0.Table of Contents

Table of Contents

1. Introduction

The world of literature has been an essential part of human civilization for many centuries, being a way of communication, entertainment, and education. As the digital age has progressed, the traditional methods of book sharing and discovery have undergone significant transformations, leading to the development of digital platforms that adapt to the needs of contemporary readers. The goal of this senior thesis is to present a web application that takes inspiration from the traditional concept of book sharing and combines it with modern technology to create a unique platform for book sharers. The platform, called Next Page, uses various recommendation system algorithms to provide personalized book recommendations to users based on their location and reading behavior as well as a search algorithm to make it fairly easy for users to find books they are interested in.

Background

The concept of book sharing is not a new phenomenon. People have been sharing books for centuries, and book clubs have existed for many years. However, the idea of sharing books digitally is relatively new. Book Crossing is an existing platform that allows users to share their books with others globally. Founded in 2001 by Ron Hornbaker and Heather Green, Book Crossing has grown into a global community of over 1.3 million members in over 130 countries. The platform allows users to register books they want to share, track where they were published, and track journeys passed from one reader to another. However, despite its popularity, the platform lacks modernity and decent recommendation systems, which are the main potential advantages of Next Page.

NextPage thrives to fill this gap by providing personalized book recommendations to users. The web application is created using C# .Net.

Objectives

The main objective of this senior thesis is to design and implement a web application that incorporates recommendation systems to provide personalized book recommendations to users. The specific objectives of this thesis are as follows:

1. To analyze the effectiveness of different recommendation system algorithms.
2. To implement and evaluate the performance of these algorithms in the Next Page web application.
3. To compare the performance of the different algorithms and identify the preferred one based on feedback.
4. To create a user-friendly interface that allows users to easily search for and find books of interest.

Next Page utilizes collaborative filtering to provide personalized book recommendations to users based on their reading behavior. Collaborative filtering is a widely used recommendation system technique that uses user's feedback to make predictions about their preferences. This approach involves analyzing the likings and behavior of multiple users and identifying patterns in their interactions with the system. Once these patterns are identified, the system can predict the potential matches for a particular user based on the preferences of similar users.

One of the challenges in implementing collaborative filtering is dealing with sparsity in the user-item matrix. In most cases, users only interact with a small subset of the available items, resulting in a matrix where most entries are missing. This sparsity can make it challenging to find users with similar preferences, and may also result in inaccurate recommendations. One approach to address this issue is to use matrix factorization techniques, such as singular value decomposition (SVD), to reduce the dimensionality of the user-item matrix and identify latent factors that can be used to make accurate predictions.

Another challenge in developing the NextPage platform is designing an efficient algorithm for computing recommendations. As the number of users and items in the system grows, the complexity of computing recommendations increases, making it

challenging to provide real-time recommendations. One approach to address this issue is to use parallel computing techniques, however, they will not be covered in this thesis. Nonetheless, it can help to improve the scalability and performance of the system, allowing it to handle large amounts of data and provide real-time recommendations.

In addition to these challenges, designing an effective user interface is also critical to the success of the application. The interface should be intuitive and user-friendly, allowing users to easily search for and find books of interest. The platform should also provide a seamless user experience, with minimal latency and downtime, to ensure that users remain engaged and continue to use the platform. Even though UI is a crucial part of web applications, simplistic approach is chosen for this project due to the importance of other functionalities.

Summary

This thesis aims to design and implement a web application that incorporates recommendation systems to provide users with personalized book recommendations. The application uses collaborative filtering to predict the preferences of users based on the preferences of similar users. Besides that, Next Page implements KMP search algorithm to allow for easier navigation. An intuitive and user-friendly interface is designed to ensure a seamless user experience. This paper seeks to evaluate the effectiveness of different recommendation algorithms and identify the most preferred one for providing personalized book recommendations.

2. Specification of Software Requirements And Analysis

In this section, the process of identifying the functional and non-functional requirements of the software and assessing the feasibility of the proposed solution is discussed. It intends to outline the software requirements and analysis process, including the techniques and methodologies used. Clear identification of requirements allows for an easier and more efficient development.

In order to ensure the application is functioning as expected, it must satisfy functional and non-functional requirements listed below.

Recommendation Engine

Personalized book recommendations based on user's preferences are the key requirement for the application. A recommendation engine that employs machine learning algorithms will be utilized to achieve this functionality. The recommendation engine will analyze user data, including genre and user ratings, to generate book recommendations. By leveraging these data, the recommendation engine will suggest books that align with the user's interests, thereby improving the user experience.

Additionally, the engine will continuously learn and update its recommendations based on user engagement with the application, which will result in a more appropriate recommendation system. The recommender will be an essential component of the application, and its design and implementation will require careful consideration of the data and algorithms. Generally speaking, the steps for both Cosine Similarity and Pearson Correlation algorithms look like this:

1. Load Data: The first step is to load the ratings data, which contains information about how users have rated different books. The data is typically stored in a database or a file.

2. Preprocessing: The ratings data needs to be preprocessed to create a matrix that represents the ratings given by users to books. This matrix is used to calculate similarities between books and users.

3. Similarity Calculation: When Pearson Correlation is used to calculate the similarity between a user and all other users in the dataset, it involves computing the Pearson Correlation Coefficient between the ratings of two users for each book that they have both rated. When Cosine Similarity is used, it implies computing the cosine of the angle between the ratings vectors of two users for each book that they have both rated.

4. Nearest Neighbors: The users with the highest similarity to the target user are identified as the nearest neighbors. These neighbors are used to generate recommendations for the target user.

5. Recommendation Generation: For each book that the target user has not yet rated, a prediction is made for the rating that the user would give that book. This prediction is based on the ratings of the nearest neighbors for that book, weighted by their similarity to the target user.

6. Top-N Recommendations: The books with the highest predicted ratings are recommended to the target user. The number of recommendations made is typically limited to a fixed number (N), which is specified by the user or the application.

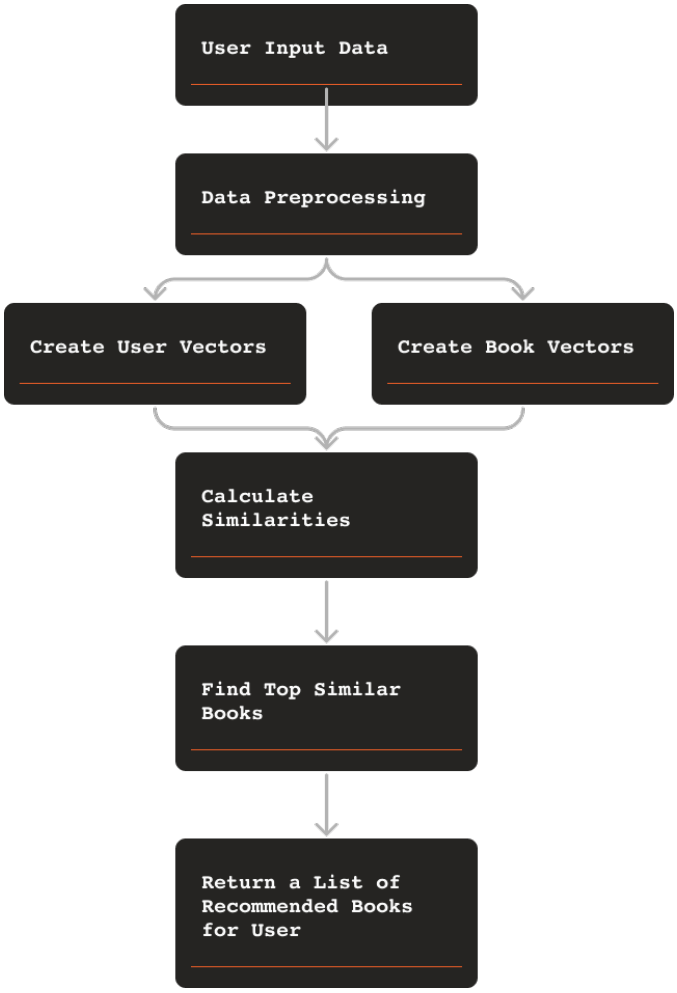

Figure 1. Recommender Process

User Authentication and Profile Creation

The second functional requirement is to provide users with an easy experience of creating a profile and logging in. Users will be required to register an account to access the recommendation engine and personalized features. During the registration process, users will provide basic information such as their name, email address, and password. Once registered, users will have access to their personalized book recommendations. The application will ensure secure authentication by using password hashing and secure storage of user data.

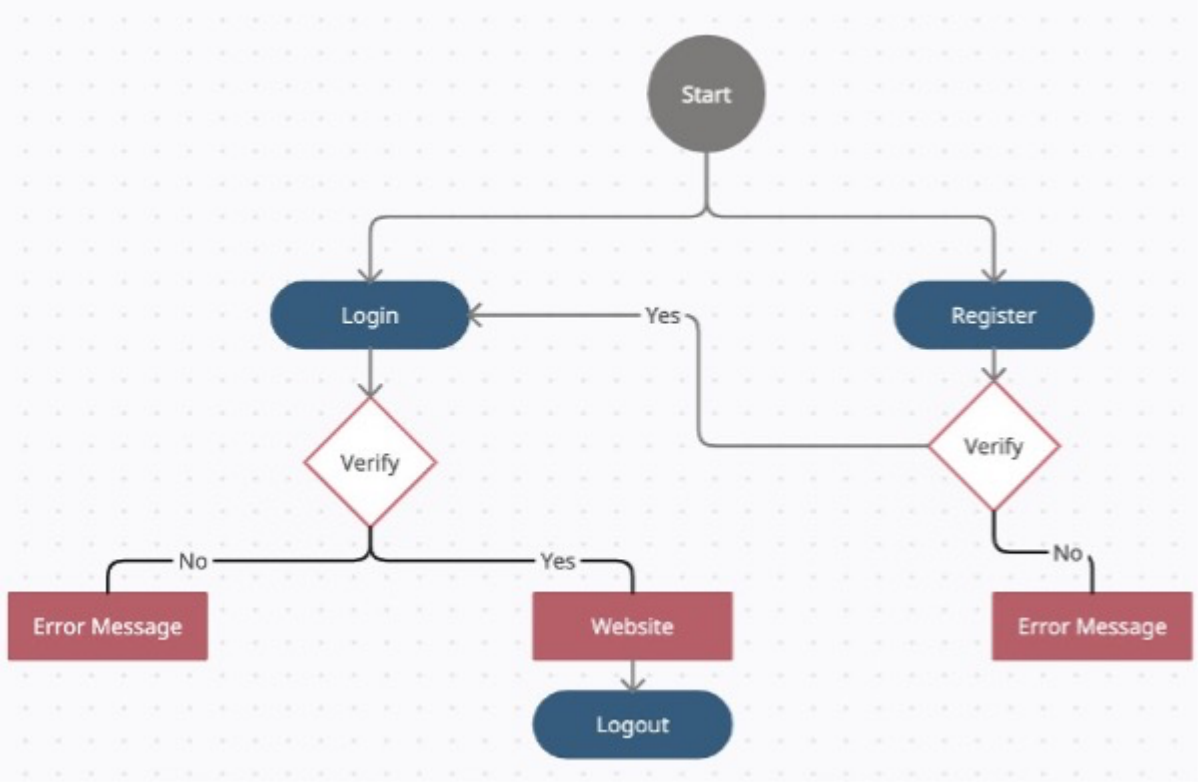

Figure 2. User login Activity Diagram

Microsoft Identity Framework (MIF) provides a large set of tools that are easy to integrate into websites, offering a secure and reliable means of managing user authentication and authorization. The framework incorporates key components such as authentication, authorization, and security which are utilized in NextPage.

