From Books to Predictions: Building a Personalized Book Recommendation Algorithm

Coralie Ostertag (300174530)

Tamara Micic (300163312)

Winter 2025

University of Ottawa

Abstract

In an era of information overload, recommender systems play a crucial role in connecting users to content that aligns with their interests. This project focuses on building a personalized book recommendation engine using the Book-Crossing dataset, enhanced with metadata retrieved from the ISBN-DB API. To address issues such as sparse user-item interactions and limited item descriptions, we enriched book profiles with subjects, synopses, and page counts, enabling more nuanced content-based filtering. We implemented and compared several recommendation strategies, including TF-IDF-based content filtering, user-based collaborative filtering with cosine similarity, SVD-based latent factor models, and KNN approaches. To further improve our model, we used LightFM, incorporating both explicit and implicit feedback. In addition, we addressed the cold-start problem to make informed recommendations for users or items with limited historical data. Overall, this study aims to explore the effectiveness of combining enhanced item metadata with multiple recommendation techniques to improve personalization and robustness in book recommender systems.

1. Introduction

1.1 Background

Many digital platforms today depend heavily on recommender systems to connect users with relevant content, guiding users toward content that aligns with their preferences and interests. Whether it's suggesting songs on Spotify, movies on Netflix, or products on Amazon, these systems help users discover relevant items in environments where the volume of choices can be overwhelming.

By analyzing user behavior and item characteristics, recommendation algorithms tailor content to individual preferences**,** improving user satisfaction and keeping them engaged. In the context of books, such systems can promote reading diversity, uncover hidden gems, and match readers with titles they might not have found otherwise.

1.2 Motivation

The Book-Crossing dataset provides a great opportunity to study recommendation systems using real-world data. It contains thousands of user-submitted book ratings, along with book metadata such as title, author, and ISBN. However, as is common with public datasets, it suffers from issues such as missing data, sparse interactions, and limited descriptive information for items.

To overcome these challenges and enrich the dataset, this project integrates additional features retrieved from the ISBNDB API, such as book subjects, number of pages and synopses. These enhancements help create a more nuanced representation of each item, which is particularly useful for content-based and hybrid recommendation approaches.

1.3 Objectives

This project aims to design, implement, and evaluate multiple recommendation approaches using the enriched Book-Crossing dataset. The specific objectives are:

- To implement a content-based filtering system using textual features such as subjects and descriptions.
- To develop a collaborative filtering system based on user-item interactions, employing both matrix factorization (SVD) and implicit feedback models.
- To build a hybrid recommender system that combines content and collaborative signals using the something like the LightFM library.
- To assess and compare the performance of these models using appropriate evaluation metrics and then possibly combining them to build a better system.

1.4 Recommender Systems

1.4.1 What is a Recommender System?

A recommender system is a type of software tool designed to suggest items to users based on their preferences, behaviors, or demographic characteristics. These systems are prevalent in various digital platforms, from e-commerce websites like Amazon to entertainment platforms like Netflix or Spotify. Their primary function is to help users discover relevant content, thereby enhancing user experience and engagement.

Recommender systems typically analyze historical user behavior (such as ratings, clicks, purchases, or views) and use this data to predict what new items a user might like. These predictions are often based on collaborative filtering (which suggests items based on the preferences of similar users), content-based filtering (which recommends items based on their characteristics, such as genres or descriptions), or a hybrid approach that combines both methods.

In the context of books, a recommender system helps readers discover titles they may not have found otherwise, thus promoting diversity in reading choices. By analyzing a user's reading history and the features of the books they have enjoyed, the system can recommend new books that match the user's interests or expand their reading horizon.

1.4.2 What Makes for a Good Recommender System?

A good recommender system provides personalized, relevant, and diverse recommendations that not only meet a user's current interests but also help them discover some new content as well. Below are the essential qualities that make a recommender system effective, focusing on both technical performance and user experience.

