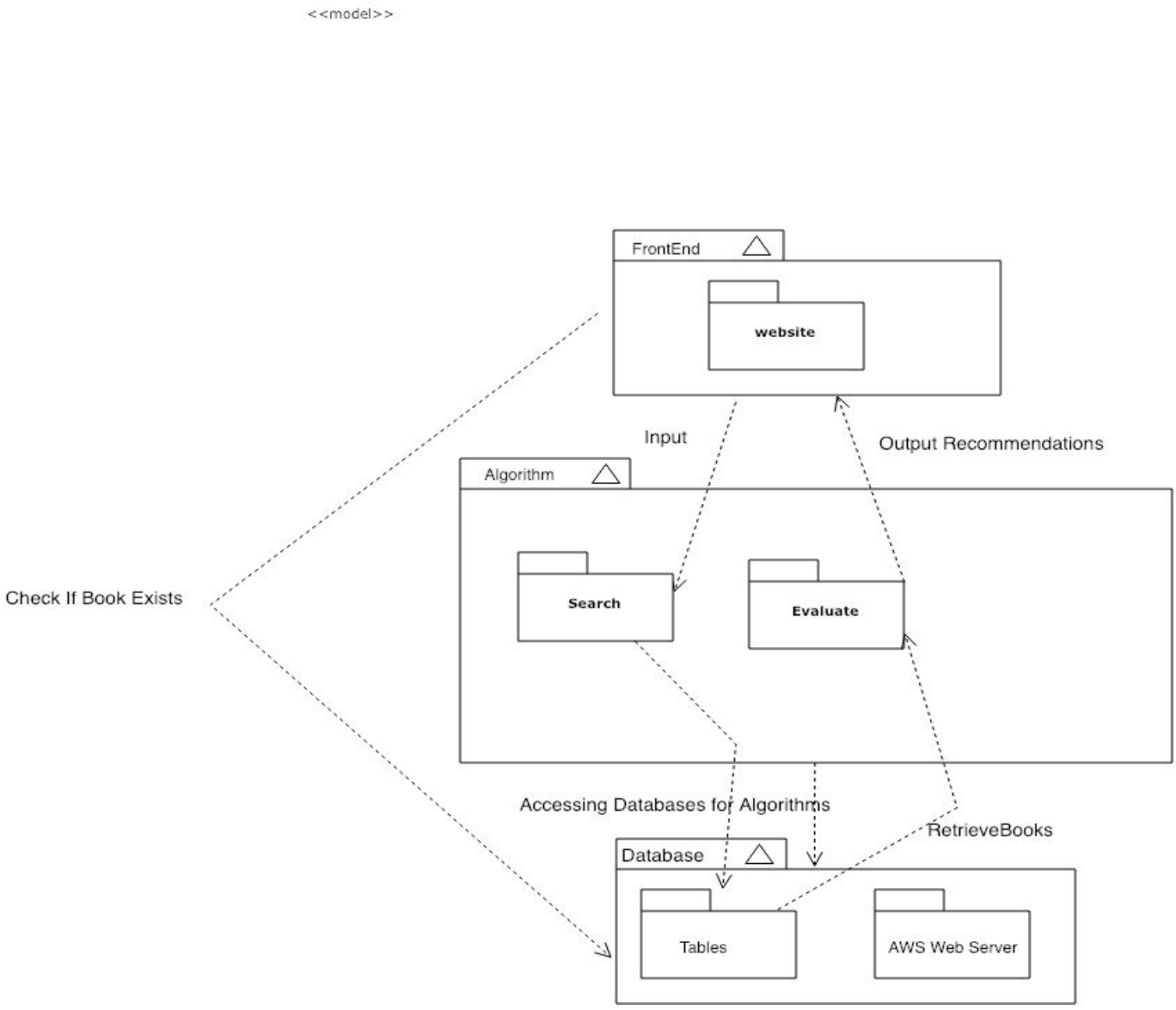
9. System Architecture and System Design

Architecture Styles:

We basically used two types of architecture styles. The main style that we used would have to be the Master-slave style. The master in our case is the site itself that takes the user input and thus the main type of data that we need. After that, the data is shared among the slaves (Tag retriever, Score giver, Sorter, Grouper). When data gets sent to the slaves, there are layers and order in which data goes through to reach the final result that we want. That is the layer portion. We also have a version of Client-Server pattern which is used for out input front end system. As the user puts in the book that they read, the data will be sent to the database and begin searching that book. Then that data will come back and present itself in the form of a dropdown menu.