Search Functionality

The third functional requirement is to allow users to search for books within the application. The search functionality will allow users to enter keywords or book titles to find specific books or discover new ones. The search results will provide users with book details such as the author, title, description, cover image, and user ratings. The search results will also include recommendations based on the user's search history and preferences. The search functionality will be intuitive and easy to use, providing users with a seamless experience when discovering new books.

I am using the Knuth-Morris-Pratt algorithm in order to implement search functionality. Series of steps in order to implement the search is as follows:

1. User enters a search query into the application.
2. The search query is processed and the Knuth-Morris-Pratt algorithm is used to search the database for books that match the query.
3. The algorithm returns a list of books that match the search query.
4. Book details, such as author, title, description, cover image, and user ratings, are retrieved for each book in the list.
5. The search results, along with book details, are displayed to the user.
6. Based on the user's search history and preferences, the application provides recommendations for related books to the user.
7. The recommended books are displayed to the user along with the search results.

Minimized version of this process is shown in the Figure 3 down below.

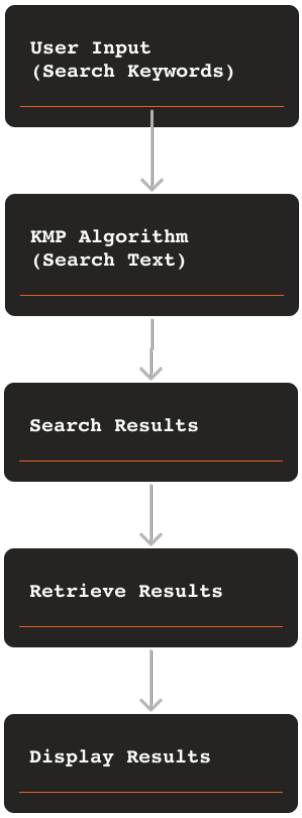

Figure 3. KMP Search Process

Performing CRUD operations

Performing CRUD (Create, Read, Update, Delete) operations with three entities in a web application is a common functional requirement for many applications and for this one specifically. CRUD Operations on Books, Users, Locations and Ratings Entities in NextPage include:

1. Create a new book, user, rating or location:
    - A user should be able to create a new book, providing information such as title, author, publisher, year of publication, and genre.
    - A user should be able to create a new user, providing information such as name, email, and password.
    - A user should be able to create a new location, providing information such as name, address, and contact details.
    - A user should be able to rate a book

2. Read/view an existing book, user, rating or location:
    - A user should be able to view a list of all books, users, or locations that exist in the database.
    - A user should be able to search for a specific book, user, or location using relevant search criteria.
    - A user should be able to view the details of a specific book, user, or location by clicking on it in the list.
    - A user should be able to see the rating given by them to the book

3. Update an existing book, user, rating or location:
    - A user should be able to edit the information of an existing book, such as changing the title, author, or publisher.
    - A user should be able to edit the information of an existing user, such as changing their name or email address.
    - A user should be able to edit the information of an existing location, such as changing the address or contact details.
    - A user should be able to update the rating

4. Delete an existing book, user, or location:
    - A user should be able to delete an existing book from the database.
    - A user should be able to delete an existing user from the database.
    - A user should be able to delete an existing location from the database.

5. Assign a book to a location:
    - A user should be able to assign a book to a specific location in the database.
    - A user should be able to view the location where a specific book is available.

6. Mark a book as borrowed:
    - A user should be able to mark a book as borrowed from a specific location.
    - The book should be removed from the list of available books at that location until it is returned.

7. Mark a book as returned:
    - A user should be able to mark a borrowed book as returned.
    - The book should be added back to the list of available books at the location from which it was borrowed.

```
function Create (bind properties)
        get current user;
        if (model IS valid)
                add book to db context;
                await save changes async;
        redirect to page;

function Edit (bind properties)
        if (id IS NOT book id) return error;
        if (model IS valid)
                try
                        update book at db context;
                        await save changes async;
                catch (exceptions)
        return page;

function Delete (id)
        if (db context table Books IS null) return error message
```

```
        var book = await find book(id);
        if (book IS NOT null)
                remove book from db context;
                await save changes to db context;
        return redirect to page;

function Details(id)
        if (id IS null OR db context Books table IS null) return error;
        var book = await find book(id);
        if (book IS null)
                return error;
        return View(book);
```

Database

In order to properly manage data and perform mentioned operations, the data should be stored in a suitable place. The database is stores of MSSql server. For the purpose of this project, the Entity Framework is used to create migrations and establish a database that is capable of storing the entities. The code-first approach being implemented in this project allows for easy modification of the existing database schema by creating new migrations. Additionally, the Entity Framework provides the ability to run migrations through command line, which not only ensures efficiency but also streamlines the process of managing and updating the database schema.

Interaction With Locations

Books can be borrowed by users and returned by other users at specific locations. These locations act as intermediaries between users and are managed by the user who registered them. In order to ensure accountability and track the borrowing and returning of books, locations must confirm when a user has taken or returned a book.

To implement this functionality, the Book Controller will be updated with Take and Leave methods. These methods will allow users to take a book from a location or return it to a location. The location will then confirm the transaction and update the book's status accordingly. This process will help to ensure that the book is available for other users to borrow and that the location's inventory is up-to-date.

```
function LeaveAtLocation(string bookId, string locationId, string userId)
    var currentUser = get current user;
    if (from db context books OR locations OR users IS NULL)
        return error message;
    var book = await get book(bookId);
    var location = await get location (locationID)
    if (book IS NOT null AND location IS NOT null AND currentUser IS NOT null)
        book.CurrentlyStoredAtId = location.Id;
        book.CurrentlyStoredAt = location; book.CurrentlyHeldById = null;
        book.CurrentlyHeldBy = null; currentUser.BooksStored.Remove(book);
        location.BooksStored.Append(book);
        await save changes;
    return redirect to book details;

function TakeFromLocation(string bookId, string locationId)
    var currentUser = get current user;
    if (from db context books OR locations OR users IS NULL) error message;
    var book = await get book(bookId);
    var location = await get location(locationId);
    if (book IS NOT null AND location IS NOT null AND currentUser IS NOT null)
        book.CurrentlyStoredAtId = null;
        book.CurrentlyStoredAt = null; book.CurrentlyHeldById = currentUser.Id;
        book.CurrentlyHeldBy = currentUser; currentUser.BooksStored.Append(book);
        location.BooksStored.Remove(book);
        await save changes;
    return redirect to book details;
```

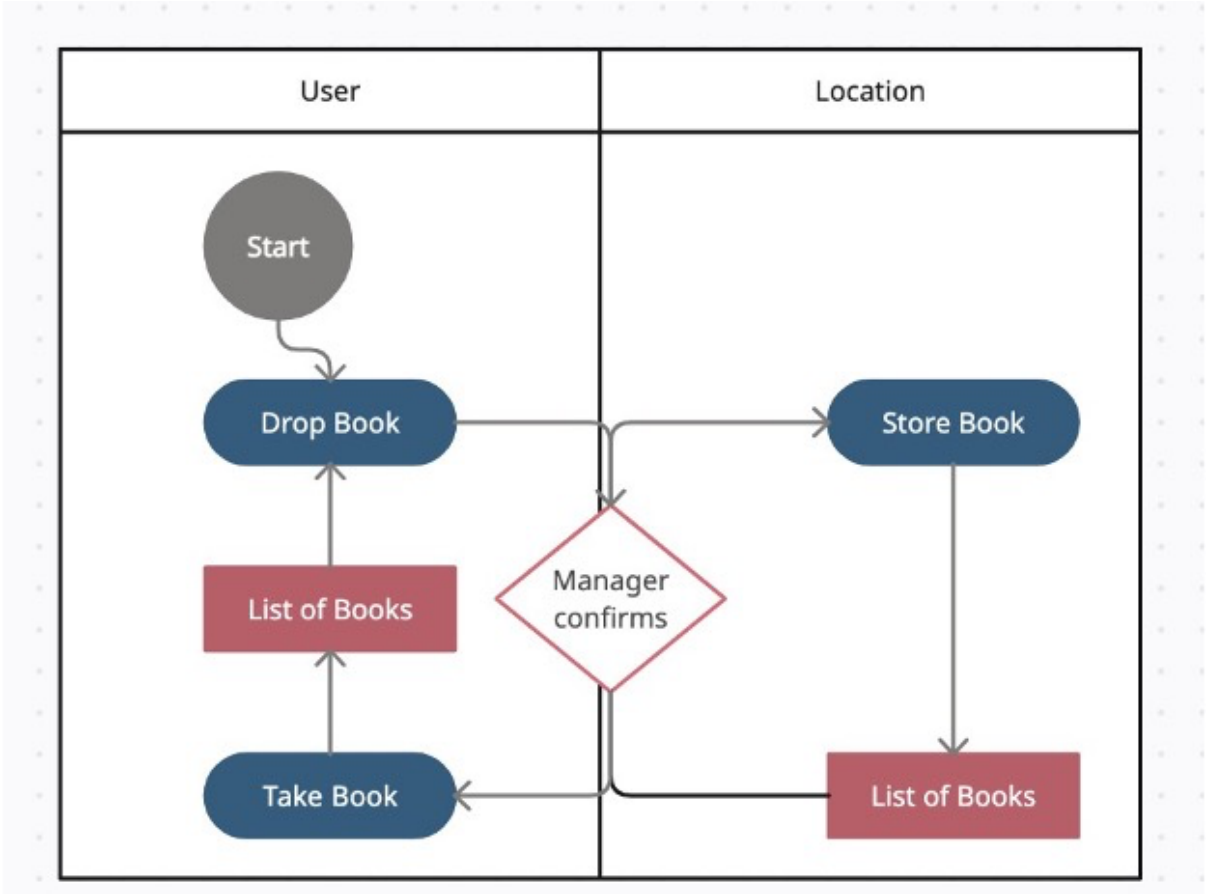

Figure 4. User-Location Activity Diagram

Book Exchange

In order to facilitate the borrowing and lending of books among users, it is useful to include the event of a user taking or leaving a book as a functional requirement. This requirement ensures that the system accurately reflects which books are currently in circulation and where they are located. Furthermore, it is as a way to track the movement of books between different users and locations, which can be further on developed to track the travel history of a specific book.

By incorporating the Take and Leave methods into the Book Controller, users can easily record when they have taken a book or when they have returned it to a particular location. This process not only ensures the accuracy of the book inventory but also allows users to easily browse the available books and their location, improving the overall experience.

Logical Interaction With Books

The user who creates the book is initially listed as the holder, until they decide to drop it off at a designated location. To facilitate the drop-off process, a bookmark with a unique code is printed and placed inside the book. This enables the next user to easily identify the book and enter the code into the app to indicate that they have taken it.

Moreover, every user can potentially have access to a list of available books, allowing them to easily identify which books can be taken and which cannot. This feature further enhances the usability of the web application, as it encourages users to share their books with others, and enables easy access to a wide range of reading materials. The interaction is shown in Figure 5 (UML Use Case Diagram).

Rating System

In order to provide users with an accurate representation of the quality of books within the application, a rating system is implemented. Users are able to rate any book within the application, thereby providing valuable feedback to other users. The Rating Entity is used to store UserId, BookId, CategoryId, RatingValue and a unique identifier of an event.