- **Accuracy:** A primary goal of any recommender system is to suggest items that the user will likely enjoy or find useful. This is measured by how well the system can predict user preferences, often using metrics such as Root Mean Squared Error (RMSE) for rating predictions or Precision/Recall for classification-based tasks. High accuracy is crucial because it directly influences user satisfaction, meaning users are more likely to engage with the system if they trust the recommendations.
- **Novelty:** Novelty in recommendations is essential for maintaining user interest and engagement. Users often enjoy discovering new content that is different from their usual preferences but still relevant. For example, a book recommender system might suggest a genre or author the user has not explored before, offering something fresh but still in line with their general interests. Novelty prevents the system from becoming repetitive by encouraging the user to try books they may not have picked up otherwise.

  In our system, novelty plays a critical role in helping users explore new genres, authors, or themes while maintaining a personalized touch. By using demographic information like age and location, we can introduce educated guesses that suggest books outside a user's typical reading pattern, thus adding an element of surprise and exploration. For instance, a user who primarily reads historical fiction might receive a recommendation for a well-rated mystery novel, which could introduce a new reading interest without straying too far from their tastes

- **Diversity:** A good recommender system offers diverse suggestions. Rather than constantly recommending similar types of content, the system should span across multiple categories, helping users discover a broader range of items. This is particularly valuable in settings like music or books, where users might appreciate suggestions from genres they don't actively seek out but still enjoy.

  Diversity helps keep the recommendations interesting and ensures that users are exposed to a variety of content. For example, our book recommendation system might suggest a mix of genres—like fiction, non-fiction, and graphic novels—rather than recommending books that are all within the same genre, providing a richer user experience.
- **Personalization:** One of the most crucial aspects of a successful recommender system is its ability to be personalized. A system that tailors its suggestions to each user's unique tastes and preferences is more likely to engage them over time. Personalization involves analyzing a user's past behavior and learning their preferences to make increasingly accurate predictions. As the system collects more data about the user, it should be able to refine its recommendations, making them more relevant and appealing.
- **Serendipity:** Serendipity refers to the element of surprise in recommendations—offering suggestions that the user may not have expected but still enjoy. A good system should balance between recommending familiar items and introducing unexpected content that enhances the user's discovery process. This quality enhances user satisfaction by delivering recommendations that feel both intuitive and pleasantly surprising.
- **Scalability and Efficiency:** A recommender system must be able to handle large datasets as the number of users and items increases. Scalability ensures that the system can continue to perform well and provide relevant recommendations even as it processes vast amounts of data. Efficiency is also important because recommendations should be delivered in real-time or near-real-time, offering users a seamless experience without delays.
- **Transparency and Explainability:** Users are more likely to trust a recommender system if they understand why a particular item is being recommended. Transparency involves explaining the reasoning behind recommendations, which can help users feel more confident in the system's suggestions. For example, a book might be recommended because it shares similar themes with other books the user has enjoyed. This kind of explanation improves user trust and engagement.
- **Cold-Start Problem Handling:** The cold-start problem is a common challenge in recommender systems, particularly when dealing with new users or items that lack sufficient data. A good system should be able to overcome this problem by using demographic data (such as age, location, or preferences) to make educated recommendations for new users. This not only helps new users get started but also introduces an element of novelty by suggesting items that are popular in their demographic group. Note: In our project, we implemented a cold-start solution that suggests books based on the user's age and location. More on this later.
- **User Engagement and Satisfaction:** Finally, a good recommender system must keep users engaged and satisfied over time. This is achieved by offering relevant, novel, and diverse recommendations that meet the evolving needs and preferences of the user. Engagement can also be enhanced by continuously updating the system's learning based on user interactions, ensuring that the recommendations remain fresh and interesting.

In summary, a good recommender system is not only accurate but also diverse, personalized, and capable of offering novel suggestions that encourage discovery. By balancing these factors, the system can create

an enjoyable and dynamic user experience that keeps users coming back for more. Novelty, in particular, is a vital element that ensures the system doesn't become predictable or repetitive, allowing users to discover new content in a way that feels both relevant and exciting.