Master Site
User Types
Database that has the store book titles
Suggest book to user
User is done
User Hits enter for a new book
Create list of books that are need
Tag Finder. Retrieve Tags associate to the books
List sent to slave 2
List sent to slave 1
Find books that have similar tags
Sort all distinct tags and the weight of each tag depending on the times they come up
Give the similar books points depending on the weight of the tags they have
Pas to the Analysis Algo
Order the books from highest score to lowest. Group them in Tiers
Pass sub list back to the site to display to the user

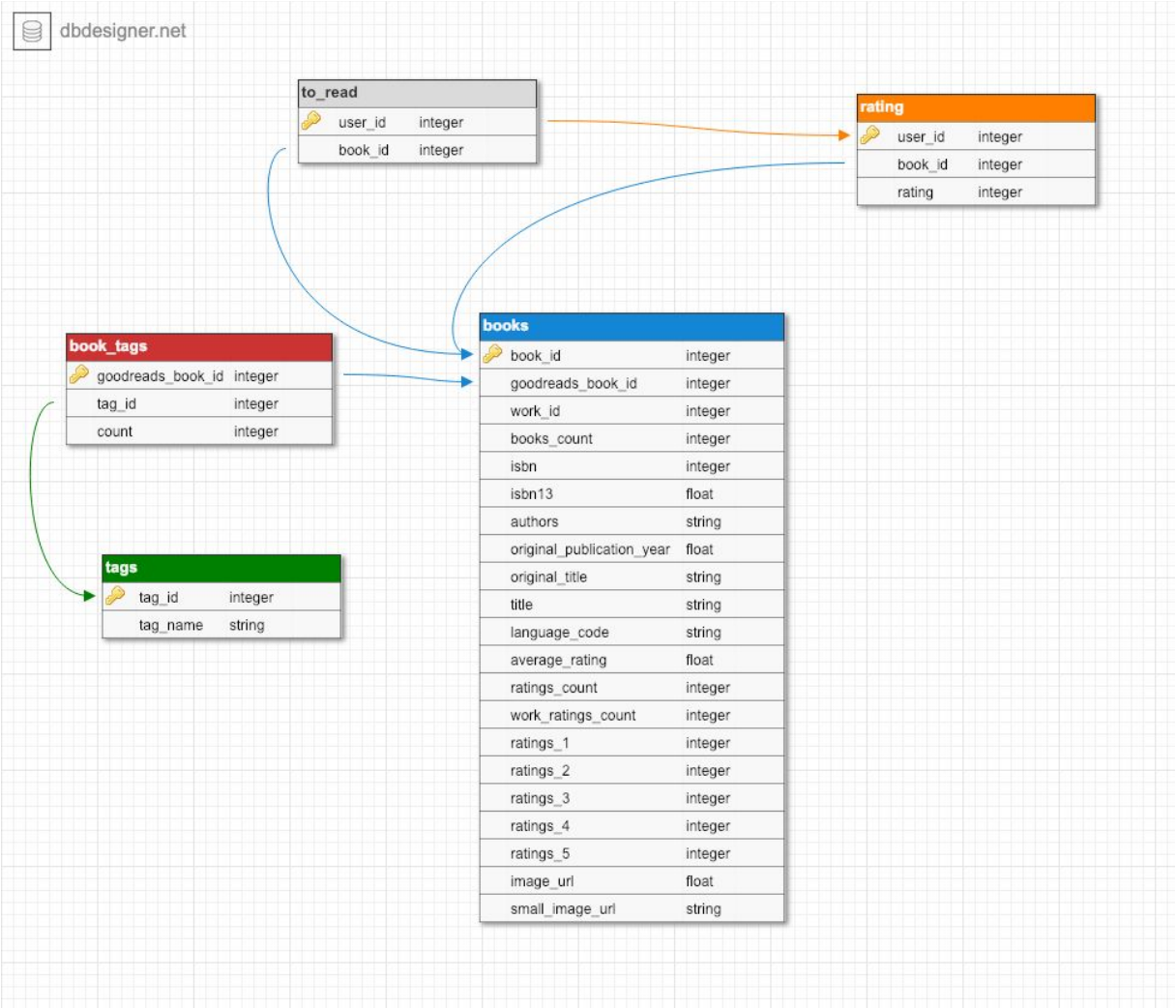
Identifying Subsystems



Mapping Systems to Hardware

Our system is expected to run on multiple computers since our book recommendation is deployed on a web-browser through AWS (Web Server). For frontend, we are implementing using Bootstrap to make a user-friendly and dynamic site. From the frontend site itself, the system is accessing the database to check for book's existence and if it exists then the system accesses the backend part to run the recommendation algorithms. The Search subsystem accesses the database and finds out all the similar books to recommend. At database level, the system communicates to Evaluate sub-system to display the recommended users on the frontend to the user.

Persistent Data Storage



Global Control Flow

Execution Orderliness:

Our system is event based since it waits for user's book title inputs before searching the appropriate books for the user. Different kinds of book genres, titles, author relevances, and other various attributes will generate different types of recommendations. Therefore, the whole functioning of the system is driven by what the user inputs.

Time Dependency:

Our system has no time dependency since we are updating are not updating the book database.

Concurrency:

Our system does not use multiple threads, but if we have time to finish the main program then, we would implement multi-threading to optimize time for faster screen time execution.

10. Algorithms and Data Structures

Algorithms

$$\text{similarity} = \cos(\theta) = \frac{\mathbf{A} \cdot \mathbf{B}}{\|\mathbf{A}\|\|\mathbf{B}\|} = \frac{\sum_{i=1}^{n} A_i B_i}{\sqrt{\sum_{i=1}^{n} A_i^2}\sqrt{\sum_{i=1}^{n} B_i^2}},$$

For the recommending algorithm, we used a common similarity algorithm called the cosine similarity algorithm. It essentially maps all the genre and author tags contained in our database on a vector coordinate system. To see how two or more books may be similar to each other it measures the angle between the two vectors and then uses the equation above to measure how close they are two each other. If the books are highly similar the value will be close to 1, but the farther away the book or vector is the score will get closer to 0. To find the set of books that are similar to a set of books inputs, we find the matrix of similarity scores for each input book and then add up the all the scores. We then only have to sort it and give back the top results that are the most similar. This seems like a lot of steps, but using the pandas library in python a lot of the brute math is done for us using the built in methods.