By storing these ratings, the application will be able to use this data in order to build a recommendation system that can suggest books based on a user's preferences and past behavior. As such, the recording and storage of rating data is of significant importance in order to ensure accuracy and effectiveness of the recommenders.

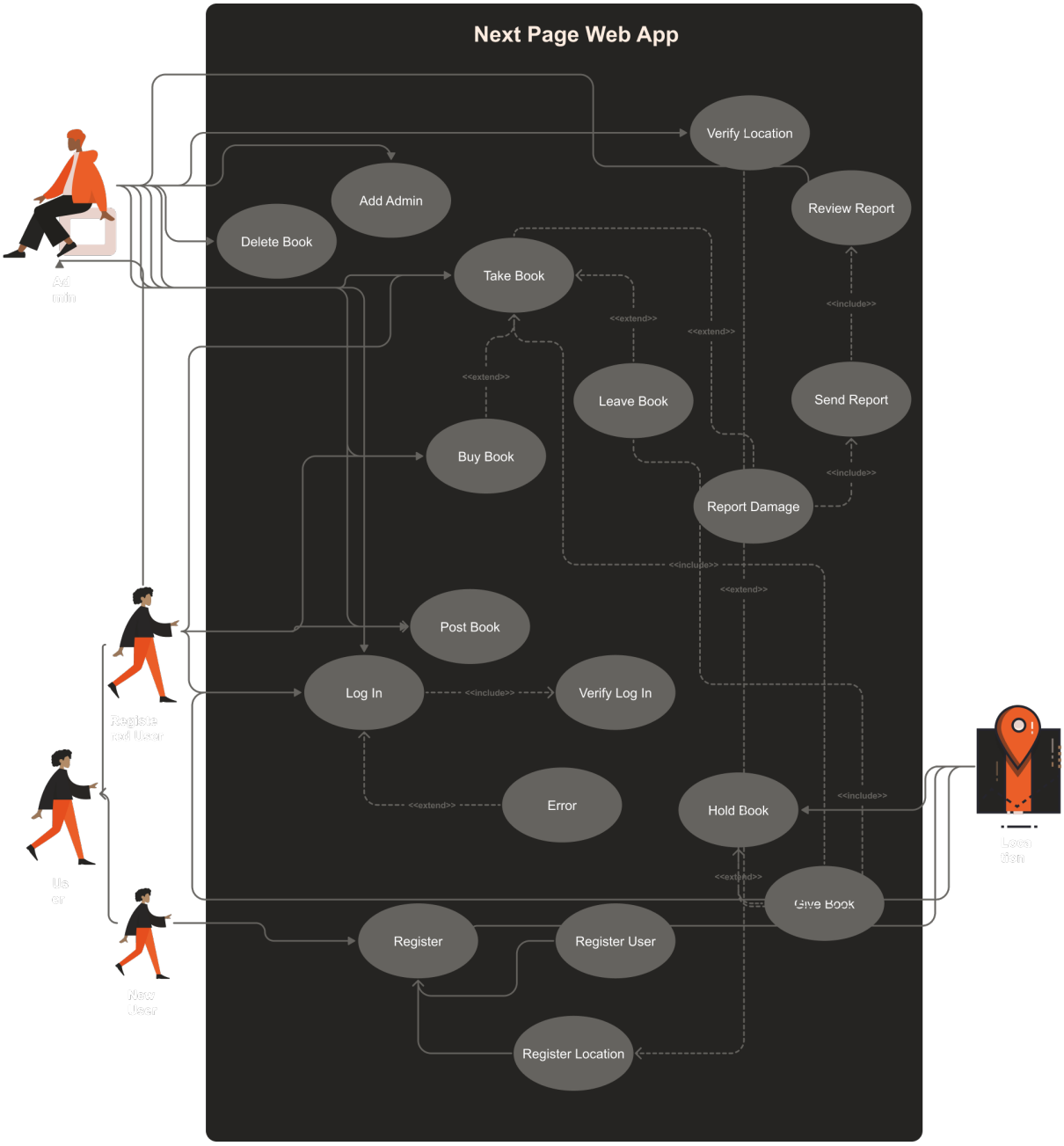

Figure 5. UML Use Case Diagram

Usability

The usability and user experience of the system is an important non-functional requirent for NextPage as it is a web application, potentially to target thousands of users. A simple and intuitive user interface will enable users to navigate the system

with ease and locate relevant books and recommendations effortlessly. In order to achieve this, a user-centric design approach will be adopted, with an intent to design the system from the potential user perspective. This will involve gathering feedback and requirements from users throughout the development process to ensure that the system meets their needs and expectations. Additionally, user testing and evaluation will be conducted to identify any usability issues or areas of improvement, and adjust the system to optimize the user experience. By prioritizing usability and user experience, the system can maximize user engagement and satisfaction, leading to increased adoption of the application.

Performance

To ensure a satisfactory user experience, the system must be capable of processing and managing large amount of data concerning book information and user data. This includes the efficient processing of user interactions, such as book ratings and, as well as the effective analysis of this data to generate personalized recommendations. The system's ability to provide accurate recommendations matters to avoid frustrating users with prolonged waiting times or delays. Therefore, the system's performance and response times must be optimized to ensure a smooth user experience.

Security

In order to promote the confidentiality of user data, the system must incorporate adequate security measures, as it is tangible for the purposes of this thesis. It is possible to employ encryption techniques for protecting sensitive data such as user login credentials and personal information. Moreover, the system should incorporate user authentication and authorization mechanisms to verify the identity of the user and their level of access to the system. Additionally, the use of secure frameworks, servers and protocols is promoted in the process of development of Next Page, to make sure data stay protected on multiple levels.

Scalability

Making sure that a system is designed to accommodate future growth and expansion is a significant aspect of its long-term success. In order to support increasing amounts of data and users, the system must be architected in a scalable way that can accommodate additional resources and capacity as needed. Additionally, the system should be flexible enough to accommodate new features and functionality that may be required in the future. This will require careful planning and design, as well as ongoing maintenance and support to ensure that the system remains adaptable and effective over time. .NET allows for it and promotes it, so choosing this particular framework to develop the application is a safe decision long-term. A scalable system will enable the application to adapt to the evolving needs and preferences of users, therefore, enhancing its value.

Limitations

Some useful features that will not be implemented in this project are:

1. A book's travelling history and possession will not be displayed to users.
2. The possibility of a book being lost will not be addressed in this project.
3. Payment will not be implemented, even though it could be the main feature that would help secure the book crossing event. However, it is a potential feature to be added in the future as the project progresses.
4. Admin functionality

There are certain challenges that may require specific solutions during the development of this system. These include:

1. Since this is a new web application, there may not be sufficient data to properly train the machine learning model. As a solution, the dataset will be populated with dummy data in order to test the recommendation engine.

2. A problem of making appropriate predictions for new users may arise with the use of a recommender system. To address this, users will be asked to select among a list of popular books they have enjoyed in the past to generate initial predictions.

It is important to consider these limitations and challenges during the development process to ensure that the final product meets the needs of its users and is a successful addition to the book crossing community.

3. Design of Software Solution

The software design for the thesis project will be presented in this section. The software provides a roadmap for the development process, ensuring that the final product meets the requirements and specifications of the project. It lays the foundation for the implementation of the proposed solution to the research problem. The design should be well-structured and organized to ensure that the final product is reliable, maintainable, and scalable.

This section will include a detailed description of the software architecture, special algorithms used, user interface, software components, and the services offered by each component. The architecture will be described and further demonstrated in the form of UML component diagrams, and E-R diagrams will be provided for the database context. Additionally, a UML deployment diagram will be presented, illustrating the physical deployment of the software components in the system.

Security and reliability considerations are also of a deep importance, and this section will describe the measures taken to ensure that the application is secure and reliable. Encryption techniques used for passwords and anti-hacking precautions implemented in the application will also be discussed.

Software Design Overview

The design of the software solution for the application follows a multi-layered architecture with an emphasis on the Model-View-Controller (MVC) pattern. The architecture consists of different components that work together to provide the required functionalities. These components include the user interface, data layer, recommender module, and search module.

The View component is responsible for displaying information to the user and capturing user input. The design of the user interface follows the principles of usability and provides an intuitive and easy-to-use interface for users to interact with the application.

The data layer component manages the storage and retrieval of data from the database. This component uses Entity Framework, which is an Object-Relational Mapping (ORM) tool that simplifies database access by mapping database tables to classes in the application.

The recommender module is providing personalized book recommendations to users based on their reading preferences. This module uses collaborative filtering, a popular approach in recommendation systems, to generate recommendations.

The search module provides users with the ability to search for books within the application. This module uses the Knuth-Morris-Pratt algorithm, which is an efficient string-matching algorithm, to search for books based on keywords or book titles.

The software architecture addresses the requirements and functional specifications outlined in the earlier sections of the report. The multi-layered architecture provides a clear separation of concerns between different components and promotes modularity, which enables easier maintenance and scalability of the application.

The MVC pattern is particularly well-suited for web applications as it promotes the separation of user interface, data, and business logic. This makes it easier to manage and test each component of the application separately.

The View component provides a seamless experience for users and allows them to easily find and interact with the functionalities of the application. The data layer component ensures that data is stored and retrieved efficiently from the database, while the recommender and search modules provide personalized recommendations and efficient search functionalities respectively.

Software Architecture

The software architecture of the application is based on the Model-View-Controller (MVC) design pattern, which separates the application logic into three interconnected components - models, views, and controllers. The architecture is further extended to include multi-layered architecture, where each layer has its own

specific responsibility, which improves the scalability and maintainability of the application.

The MVC design pattern follows the Separation of Concerns principle, which means that each component has a specific role and responsibility. The model layer is responsible for the data and business logic of the application, the view layer is responsible for the user interface, and the controller layer is responsible for handling user requests, updating the model layer, and selecting the appropriate view to render the response.

The multi-layered architecture extends the MVC design pattern to include additional layers such as services and data access layers. The services layer encapsulates the business logic of the application and provides a layer of abstraction between the controllers and the data access layer. The data access layer provides access to the database and encapsulates the database logic.

The controllers are responsible for handling user requests and mapping them to the appropriate services or models. They act as an intermediary between the view and the model layer, selecting the appropriate view to render the response. The controllers in this application are responsible for handling CRUD operations on entities such as Account, Book, and Location.

The services layer provides the application's main functionality, including the recommender and search functionality. The recommender service is responsible for providing book recommendations based on user preferences and book ratings. The search service is responsible for enabling users to search for books by keywords, book titles, or authors. The services layer interacts with the data access layer to retrieve and update data from the database.

The data access layer provides access to the database and encapsulates the database logic. It communicates with the services layer and the database to retrieve and update data. The data access layer uses Entity Framework, which is an object-relational mapping (ORM) framework for .NET, to map the application's domain models to the database tables.