1.4.3 Types of Recommender Systems

Recommender systems come in various forms, each using different methodologies to generate suggestions. The three primary types are content-based filtering, collaborative filtering, and hybrid systems, which combine both approaches. These methods vary in how they generate recommendations, relying on different types of data and algorithms.

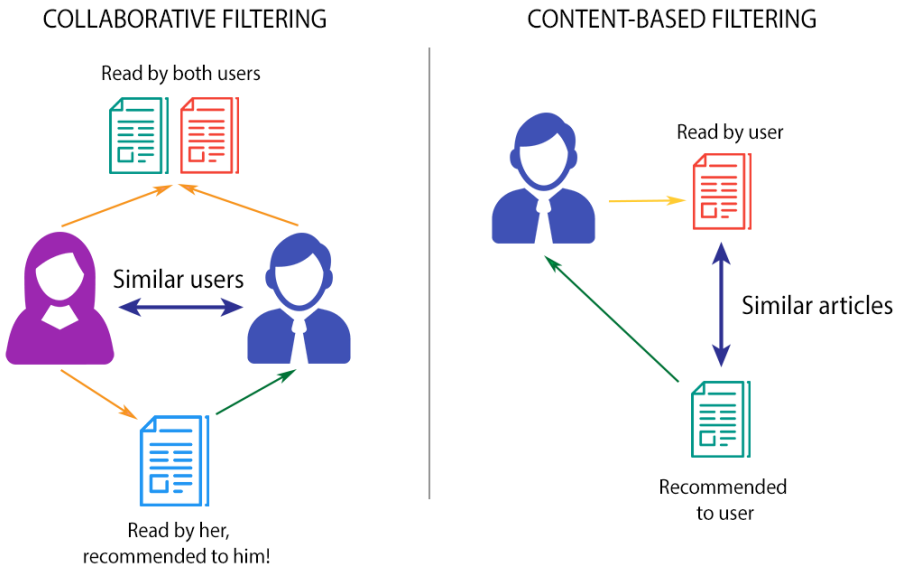

Figure 1.1: A Visual Way of Explaining Content-Based vs. Collaborative Filtering from https://medium.com/data-science/brief-on-recommender-systems-b86a1068a4dd

1.4.3.1 Content-Based Filtering

Content-based filtering recommends items based on the characteristics of the items themselves and the preferences demonstrated by the user. This approach relies heavily on item attributes—such as genre, author, keywords, or descriptions—and uses these features to find similarities between items. The idea is that if a user has shown a preference for certain characteristics (e.g., books by a particular author or in a specific genre), the system will recommend items with similar attributes.

For example, in a book recommendation system, content-based filtering might suggest books with similar themes, writing styles, or topics that align with the user's past reading behavior. This is particularly useful when there is abundant metadata about the items, such as detailed descriptions or structured features.

Advantages:

- Transparency: Users can easily understand why an item was recommended (e.g., "This book is similar to other books you've read").
- No Cold-Start Problem for Items: New items can be recommended if their metadata is available, even if they don't have any user ratings.

Disadvantages:

- Limited Discovery: The system may suggest items that are too similar to what the user has already seen, limiting the novelty and diversity of recommendations.
- Relies on Metadata: The effectiveness of content-based filtering is dependent on the availability and quality of item metadata. Inaccurate or sparse metadata can reduce the system's performance.

1.4.3.2 Collaborative Filtering

Collaborative filtering is one of the most widely used methods in recommender systems. It makes recommendations based on the behavior and preferences of similar users, rather than relying on item features. Collaborative filtering can be divided into two main types: user-based and item-based.

1. User-Based Collaborative Filtering: This method suggests items by finding users with similar preferences and recommending items that those similar users have liked. For instance, if User A and User B both rated books highly in the same genre, User A might be recommended books that User B has enjoyed but User A hasn't yet read.
2. Item-Based Collaborative Filtering: This method works by finding items that are similar to those the user has already rated or interacted with. For example, if User A likes Book X, the system might recommend other books that are similar to Book X based on the ratings of users who have liked Book X.