Data Structures

For our data structure we are using Lists from python. The reason that we are using lists is because lists provide us with the ability to add elements as we go. This feature is very useful due to the flexibility it allows since part of our algorithm searches the books entered, and collects all the tags as they go through the books. The flexibility plays a part in the role of the structure because the user can input an infinite amount of books. Secondly, with the use of list we are able to combine multiple lists together in order to create a resultant pseudo hashtable. For example, if we want to see how many times a certain tag was found; we could have one list for the tags generated from the data of each book, then a separate list for the count of the occurence of each tag.After the data from the user ceases to grow any further, we can simply combine the two lists into one list that has (Tag,Count) as their elements. This makes it much easier for us to visualize the data and go through it. The main idea of the approach was to emphasize on flexibility due to the large amounts of data we have, and how vulnerable these data is to change.

11. User Interface Design and Implementation

From the mock-up, it was originally planned to display the books with their images on the homepage of the site. However, on the home page the only features that will appear on there are just the search bar and a horizontal menu which consists of a genre button and a favorites button. The genre button gives a dropdown of all possible genres for books and the favorites gives a dropdown of the user's favorite books. The horizontal menu will be positioned at the top and the search bar is centered in the middle of the home page.

12. Design of Tests

Test Case Description

a. The test cases for our program are fairly simple because we just have to input in various books ranging from genres that are related like Harry Potter and Lord of the Rings, to books that are different like Romeo and Juliet and the Hunger Games. We will also input incorrect spelling of titles, null values, and books that do not exist. (enter some example test cases if you can)
b. The main idea of the list data structure is in order break down the describing characteristics of our user input, and put all the information of the inputted tags all in one place (a different list). So the testing for this data structure was fairly simple as we only needed to make sure that the number of times certain tags came up in our inputted list was recorded. So being as that we already have developed a work-in-progress database, we used some of the books from that database in order to test different quantities of

inputs from a user, and made sure that the list accurately batched up all of the data from those inputs accurately.