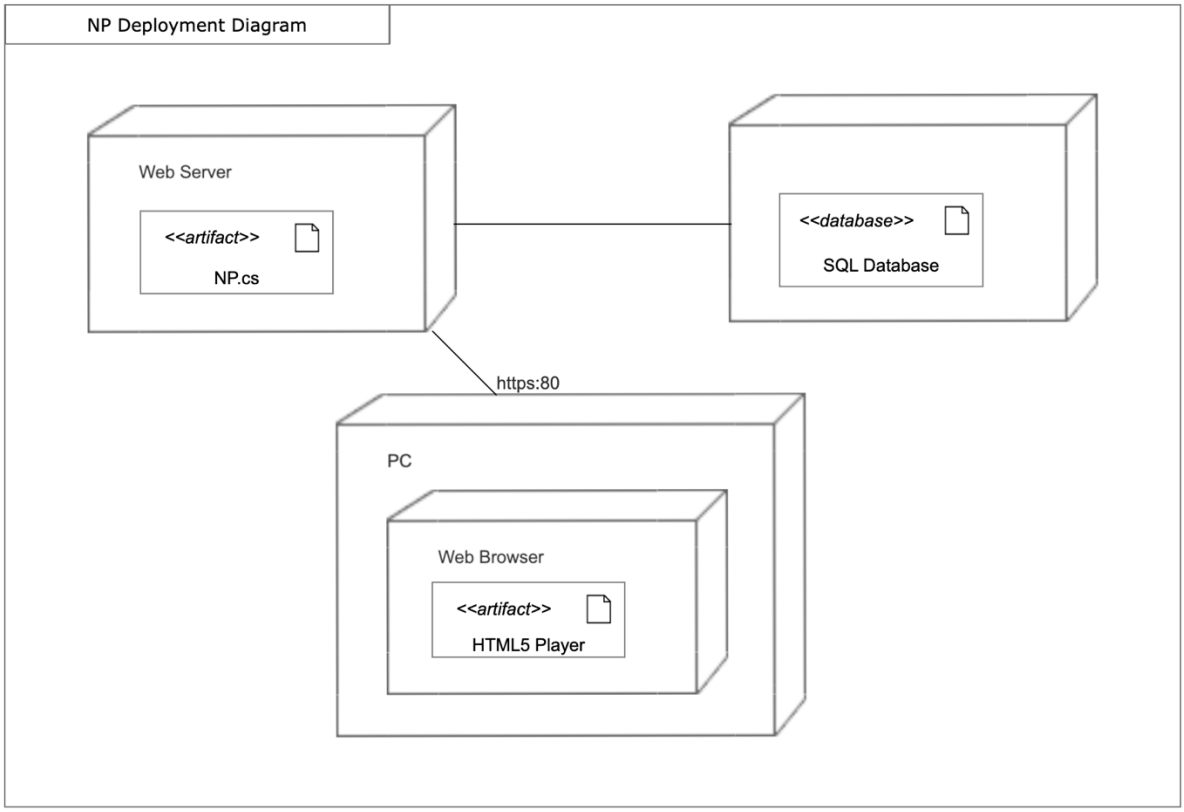

Figure 6. UML Deployment Diagram

User Interface

The user interface of a web application is a crucial aspect of its design, as it determines how users interact with the system and influences their overall experience. In the context of the Next Page application, the user interface is designed to be intuitive and easy to use, featuring standard elements commonly found in web applications.

Navigation menus are an essential component of any web application, allowing users to quickly and easily move between different sections of the system. Forms are another critical element, enabling users to input data into the system and interact with its features. Buttons are used to trigger actions within the application, such as submitting a form or navigating to a new page. Text boxes are utilized to display information to the user, such as book descriptions or user reviews, or to allow the user to enter text for searching or filtering purposes.

Images are a powerful tool for enhancing the user experience, making the books and locations more recognizable and visually appealing. The use of book covers and location photos can provide additional context and help users make informed decisions about which books to borrow and where to find them.

User interface of the Next Page application is designed to provide users with a seamless and enjoyable experience, while also ensuring that they can easily access the system's features and functionality.

Special Algorithms

Singular Value Decomposition (SVD)
SVD is a matrix factorization technique that decomposes a matrix into three matrices: U, S, and Vt. The U and Vt matrices are orthogonal matrices, and the S matrix is a diagonal matrix that contains the singular values of the original matrix. SVD is a widely used technique in machine learning and data analysis, and it has many applications, such as data compression, dimensionality reduction, and matrix approximation. In the context of collaborative filtering, SVD can be used to decompose the user-book rating matrix into latent features, which can then be used to make recommendations.

One limitation of the SVD algorithm is that it requires the entire dataset to be stored in memory, which can be a problem when working with large datasets. In such cases, it can be challenging to fit the entire dataset into memory, which may lead to computational inefficiencies or even memory errors. More scalable versions of the SVD algorithm exist, such as stochastic gradient descent (SGD) and randomized SVD.

Another limitation of the SVD algorithm is its sensitivity to missing data. If the input data contains many missing values, it may be challenging to perform accurate factorization. In such cases, imputation techniques may be used to estimate missing values before applying the SVD algorithm.

Also, the SVD algorithm may not always be the best choice for recommendation systems, especially when dealing with sparse data. In such cases, other algorithms such as neighborhood-based methods, content-based filtering, or hybrid approaches may be more effective.

However, for the purposes of this project, I implemented this particular algorithm as I did not plan on having an extraordinary amount of data.

Matrix Preprocessing

Matrix preprocessing refers to a set of techniques that are applied to a matrix before it is used for analysis. Preprocessing can involve removing outliers, filling in missing data, normalizing the data, or transforming the data in some other way. The goal of preprocessing is to improve the quality of the data and make it more suitable for analysis. In the context of collaborative filtering, preprocessing can be used to clean the user-book rating matrix by removing outliers and filling in missing values. This can help to improve the accuracy of the recommendations that are generated by the collaborative filtering algorithm.

Cosine Similarity

The project utilizes a special algorithm called cosine similarity to recommend articles to users. This algorithm works by calculating the cosine similarity between two articles, which measures the similarity of their content based on the angle between their feature vectors.

The feature vectors are created using a technique called term frequency-inverse document frequency (TF-IDF), which assigns weights to words based on their frequency in a document and their frequency across all documents in the dataset. This allows the algorithm to account for the fact that some words may be more important than others in determining the content of an article.

To recommend articles to a user, the algorithm first calculates the cosine similarity between the user's previous article views and all the other articles in the dataset. It

then selects the top N articles with the highest similarity scores and recommends those to the user.

This algorithm has several advantages over other recommendation algorithms. First, it does not require any user data other than the user's previous article views, which means it can be used in situations where user privacy is a concern. Second, it is computationally efficient, which means it can scale to very large datasets. Finally, it is effective at recommending articles that are similar in content to the user's previous article views, which can lead to a more personalized and engaging user experience.

Pearson Correlation

Pearson correlation is a widely used algorithm in the field of computer science and machine learning for collaborative filtering. It is a statistical method that measures the linear relationship between two variables, in our case, between users and their ratings of items.

The Pearson correlation coefficient (r) is calculated by taking the covariance of two variables (X and Y) and dividing it by the product of their standard deviations. The result is a value between -1 and 1, where -1 indicates a perfect negative correlation, 0 indicates no correlation, and 1 indicates a perfect positive correlation.

In the context of collaborative filtering, Pearson correlation is used to calculate the similarity between two users based on their ratings of items. It is assumed that users who have similar preferences for certain items will also have similar preferences for other items. Therefore, the algorithm calculates the Pearson correlation coefficient between the ratings of each pair of users, and then recommends items that have been highly rated by users with similar preferences.

One advantage of the Pearson correlation algorithm is that it is not affected by the mean rating of users. This means that users who tend to rate items higher or lower than others will not bias the similarity calculation. Another advantage is that it is a more accurate measure of similarity than cosine similarity, especially in situations where the data has a linear relationship.

However, Pearson correlation also has some limitations. It assumes that the data is normally distributed, which may not always be the case in real-world situations. It is also sensitive to outliers, which can affect the similarity calculation. Therefore, it is important to preprocess the data and remove outliers before using Pearson correlation.

Knuth-Morris-Pratt

The Knuth-Morris-Pratt (KMP) algorithm is a string searching algorithm that is used to find occurrences of a particular string pattern within a larger text. The algorithm was developed by Donald Knuth, James Morris, and Vaughan Pratt in 1977 and is considered to be one of the most efficient string search algorithms available.

The KMP algorithm's main advantage is that it avoids the need to backtrack while searching for the pattern in the text, which significantly improves its efficiency. This is achieved by using the information stored in the pre-processed table to determine how much the pattern can be shifted without missing a match.

When searching for a pattern in a text using the KMP algorithm, the algorithm moves through the text one character at a time, comparing each character to the corresponding character in the pattern. If a mismatch is found, the algorithm uses the pre-processed table to determine how much the pattern can be shifted without missing a match, thereby avoiding the need to backtrack.

In the context of the book class in the Next Page web app, the KMP algorithm can be used to implement text search functionality based on the book's name or author. When a user enters a search query in the search bar, the KMP algorithm can be used to search through the book titles and author names to find matches.

Boyer-Moore and Rabin-Karp algorithms are other commonly used text search algorithms. The Boyer-Moore algorithm is known for its fast average-case performance, but its worst-case performance is still O(mn) when the pattern contains many occurrences of the last character. The Rabin-Karp algorithm uses a rolling

hash function to compare the pattern with the text and can be used for multiple pattern search as well. However, it suffers from hash collisions and has a worst-case time complexity of O(mn) when the hash function is poorly chosen or there are many spurious hits.

Consequently, I chose the KMP algorithm over other search algorithms because of its superior time complexity of O(m+n) and its ability to efficiently search for patterns in text, even for patterns with repeating characters. In addition, the KMP algorithm is easy to understand, implement, and test, which is important for maintainability.

Both recommender system and search system are plugged into a Book Controller as represented in the figure below.

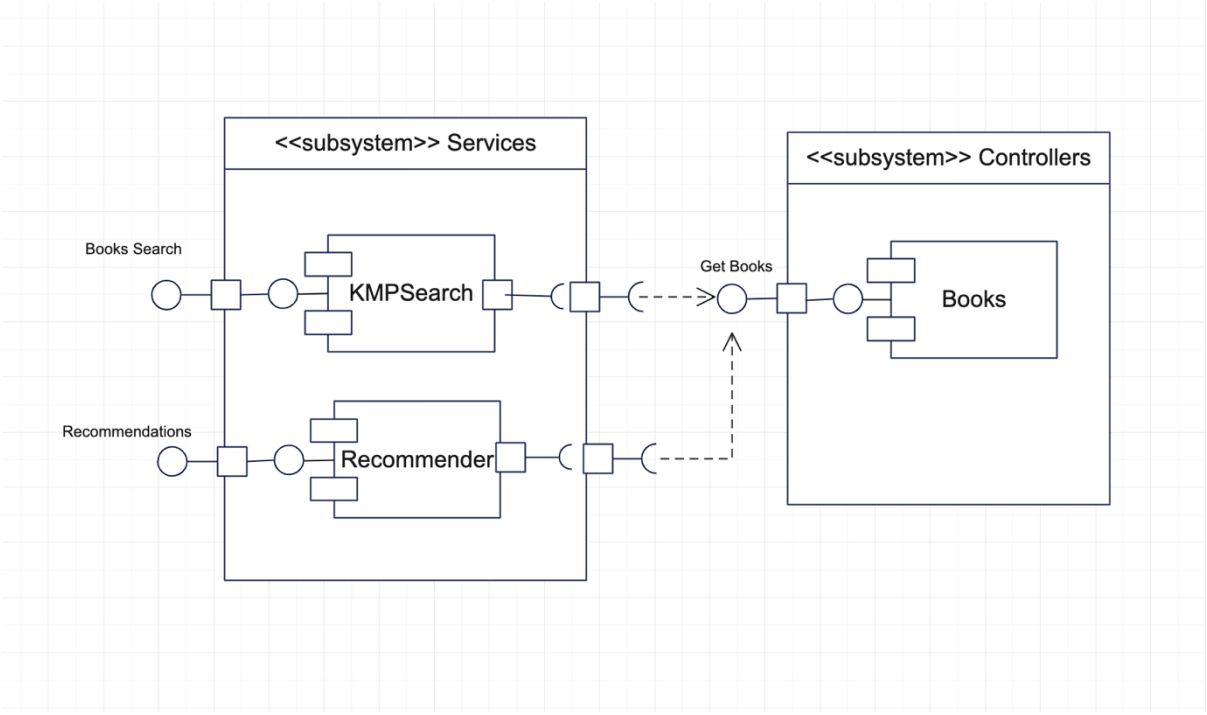

Figure 7. UML Component Recommender And Search Diagram

Security And Reliability

One of the primary security features that has been implemented in the application is the encryption of user passwords. Passwords are encrypted using a secure one-way

hashing algorithm, which guarantees that even if an unauthorized individual gains access to the database, the original password cannot be decoded. Furthermore, password complexity rules are enforced and users are required to select strong passwords to enhance security.

To ensure the reliability of the application, several measures have been taken. Regular backups of the database and application data are conducted to prevent data loss or corruption and allow the application to be restored to its previous state. Moreover, error handling and logging mechanisms have been implemented to identify and diagnose any issues that may arise and to rapidly resolve them.

Entity Relationships

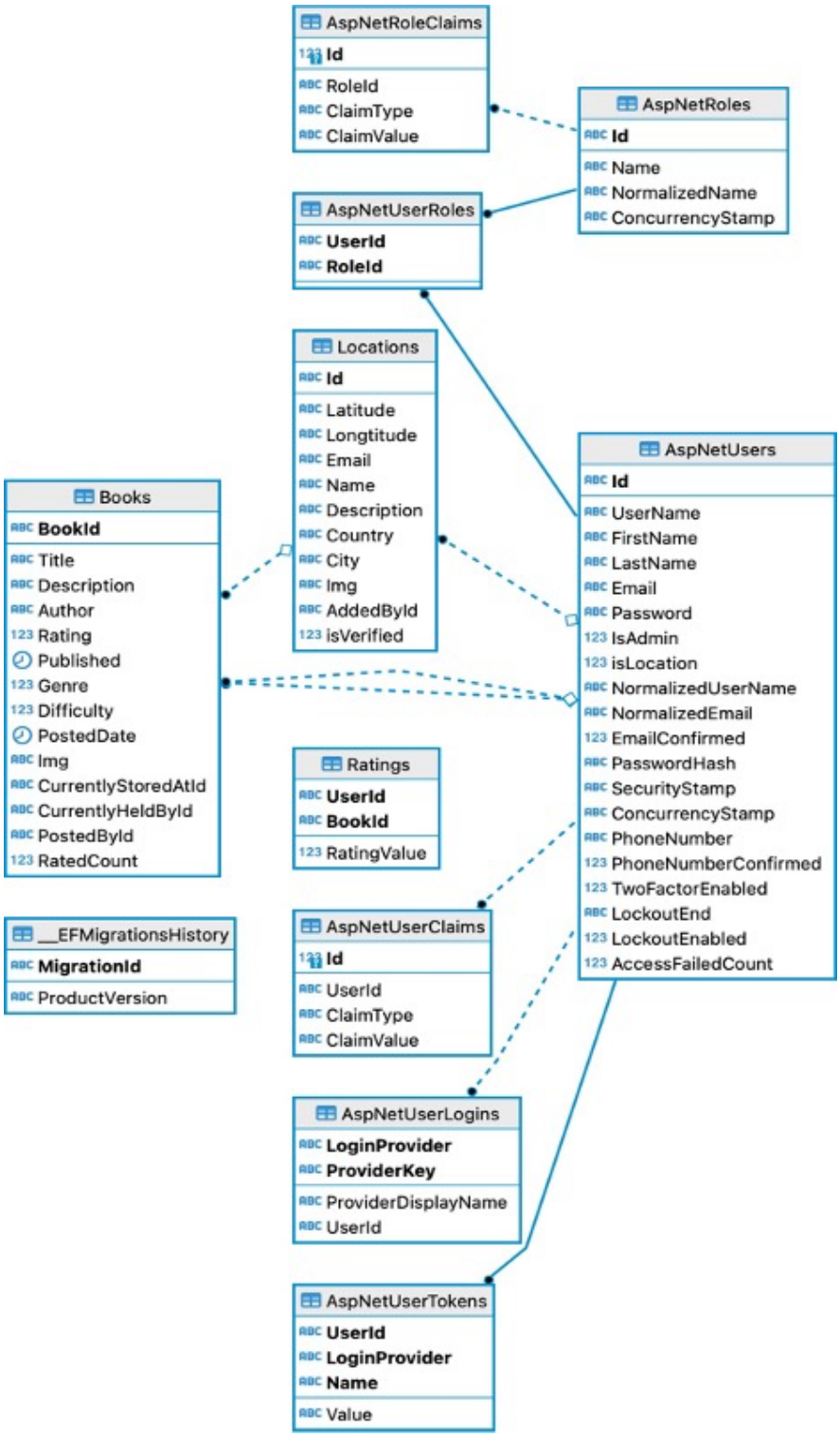

Figure 8. E-R Diagram

4. Implementation

Computing Platfrom

In the context of this project, a web browser can be considered a computational tool that serves the purpose of providing a user interface that is easily accessible to all users. Unlike traditional software applications, web browsers do not require installation and can be accessed through a variety of devices with internet connectivity. Although registration is preferred to access the full range of functionality, it is not always required.

To ensure the efficient and sustainable functioning of the application, various frameworks have been incorporated into the design. These frameworks are intended to facilitate scalability and provide access to essential libraries, particularly those that are provided by the .Net framework. The use of these frameworks enables the development of a reliable and robust system that can handle complex computational tasks while minimizing the risk of errors or system failures. By leveraging these tools and technologies, the project aims to deliver a high-quality user experience and achieve its objectives in an effective and efficient manner.

Frameworks

1. Microsoft.AspNetCore

   ASP.NET Core is a popular open-source web framework that facilitates the development of modern, cloud-based, and internet-connected applications. Its modular, lightweight, and cross-platform architecture allows developers to create web applications that can run on various operating systems, such as Windows, Linux, and macOS. This framework offers a broad range of features, including a robust dependency injection system, integrated middleware, and support for cloud-based hosting providers. Moreover, it features a built-in web server that allows developers to rapidly set up and test their applications. Overall, ASP.NET Core is a versatile framework that enables developers to build high-performance web applications that are scalable, secure, and easy to maintain. Its wide range

of features and cross-platform capabilities make it a popular choice for developing web applications.

2. Microsoft.AspNetCore

Http The Microsoft.AspNetCore.Http library provides a set of classes that assist developers in working with HTTP requests and responses. The library includes classes such as HttpRequest, HttpResponse, and HttpClient, which provide a rich set of methods and properties for managing HTTP headers, handling cookies, managing authentication, and more. These classes are beneficial for obtaining user data in order to manipulate and present relevant information for the user. By utilizing the classes from this library, developers can efficiently handle HTTP requests and responses, which are essential components in web application development. This library is particularly useful for this project, as it allows for effective communication between the user and the web application.

3. Microsoft.EntityFrameworkCore

In this thesis, the Microsoft.EntityFrameworkCore library plays a critical role in facilitating the interaction with databases through an object-oriented approach rather than relying on raw SQL. This framework offers a mechanism for mapping relational database tables to corresponding C# objects and vice versa.

The use of this library allows for the abstraction of database-specific implementation details from the application logic. As a result, the project can focus on the business logic and reduce the time and effort required for writing and maintaining low-level database code.

The Microsoft.EntityFrameworkCore library offers a range of additional features that further enhance the development experience in this project. These features include support for database migrations, automatic tracking of changes to the database, and lazy loading of related entities. The library also supports LINQ (Language Integrated Query) for querying databases, which simplifies the process of retrieving and filtering data from the database. The use of Microsoft.EntityFrameworkCore library in the Next Page project is an essential tool to promote maintainability, flexibility, and scalability of the application.

4. AspNet.EntityFrameworkCore

The AspNet.EntityFrameworkCore library is a part of the ASP.NET Core framework and provides a set of tools and services for working with databases using the Entity Framework Core (EF Core):

- DbContext: A base class for creating a database context class, which represents a session with the database and provides a way to query and save data.
- DbSet: A class that represents a table or view in the database and provides a way to query and modify data.
- Migration: A service that enables the project to manage database schema changes using code-first migrations. By allowing the creation of database schema changes through code, it provides a flexible way to manage database changes in a project.
- Repository: A pattern for encapsulating data access logic in a single place, making it easier to test and maintain. This approach helps to reduce code duplication and improves code maintainability.

The AspNet.EntityFrameworkCore library provides several benefits to this project. The use of DbContext and DbSet classes allows the project to interact with the database using an object-oriented approach, thereby simplifying the management of data. Additionally, the use of code-first migrations enables the project to easily manage database schema changes. Finally, the use of the Repository pattern provides a more structured approach to data access, reducing code duplication and improving code maintainability.

5. Microsoft.EntityFrameworkCore.SqlServer

This is a NuGet package designed to extend the capabilities of Entity Framework Core (EF Core) by offering support for Microsoft SQL Server databases. The package provides a set of classes and services that enable developers to seamlessly interact with SQL Server databases using the EF Core ORM framework. This package offers a robust and efficient solution for developers who need to work with SQL Server databases in their applications.

In the current project, Microsoft.EntityFrameworkCore.SqlServer proved to be a valuable tool as it allowed the creation of a database server for the application. The package facilitated the interaction with the SQL Server database through EF Core.

6. Microsoft.EntityFrameworkCore.Design

Within the context of this project, the Microsoft.EntityFrameworkCore.Design package provides invaluable design-time functionality for working with Entity Framework Core (EF Core). This package includes a range of tools that are used to create and manage database migrations, scaffold database contexts, and generate code from database schemas. The ability to work with these tools directly from within the project simplifies the development process, saves time, and once again promotes maintainability.

7. Microsoft.EntityFrameworkCore.Tools

The Microsoft.EntityFrameworkCore.Tools package encompasses the command-line interface (CLI) of EF Core that enables the creation and management of database migrations, scaffolding of database contexts, and execution of EF Core commands through the command prompt or terminal. This feature is especially advantageous for this project, given that the development is done on a Mac OS system where the ability to run terminal commands is essential.

8. MathNet.Numerics

This project utilized the MathNet.Numerics library to improve the efficiency and accuracy of data preprocessing in a recommendation system. The library provided a set of pre-built functions for numerical computation, including matrix and vector operations, which were utilized in the Singular Value Decomposition (SVD) process of the recommendation system.

The SVD algorithm decomposes the user-book rating matrix into three smaller matrices, which represent the user preferences, item attributes, and their interactions. The MathNet.Numerics library was used to perform the SVD operation on the rating matrix, providing efficient and accurate computation of the

U, S, and Vt matrices. These matrices were then used to generate recommendations for users based on their historical ratings.

Additionally, the library was utilized to preprocess the data by removing outliers and filling in missing data. The library's functions provided an efficient way to calculate the mean and standard deviation of each column, which were used to identify and remove outliers. The library also provided a function to fill in missing data using column means, improving the accuracy of the recommendation system by ensuring that all data points were accounted for.