Test Coverage

a. The test coverage of our tests is mainly for the front end because by the time the inputs arrive to the backend to be used in the algorithm they should already be preprocessed to make sure the spelling is correct, the books exist, and the correct numbers of inputs have typed. The front end is in charge of telling the user to keep on entering the correct inputs until the desired format of inputs is correct. At that point we can assume, the program will return the recommended books without any problem. Like any recommendation programs, depending on the inputs sometimes you may get back books that are actually similar to the books you inputted, but if you input a bunch of drastically different books the program will have a hard time returning back similar books for you to read.
b. Testing for the backend will require more effort. Using a sample set of books and estimating return values, we can adjust our algorithm multipliers (for different tag types) in order to return books that are most like the books we've inputted. With repetition we should be able to enter any list of books and receive a list of books which has strong correlation to those entered.

Integration Testing Strategy

a. Our integration testing is going to be focused on multiple book input and ensuring that having more than one changing variable will not drastically affect the outcome of our product.
b. We are going to show that our system can take infinite book inputs and generate the recommended output almost instantly. For this, the length of book inputs is until the last input added in the search bar (of our front end site) until the Search Icon is clicked.
c. We are going to show that the system is able to detect spelling errors and predict an appropriate book name based on the list we have in our database. For this, we will intentionally make some spelling mistakes for some book of our choice and show that the suggestion matches the input we are looking for.

13. History of Work, Current Status, and Future Work

Starting from backend programming, it was a challenge for us to figure what source library to refer to that could possibly have a vast variety of books in its database. Eventually, after referring to some book service sites, we came across Goodreads and referred to its book libraries and added them to our database. For frontend, we were able to outline a design for how it will look to user that visits our site. At every search, the system is suppose to take in all the book inputs and store the output of recommended books in the database and clear it during the next search. However, a challenge that exists whether there will be erroneous data produced if the site is accessed from multiple devices, generating wrong outputs for users. Therefore, we were researching ways to restrict the system functioning to unique devices so that the one person's inputs do not generate outputs on other people's devices.

For the frontend, the bootstrap has been initialized in that the html file contains a search bar for the user's input and a search button that can be clicked on. When clicking on it, the animation shows that it can be clicked on. None of the use cases have been implemented yet because the algorithms are still being worked on.

Our team first divided into 3 subgroups in order to focus on each subdivision of the project. We divided the project into frontend, backend, and bug testing/ documentation. As Akshat, Anthony and Joel worked on the frontend part of the project, including the UI development and appearance of the website, Avani recorded all the information and helped communicate it to all the backend people working on the algorithm. The backend developers including Vedanta, Shazidul, and Alan all worked to get the classes developed as well as set up the database for the project, along with the tier development system. Avani and Kutay sat with both groups and debugged the issues that had come up while the algorithms were in the early stages. They went through each of the 10,000 lines in the excel database and checked and fixed errors where many characters either were not readable words or were just complete gibberish. Kutay relayed information from the ~ backend group to the frontend group about progress updates and what the future plans were going to be before the integration aspects would occur.

However, as the project continued there were a lot of changes made towards our sub-project teams. Akshat and Anthony went ahead to develop all the frontend, which communicated data back and forth with the backend process. For the backend, Alan and Joel worked on coming up with the recommendation algorithm and set up the server. Shazidul developed the tier system for the book outputs. This was the main programming team for the whole project. As the project progressed, the programmers in subgroups ended up doing the bug testing on their own by communicating with other programmers to see if the backend and frontend sections communicate properly when the program is run. For example, while launching

the program with Flask server, backend team communicated with the frontend team on if the application communicated the server data with the HTML site. Avani and Kutaya focused on typing and formatting the documentation for the report, mainly referring to the data provided by the programmers. Vedanta, initially, provided insights and ideas on how to move along with the project till the first demo but after that, there was no contribution by him.

If we had more time, in the future, we would make the following changes -

- Backend - Have the system accept entries that included foreign language inputs and launch the program on AWS instead of a remote server.
- Frontend - Have an option on the site for the user to look through the books and select books from that as input. Make book tile outputs clickable such that, clicking on a book links to the respective Amazon page (based on it's ISBN). Make a category option for what kind of books to recommend.
- Database - Import more categories of books for textbooks, theory books, in addition to the casual Goodreads books. Update the database with new books in real time.