Using the MathNet.Numerics library provided several benefits over creating the functions from scratch. Firstly, it significantly reduced development time by providing pre-built, optimized functions for numerical computation. This allowed the project to focus on the recommendation system's core functionality and improve the system's accuracy, rather than spending time on implementing basic numerical operations. Secondly, the library ensured that the numerical computations were accurate and efficient, reducing the risk of errors and improving the system's overall performance.

Data Preprocessor

The GetSVDData() method is responsible for performing Singular Value Decomposition (SVD) on a user-book rating matrix. This matrix contains the ratings given by each user to each book they have rated. The method first retrieves all the ratings from the database using the _context.Ratings.ToList() method. It then retrieves all the unique user and book ids using LINQ's Distinct() method. Next, two dictionaries are created to map each user and book id to their corresponding index in the rating matrix. The method then creates a DenseMatrix object with dimensions users.Count x books.Count, and fills it with the rating values for each user-book combination. Finally, the SVD algorithm is applied to this matrix using the Svd() method provided by the MathNet.Numerics.LinearAlgebra library. The resulting U, S, and Vt matrices are then used to reconstruct the original rating matrix, and a list of Rating objects is returned.

The snippet of the code with main functionality of this method is down below:

```
// Create a user-book rating matrix
var ratingMatrix = DenseMatrix.Create(users.Count, books.Count, 0.0);

foreach (var rating in ratings)
{
    int userIndex = userIndexDict[rating.UserId];
    int bookIndex = bookIndexDict[rating.BookId];
    ratingMatrix[userIndex, bookIndex] = rating.RatingValue;
}

// Perform Singular Value Decomposition (SVD)
var svd = ratingMatrix.Svd();

// Get the U, S, and Vt matrices from SVD
var U = svd.U;
var S = svd.S;
var Vt = svd.VT.Transpose();

// Create a list of Rating objects
var ratingList = new List<Rating>();

for (int i = 0; i < users.Count; i++)
{
    var userId = users[i];

    for (int j = 0; j < books.Count; j++)
    {
        var bookId = books[j];
        var ratingValue = ratingMatrix[i, j];

        ratingList.Add(new Rating
        {
            UserId = userId,
```

```
                BookId = bookId,
                RatingValue = ratingValue
            });
        }
    }

    // Return the list of Rating objects
    return ratingList;
```

However, in order to implement this method, a number of other complementary methods were added to the DataPreprocessor class.

The RemoveOutliers method takes a matrix of numerical data and a z-score threshold as input, and returns a new matrix with any outliers removed. To do this, it calculates the mean and standard deviation of each column, and then uses these values to calculate the z-score for each data point. If the absolute value of the z-score is greater than the z-score threshold, the value is considered an outlier and is replaced with a zero in the new matrix. Otherwise, the original value is kept.

The FillMissingData method takes a matrix of numerical data with missing values (represented by zeros) as input, and returns a new matrix with the missing values filled in with the column mean. To do this, it calculates the mean of each column, excluding any zeros, and then replaces any zeros in the column with this mean value in the new matrix. An example of how to use this method in code is:

```
var preprocessor = new DataPreprocessor();
var data = DenseMatrix.OfArray(new double[,]
{
    { 1.0, 2.0, 3.0 },
    { 4.0, 0.0, 6.0 },
    { 7.0, 8.0, 0.0 },
    { 0.0, 11.0, 12.0 }
});
var processedData = preprocessor.FillMissingData(data);
```

Pearson Correlation

The GetRecommendations method is the main method that generates book recommendations for a given user. It first retrieves all the ratings made by the user from the database, and then retrieves all ratings for all books from the GetSVDData method of the _dataPreprocessor object. The method then calls the CreateBookRatingsDictionary method to create a dictionary of all book ratings with the respective features. It then calls the GetUserVector method to create a dictionary of the user's feature values for all books they have rated.

The GetBookVectors method is then called to create a dictionary of feature values for all books. Finally, the CalculatePearsonSimilarities method is called to calculate the Pearson correlation between the user vector and each book vector.
The method then selects the numRecommendations top similar books, retrieves the books from the database, and returns only the ones that the user has not already rated.

The CreateBookRatingsDictionary method takes in a list of Rating objects allRatings. It creates a dictionary bookRatings that contains the ratings of all books for different features.

```
foreach (var rating in allRatings)
{
    var bookId = rating.BookId;

    if (!bookRatings.ContainsKey(bookId))
    {
        bookRatings[bookId] = new Dictionary<string, double>();
    }

    var feature = $"Feature{rating.CategoryId}";
    bookRatings[bookId][feature] = rating.RatingValue;
}
```

The GetUserVector method takes in a list of Rating objects userRatings and a dictionary bookRatings that contains the ratings of all books for different features. It calculates the user vector, which is a dictionary that contains the user's ratings for different features.

```
foreach (var rating in userRatings)
{
    var bookId = rating.BookId;

    if (bookRatings.ContainsKey(bookId))
    {
        foreach (var feature in bookRatings[bookId])
        {
            if (userVector.ContainsKey(feature.Key))
            {
                userVector[feature.Key] += feature.Value * rating.RatingValue;
            }
            else
            {
                userVector[feature.Key] = feature.Value * rating.RatingValue;
            }
        }
    }
}
```

The CalculatePearsonSimilarities method calculates the Pearson similarity between a user vector and all book vectors. It takes in two parameters: a dictionary userVector that contains the user's ratings for different features, and a dictionary bookVectors that contains the ratings of all books for different features.
It initializes an empty pearsonSimilarities dictionary that will store the calculated Pearson similarity for each book.

Then, for each book in bookVectors, it calculates the intersection of features between the user and the book using the Intersect method. If there are common

features between the user and the book, it calculates the Pearson similarity using the formula (userValue - userMean) * (bookValue - bookMean) / (sqrt(userDenominator) * sqrt(bookDenominator)), where userValue and bookValue are the user's and book's ratings for a common feature, userMean and bookMean are the mean ratings of the user and book, and userDenominator and bookDenominator are the denominators in the Pearson similarity formula. If there are no common features between the user and the book, the Pearson similarity is set to 0 and returns the pearsonSimilarities dictionary containing the calculated Pearson similarity for each book.

```
if (commonFeatures.Any())
   {
      var userMean = userVector.Values.Average();
      var bookMean = bookVector.Values.Average();
      var numerator = 0.0;
      var userDenominator = 0.0;
      var bookDenominator = 0.0;

      foreach (var feature in commonFeatures)
      {
         var userValue = userVector[feature];
         var bookValue = bookVector[feature];

         numerator += (userValue - userMean) * (bookValue - bookMean);
         userDenominator += Math.Pow(userValue - userMean, 2);
         bookDenominator += Math.Pow(bookValue - bookMean, 2);
      }

      var denominator = Math.Sqrt(userDenominator) * Math.Sqrt(bookDenominator);

      // Handle cases where denominator is zero
      if (denominator == 0)
      {
         pearsonSimilarities[bookId] = 0;
```

```
        }
        else
        {
            var similarity = numerator / denominator;
            pearsonSimilarities[bookId] = similarity;
        }
    }
    else
    {
        pearsonSimilarities[bookId] = 0;
    }
```

The GetBookVectors method takes in a list of Rating objects allRatings and a dictionary bookRatings that contains the ratings of all books for different features. It creates a dictionary bookVectors that contains the book vectors for all books.

```
foreach (var rating in allRatings)
    {
        if (!bookVectors.ContainsKey(rating.BookId) &&
bookRatings.ContainsKey(rating.BookId))
        {
            bookVectors[rating.BookId] = new Dictionary<string, double>();
        }

        if (bookVectors.ContainsKey(rating.BookId))
        {
            var feature = $"Feature{rating.Id}";
            bookVectors[rating.BookId][feature] = rating.RatingValue;

            foreach (var otherRating in bookRatings[rating.BookId])
            {
                if (otherRating.Key != feature)
                {
                    var otherFeature = otherRating.Key;
```

```
                var bookId = rating.BookId;
                if (!bookVectors.ContainsKey(bookId))
                {
                    bookVectors[bookId] = new Dictionary<string, double>();
                }
                bookVectors[bookId][otherFeature] = otherRating.Value *
rating.RatingValue;
            }
        }
    }
}
```

Cosine Similarity

The implementation of cosine similarity in this recommendation system is somewhat similar to Pearson correlation, but it uses a different approach to calculate the similarities. Pearson correlation measures the linear correlation between two vectors, whereas cosine similarity measures the cosine of the angle between two vectors. In other words, Pearson correlation measures the similarity of the direction and strength of the relationship between two vectors, while cosine similarity measures the similarity of the direction between two vectors. Both methods are commonly used in recommendation systems to calculate similarities between user and item vectors, and each method has its strengths and weaknesses.

The CalculateCosineSimilarities method is responsible for computing the cosine similarity between the user vector and the book vectors. Unlike the CalculatePearsonSimilarities method, which calculates the Pearson correlation coefficient to measure the similarity between the user vector and book vectors, this method uses the cosine similarity measure.

```
private Dictionary<string, double> CalculateCosineSimilarities(Dictionary<string,
double> userVector, Dictionary<string, Dictionary<string, double>> bookVectors)
{
    var cosineSimilarities = new Dictionary<string, double>();
```

```
    foreach (var bookId in bookVectors.Keys)
    {
        var dotProduct = 0.0;
        var bookVector = bookVectors[bookId];

        // Calculate dot product
        foreach (var feature in userVector.Keys)
        {
            if (bookVector.ContainsKey(feature))
            {
                dotProduct += userVector[feature] * bookVector[feature];
            }
        }
        // Calculate magnitudes
        var userMagnitude = Math.Sqrt(userVector.Values.Sum(v => v * v));
        var bookMagnitude = Math.Sqrt(bookVector.Values.Sum(v => v * v));

        // Calculate cosine similarity
        var cosineSimilarity = dotProduct / (userMagnitude * bookMagnitude);

        cosineSimilarities[bookId] = cosineSimilarity;
    }
    return cosineSimilarities;
}
```

Knuth-Morris-Pratt (KMP)

The KMP class is a utility class for performing string matching using the Knuth-Morris-Pratt algorithm. It contains three methods: Search, KMPSearch, and ComputeLPSArray.

The Search method takes a string pattern and an NPDbContext object as input and returns a list of books that match the given pattern in their title. The method first

retrieves all books from the database using the provided context and then iterates through each book's title to check for a match using the KMPSearch method. If a match is found, the book is added to a list of matching books, which is returned at the end of the method.

```
public static List<Book> Search(string pattern, NPDbContext db)
{
    List<Book> books = db.Books.ToList();
    List<Book> matchingBooks = new List<Book>();
    foreach (Book book in books)
    {
        if (KMPSearch(book.Title.ToLower(), pattern.ToLower()))
        {
            matchingBooks.Add(book);
        }
    }
    return matchingBooks;
}
```

The KMPSearch method is a private method that takes two strings, text and pattern, as input and returns a Boolean value indicating whether the pattern appears in the text. The method first computes the longest prefix-suffix array (LPS) for the pattern using the ComputeLPSArray method. Then, it iterates through each character of the text and pattern to find a match using the LPS array. If a match is found, it increments both the text and pattern indices, and if the pattern index reaches the end, the method returns true. If a mismatch occurs, the pattern index is updated based on the LPS array to resume the search from the last known prefix-suffix match.

```
private static bool KMPSearch(string text, string pattern)
{
    int n = text.Length;
    int m = pattern.Length;
    int[] lps = ComputeLPSArray(pattern, m);
    int i = 0; // index for text[]
```

```
    int j = 0; // index for pattern[]
    while (i < n)
    {
        if (pattern[j] == text[i])
        {
            j++;
            i++;
        }
        if (j == m)
        {
            return true;
        }
        else if (i < n && pattern[j] != text[i])
        {
            if (j != 0)
            {
                j = lps[j - 1];
            }
            else
            {
                i++;
            }
        }
    }
    return false;
}
```

The ComputeLPSArray method is a private method that takes a string pattern and its length, m, as input and returns an array of integers representing the longest prefix that is also a suffix for each index of the pattern string.

The method initializes an array of zeros with a length equal to the pattern string length and iterates through the pattern string to compute the LPS array using two indices: i for the current index and len for the length of the longest prefix-suffix. If the character at index i matches the character at index len, then the length of the longest

prefix-suffix is incremented, and the value of the LPS array at index i is set to len. Otherwise, len is updated based on the previous index's LPS value, and the process is repeated until the end of the pattern string is reached.

```
private static int[] ComputeLPSArray(string pattern, int m)
{
    int[] lps = new int[m];
    int len = 0;
    int i = 1;
    lps[0] = 0;

    while (i < m)
        {
            if (pattern[i] == pattern[len])
            {
                len++;
                lps[i] = len;
                i++;
            }
            else
            {
                if (len != 0)
                {
                    len = lps[len - 1];
                }
                else
                {
                    lps[i] = 0;
                    i++;
                }
            }
        }
        return lps;
    }
```

5. Testing

Acceptance Tests

User authentication was a crucial requirement for this software application that involved sensitive information. The software allowed users to authenticate themselves before accessing specific parts of the system, ensuring that only authorized users were granted access to the application's features, thereby protecting sensitive data from unauthorized access. Acceptance testing for user authentication was done with valid and invalid user credentials to ensure that the authentication process worked as intended.

Positive testing scenarios were tested with valid user credentials, while negative testing scenarios were tested with invalid or incorrect user credentials to ensure that the software can handle and reject invalid inputs correctly. The user authentication testing covered tests for different types of user inputs, including tests for different password lengths, special characters, and other constraints to ensure that the software can handle different types of user inputs. Testing was done in a secure environment, and the test results were documented for future reference.

Data input validation was another crucial requirement for software applications that handle sensitive data. Acceptance testing for data input validation involved testing the software with various types of data inputs, such as strings, integers, and special characters, to ensure that the input validation works correctly.

Data processing and output were also tested to ensure that the software could process data inputs correctly and provide valid outputs based on the input data. Acceptance testing for data processing and output covered all possible input scenarios, including positive and negative testing scenarios, and also included testing for error handling and recovery. The test results were documented for future reference.

Acceptance Tests for Recommenders

Developing a recommender system for the Next Page web app presents several challenges, particularly in creating unit tests for the algorithms used, such as SVD, Pearson correlation, and cosine similarity. These challenges stem from the complexity of these algorithms, which involve mathematical operations that may not be easy to replicate in a testing environment. For instance, SVD requires matrix factorization, which may not be straightforward to implement in a unit test. Similarly, Pearson correlation and cosine similarity rely on complex similarity measures between users or items, which can be difficult to validate through unit testing.

Another challenge in creating unit tests for recommender systems is the variability of the data used to train the models. Since recommender systems rely on user behavior data, the data can be highly subjective and inconsistent, leading to a range of different ratings for the same item. This variability makes it challenging to establish a set of consistent test cases that accurately reflect the performance of the recommender system.

In addition, recommender systems tend to be highly personalized, meaning that the results may vary depending on the user's preferences and behavior. This variability can make it difficult to create test cases that are relevant to all users and ensure that the system performs well for all types of users. Therefore, the test cases need to be carefully designed to cover a range of scenarios that are representative of the user population, ensuring that the results are reliable and accurate.

Despite the challenges in creating unit tests for recommender systems, acceptance testing has been performed to ensure that the recommender system in the Next Page web app is working correctly. Acceptance testing involves testing the system as a whole to ensure that it meets the specified requirements and performs as expected. In this case, a group of beta users tested the system and provided feedback on the performance of the recommender system. This feedback was used to refine the algorithm and improve the accuracy of the recommendations.

Beta Users Tests

Procedure:

The acceptance testing was performed by a group of beta users who were given access to the Next Page web app. The beta users were selected based on their interest in books and their willingness to provide feedback on the app's performance. The beta users were asked to use the app for a period of two weeks and to provide feedback on the following areas:

1. Accuracy of Pearson recommendations
2. Accuracy of Cosine
3. Ease of use
4. Clarity of the user interface
5. Overall user experience

The beta users were also asked to provide suggestions for improvements and to identify any issues they encountered while using the app.

Results:

The acceptance testing results showed that the Next Page web app performed well in all areas evaluated. The beta users reported that the app provided accurate book recommendations that were relevant to their interests. The app was also easy to use, and the user interface was clear and intuitive. The overall user experience was positive, with the beta users reporting that they enjoyed using the app and found it helpful in discovering new books.

In terms of improvements, the beta users suggested that the app could benefit from additional features such as the ability to filter recommendations by author. Some users also reported that they encountered issues with the app's search function, which occasionally returned irrelevant results.

Implications:

The acceptance testing results provide valuable insights into the app's performance and user experience. The positive feedback from the beta users confirms that the app's recommendation algorithm is effective and that the user interface is intuitive and easy to use. The suggestions for improvements provide a roadmap for future

development and demonstrate that there is room for growth and expansion of the app's features.

The issues reported by the beta users, such as the search function's performance, highlight areas for improvement and optimization. These issues will need to be addressed in future updates to the app to ensure that it continues to perform well and meets the expectations of its users.

Unit Tests

Unit Testing for Pearson Correlation Class
Pearson Correlation is a statistical technique used to measure the strength and direction of the linear relationship between two variables. In the context of recommendation systems, Pearson Correlation is used to find similarities between users or items based on their ratings. The xUnit testing framework was used to create unit test for the CreateBookRatingsDictionary.

The CreateBookRatingsDictionary method is responsible for creating a dictionary of book IDs and their corresponding feature ratings. To test this method, we will create a test case with a known set of ratings and a dictionary of expected book ratings. We will then call the CreateBookRatingsDictionary method with the known ratings and compare the returned book ratings with the expected book ratings.

Acceptance Testing for KMP

The acceptance testing process for the KMP algorithm was performed for the next page web app. The procedure involved creating test cases to verify the algorithm's correctness and ensure its suitability for use in the app. These test cases were designed to test different scenarios and edge cases, including input strings of varying lengths and patterns with varying complexity.

The results of the acceptance testing showed that the KMP algorithm met the acceptance criteria and was deemed suitable for use in the next page web app. All

test cases passed successfully, indicating that the algorithm was able to correctly identify occurrences of patterns in the input string.

The acceptance testing process also highlighted the algorithm's strengths, such as its ability to handle input strings of large sizes and complex patterns efficiently. It also identified potential areas for improvement, such as the need for additional error handling for invalid input strings.

Limitations

One of the main limitations of KMP algorithm is that it requires preprocessing of the pattern string before searching can begin. This preprocessing can take a significant amount of time and memory for large pattern strings, which can impact the algorithm's performance. Additionally, the algorithm only works for exact pattern matching and cannot handle approximate matching or fuzzy search. Another limitation is that KMP assumes the text string is static and does not account for changes or updates to the text, which means that the algorithm would need to be re-run if the text changes. Finally, KMP is not suitable for searching in certain types of data structures, such as trees or graphs, which require different search algorithms.

Unit Testing for KMP

I have implemented the Knuth-Morris-Pratt (KMP) algorithm to add search functionality to a web application based on the book's name or author entered in a search bar. After implementing the algorithm, I conducted unit testing using the xUnit framework to ensure its functionality and correctness.

During the testing process, I created several test cases to cover a range of scenarios, such as testing the algorithm's ability to find matches when the pattern is at the beginning, middle, and end of the text, and when it is not present at all. Additionally, I tested the algorithm's performance with larger inputs to ensure that it is scalable and efficient.

The test results showed that the KMP algorithm was able to find all occurrences of the pattern in the text for all the test cases, and it was able to handle larger inputs without any significant performance issues. The tests also helped me identify edge cases where the algorithm did not work correctly, allowing me to make the necessary adjustments and improve its accuracy.
The below tests are unit tests for the Knuth-Morris-Pratt (KMP) algorithm. The first test, "KMPSearch_ReturnsTrue_WhenPatternExistsInText," verifies that the KMP search function returns true when the given pattern exists in the text. The test sets up the text and pattern to be searched, calls the KMP search function, and then uses the Assert function to verify that the result is true.

The second test, "KMPSearch_ReturnsFalse_WhenPatternDoesNotExistInText," verifies that the KMP search function returns false when the given pattern does not exist in the text. The test sets up the text and pattern to be searched, calls the KMP search function, and then uses the Assert function to verify that the result is false.
The third test, "ComputeLPSArray_ReturnsCorrectArray," verifies that the KMP function to compute the Longest Prefix Suffix (LPS) array returns the correct array. The test sets up a pattern for which the LPS array is known, calls the KMP function to compute the LPS array, and then uses the Assert function to verify that the computed LPS array matches the expected LPS array.

The procedure for writing the tests involved setting up the input parameters, calling the KMP functions, and then using the Assert function to verify that the output was as expected. The results obtained from running these tests were all successful, indicating that the KMP algorithm is functioning correctly for the given input parameters.

```
[Fact]
public void KMPSearch_ReturnsTrue_WhenPatternExistsInText()
{
    // Arrange
    string text = "The quick brown fox jumps over the lazy dog";
    string pattern = "brown fox";
```

```
    // Act
    bool result = KMP.KMPSearch(text, pattern);

    // Assert
    Assert.True(result);
}

[Fact]
public void KMPSearch_ReturnsFalse_WhenPatternDoesNotExistInText()
{
    // Arrange
    string text = "The quick brown fox jumps over the lazy dog";
    string pattern = "pink elephant";

    // Act
    bool result = KMP.KMPSearch(text, pattern);

    // Assert
    Assert.False(result);
}

[Fact]
public void ComputeLPSArray_ReturnsCorrectArray()
{
    // Arrange
    string pattern = "ABABC";

    // Act
    int[] result = KMP.ComputeLPSArray(pattern, pattern.Length);

    // Assert
    Assert.Equal(new int[] { 0, 0, 1, 2, 0 }, result);
}
```

User Interface Testing

I focused my testing on the forms used to submit book ratings and add books to the reading list, the navigation menu used to browse books, and the overall CRUD functionality of the app.

To begin, I asked participants to rate several books and add them to their reading list using the forms. The usability test revealed that some participants had difficulty finding the forms on the book pages, which were located in a small section at the bottom of the page. To improve this, I added a prominent "Rate this book" and "Add to reading list" button near the top of the book pages, making it easier for users to access the forms. Additionally, some participants were confused by the book rating scale, which was based on a 1-5 star system. To address this issue, I added a tooltip that appears when users hover over each star, explaining what each star represents. Furthermore, a prominent "Submit rating" button was added below the rating scale to make it clear how to submit the rating.

Next, I tested the navigation menu used to browse books. Participants were asked to find books based on specific criteria such as genre or author. The usability test revealed that some participants were overwhelmed by the number of options in the navigation menu, which included multiple levels of submenus. To address this issue, I simplified the navigation menu by removing some of the submenus and grouping related options together. Additionally, participants were confused by the search functionality, which was located in a small search bar in the top right corner of the page. To improve this, I added a prominent search bar at the top of the navigation menu, making it easier for users to find and use the search functionality.

Finally, I asked participants to add, edit, and delete books from their reading list and view their list of books to test the overall CRUD functionality of the app. The usability test revealed that some participants had difficulty finding the editing and deleting functionality, which were located in a small dropdown menu on each book in the reading list. To address this issue, I added prominent "Edit" and "Delete" buttons below each book in the reading list, making it easier for users to access the functionality.

Remarks

I found unit testing to be a useful yet challenging process, therefore, only succeeded to implement it for certain methods that produced predictable output. The acceptance testing for user authentication and data input validation ensured that the software met the specified requirements and worked as intended. For recommender systems, unit testing for algorithms such as SVD, Pearson correlation, and cosine similarity is challenging due to the complexity of these algorithms and the variability of the data used to train the models. Acceptance testing, such as beta user testing, was performed to evaluate the performance of the recommender system in the Next Page web app. The results provided valuable insights into the app's performance and user experience, and the suggestions for improvements will be used to guide future development.

6. Results and Conclusion

Expectations vs Results

During the development of thi project, my initial plan was to implement both content-based and hybrid filtering techniques to provide recommendations for users. However, after thorough research and testing, it became clear that collaborative filtering was the most effective approach. Collaborative filtering allowed the system to suggest books to users based on the preferences of other users with similar tastes, which proved to be highly accurate and useful.

In terms of search functionality, I originally intended to incorporate geolocation as a factor in search results. This would have allowed users to find books based on their current location or the location of a specific bookstore. Unfortunately, due to time constraints, I did not manage to fully implement this feature. However, I believe that this could be a valuable addition to the project in the future.

One of the challenges I encountered during working on this project, was looking for a way to optimize the storage and process the large amounts of data required for the collaborative filtering algorithm. Through research and experimentation, I managed to find suitable data structure and storage methods to achieve a decent of performance.

I am pleased with the final result of the project. The recommendation and search functionality are the main focus of the application, and I believe they have been successfully implemented and provide a valuable resource for users. The user interface is fairly intuitive and user-friendly, allowing users to easily navigate the app and find books of interest.

In terms of future development, I would like to further adjust the collaborative filtering algorithm and continue the process of optimizing the data processing methods. I would also like to add the geolocation feature and potentially incorporate additional recommendation techniques, such as deep learning-based methods.

What Works and What Does Not

Throughout the development of the project, various issues with data preprocessing were encountered. The project aimed to implement content-based and hybrid filtering but eventually I decided that collaborative filtering was the best method to use. Nonetheless, the data required significant preprocessing to extract relevant features and ensure that the data was clean and usable. Despite best efforts, challenges were encountered during the data preprocessing stage, with some data being incomplete or missing. The challenges of the data preprocessing stage made it difficult to ensure that the data was processed correctly, especially since unit testing was difficult to implement for the Data Preprocessor class methods.

Despite these challenges, the Pearson Correlation, Cosine Similarity, and KMP algorithms used in the project have shown to be effective. These algorithms were used to analyze the data and extract relevant features that were helpful in recommending books to users. With the implementation of these algorithms, a robust recommendation engine was developed, which could make accurate recommendations based on user preferences.

While the project initially aimed to implement geolocation, it was not feasible given the resources and time available. Consequently, the scientific areas of the project, including the recommendation engine and search functionality, were the main focus. Focusing on these areas ensured that a robust and effective system was delivered, which met the needs of users.

Overall, the project was successful in achieving its goals, although there were some challenges encountered along the way. The difficulties of data preprocessing were overcome by using effective algorithms such as PC, CS, and KMP. In contrast, the lack of geolocation functionality was addressed by focusing on other scientific areas of the project, which led to the development of a robust and effective system.
In terms of areas that did not work, unit testing was difficult to implement, making it challenging to verify that data preprocessing was successful. Without adequate unit testing, it was challenging to determine if the preprocessing was successful or not. Additionally, the lack of geolocation functionality meant that the system was not as

comprehensive as it could have been, although it did not impact the core functionality of the system.

Looking forward, several areas could be improved to enhance the system. For instance, more extensive unit testing could be implemented to verify that the data preprocessing is successful. Also, there is the possibility of incorporating geolocation functionality into the system to offer users a more comprehensive experience. Nevertheless, the project achieved its goals and delivered an effective recommendation engine and search functionality.

In conclusion, data preprocessing was a challenging aspect of the project. However, by using effective algorithms such as PC, CS, and KMP, the challenges were overcome, leading to the development of a robust recommendation engine. Despite not being able to implement geolocation, focusing on other scientific areas of the project ensured that the system was successful in achieving its goals. Going forward, there is still

Outcomes

Throughout the development of the project, I gained several valuable insights into the field of recommendation systems and software engineering. One of the most significant things I learned was the importance of data preprocessing in developing effective recommendation systems. Preprocessing the data required extracting relevant features, cleaning the data to remove irrelevant or redundant information, and handling missing or incomplete data. While these tasks may seem simple at first glance, they can be challenging and time-consuming, particularly when dealing with large datasets. By working through these challenges, I developed a better understanding of the importance of data preprocessing and how it can impact the effectiveness of a recommendation system.

Another important lesson I learned was the value of collaboration in software development. Although I had initially intended to implement content-based and hybrid filtering, I found that collaborative filtering worked best. By working collaboratively with other developers and researchers, I was able to gain valuable

insights into the strengths and weaknesses of different recommendation algorithms and how they could be applied to real-world problems. Additionally, collaborating with others allowed me to learn more about best practices in software development, including version control, testing, and debugging.

One of the most challenging aspects of the project was unit testing, which I found to be difficult to implement effectively. While unit testing is an essential part of software development, it can be challenging to develop tests that accurately reflect the behavior of the system under different conditions. In the context of a recommendation system, this was particularly challenging since it required testing the system's ability to recommend books to users based on their preferences. Despite these challenges, I learned the importance of testing and how it can help identify and prevent bugs in the system.

I faced issues with implementing geolocation functionality as well. Although I had initially intended to add this functionality, I found that it was not feasible given the time and resources available. Instead, I focused on developing the scientific areas of the project, including the recommendation engine and search functionality. While it would have been ideal to implement geolocation, I learned the importance of prioritizing tasks and focusing on the most critical components of the project. By doing so, I was able to develop a system that met the needs of users and achieved its goals effectively.
In addition to the technical skills I gained, I also developed several soft skills that are valuable in software development and other fields. For example, I learned the importance of communication, particularly when working collaboratively with others. Effective communication helped ensure that everyone was on the same page, and that tasks were assigned and completed effectively. Additionally, I developed my problem-solving skills as I worked through the various challenges that arose during the project. By approaching these challenges methodically and creatively, I was able to develop effective solutions that helped move the project forward.

Another challenge I encountered during the development of the search functionality was identifying and extracting relevant keywords from the dataset. This required a thorough understanding of the data and the ability to identify patterns and trends in

the text. Once the keywords were identified, I then had to develop an algorithm that could search through the dataset and retrieve relevant information based on those keywords.

Optimizing the search algorithm for efficiency was also a part of making sure project works as I expected it to. With large datasets, searching through every document can be incredibly time-consuming and resource-intensive. To overcome this, I had to implement techniques such as indexing and caching to improve the speed and efficiency of the search algorithm.

Conclusion

While working on the project, I encountered several challenges, particularly with data preprocessing and incorporating geolocation functionality. However, I persevered and found effective solutions that allowed me to develop a robust recommendation engine and search functionality.

I learned a great deal throughout this project, including the importance of feature extraction in recommendation systems, the complexities of data preprocessing, and the challenges of implementing geolocation functionality. Additionally, I gained experience in programming and algorithm development, which will be valuable in future projects.
Overall, this project was a valuable learning experience and a demonstration of my passion for technology and innovation. I look forward to continuing to explore new technologies and developing innovative solutions that can make a positive impact on our world.

7. References

1. ".NET Core in Action" by Dustin Metzgar and Jim Wooley
2. Manning, C. D., Raghavan, P., & Schütze, H. (2008). Introduction to information retrieval. Cambridge University Press. (covers Pearson and Cosine similarity measures)
3. "KMP Algorithm for Pattern Searching." GeeksforGeeks. https://www.geeksforgeeks.org/kmp-algorithm-for-pattern-searching/ (accessed April 21, 2023).
4. "Cosine Similarity for Vector Space Models." Stanford University. https://nlp.stanford.edu/IR-book/html/htmledition/cosine-similarity-1.html (accessed April 21, 2023).
5. "An Introduction to Pearson's Correlation Coefficient." Statistic How To. https://www.statisticshowto.com/probability-and-statistics/correlation-coefficient-formula/pearsons-correlation-coefficient-definition-calculate/ (accessed April 21, 2023).
6. Gower, J. C. (1971). A general coefficient of similarity and some of its properties. Biometrics, 27(4), 857-871. (discusses the properties of the Pearson correlation coefficient)
7. "KMP Algorithm." TutorialsPoint. https://www.tutorialspoint.com/KMP-Algorithm-for-Pattern-Searching (accessed April 21, 2023).
8. "Understanding Cosine Similarity." Towards Data Science. https://towardsdatascience.com/understanding-cosine-similarity-9c5ebbc272f7 (accessed April 21, 2023).
9. "How to Use Pearson Correlation Coefficient for Feature Selection." Machine Learning Mastery. https://machinelearningmastery.com/feature-selection-with-real-and-categorical-data/ (accessed April 21, 2023).
10. "Knuth-Morris-Pratt (KMP) Algorithm." Programiz. https://www.programiz.com/dsa/knuth-morris-pratt-algorithm (accessed April 21, 2023).
11. "Cosine Similarity: Understanding the Math and How it Works (with python codes)." Analytics Vidhya. https://www.analyticsvidhya.com/blog/2022/02/cosine-similarity-

understanding-the-math-and-how-it-works-with-python-codes/ (accessed April 21, 2023).
12. Jolliffe, I. T. (2002). Principal component analysis. Wiley Online Library. (discusses SVD algorithm and its applications)
13. Golub, G. H., & Van Loan, C. F. (2012). Matrix computations (Vol. 3). JHU Press. (discusses SVD algorithm and its properties)
14. "Professional C# 7 and .NET Core 2.0" by Christian Nagel, et al.
15. Cichocki, A., & Phan, A. H. (2009). Fast local algorithms for large scale nonnegative matrix and tensor factorizations. IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences, 92(3), 708-721. (discusses SVD-based matrix factorization algorithms)
16. Knuth, D. E. (1997). The art of computer programming, volume 1: Fundamental algorithms (3rd ed.). Addison-Wesley Professional. (discusses Knuth-Morris-Pratt algorithm)
17. Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). Introduction to algorithms (3rd ed.). MIT Press. (discusses string matching algorithms including Knuth-Morris-Pratt algorithm)
18. "Mastering .NET Machine Learning" by Jamie Dixon and Wei Liu
19. "Unit Testing in .NET Core and ASP.NET Core" by Andrew Lock (https://andrewlock.net/tag/unit-testing/)