Advantages:

- No Need for Item Metadata: Collaborative filtering only requires user-item interaction data (e.g., ratings, clicks) and does not rely on item attributes. This is especially useful when metadata is sparse or unavailable.
- Captures Implicit Preferences: It can reveal hidden relationships between users and items that may not be immediately apparent through content attributes.

Disadvantages:

- Cold-Start Problem for Users and Items: New users or items without any ratings make it difficult for the system to generate accurate recommendations.
- Data Sparsity: When user-item interaction data is sparse (e.g., not many users have rated the same items), the system may struggle to make meaningful connections and generate good recommendations.

1.4.3.3 Hybrid Recommender Systems

Hybrid recommender systems combine the strengths of both content-based filtering and collaborative filtering to overcome the limitations of each method. These systems use a variety of techniques to provide more accurate and diverse recommendations, often combining the insights from different types of data sources.

There are several ways to combine these methods:

- Weighted Hybrid: This approach assigns weights to different recommendation algorithms and combines their outputs into a final recommendation list.

- Switching Hybrid: In this approach, the system selects one method based on the context, such as using content-based filtering when there is limited user history and collaborative filtering when sufficient data is available.
- Mixed Hybrid: This combines recommendations from different algorithms and presents them together to the user, ensuring diversity in the suggestions.

Advantages:

- Better Performance: By combining methods, hybrid systems can provide more accurate and relevant recommendations than either approach alone.
- Overcomes Limitations: For example, while collaborative filtering struggles with the cold-start problem for new users or items, content-based filtering can still generate recommendations based on available metadata.

Disadvantages:

- Complexity: Hybrid systems are more complex to implement and maintain. They require balancing multiple algorithms and integrating different data sources.
- Computational Cost: Combining multiple methods can increase the computational resources required for processing and providing real-time recommendations.

Each of the three main types of recommender systems—content-based, collaborative filtering, and hybrid systems—offers unique strengths and challenges. Content-based filtering excels in transparency and handling new items, while collaborative filtering thrives in capturing patterns from user behavior but struggles with data sparsity and cold-start issues. Hybrid systems combine the best of both worlds, overcoming the limitations of individual approaches but at the cost of added complexity and computational demand.

By understanding the characteristics of these methods, we can tailor our approach to designing and implementing a more effective recommender system for any given dataset, such as the Book-Crossing dataset in this project, to meet the specific needs of the user base and context.

2. The Data

Finding a suitable dataset turned out to be more difficult than we initially anticipated. We first set our sights on Spotify's Million Playlist Dataset. Given our shared interest in music, we thought building a recommender system to suggest songs would be a perfect project. Music is something most people enjoy, but preferences vary—some like rock, others prefer pop, and the same goes for albums and artists. There was a coding challenge in 2018 tied to the dataset, but after some investigation, it appeared the dataset had been since taken down, and the only way to access it was through contacts at Spotify. We reached out to every contact they provided, along with others we found, but unfortunately, we couldn't find a way to gain access. This meant we had to adjust our plans. After doing more research, we discovered the Book-Crossing dataset. It felt like a solid alternative since books, like songs, come in many genres, and we could still create something engaging that aligned with our original idea.

2.1 The Book-Crossing Dataset

BookCrossing is a global community where book lovers share, rate, and exchange books. Members of the platform can release their books into the world for others to find or pass them along to fellow BookCrossing members. The BookCrossing platform has all this information on users, books, and ratings, reflecting the exchange of books within the community. With over 1.9 million members, the platform connects readers worldwide, encouraging the joy of sharing books and discovering new ones.

2.1.1 An Overview of the Book-Crossing Dataset

The Book-Crossing dataset is a well-known dataset in the recommendation systems and data science community. It was collected by Cai-Nicolas Ziegler in 2004, with the permission of Ron Hornbaker, CTO of Humankind Systems (the BookCrossing's parent company). It is often used for research in collaborative filtering and recommendation algorithms. The dataset contains four main files:

- `users_info.csv`: This file holds demographic information about users, such as User-ID, Location, and Age, which can be useful for analyzing patterns in user preferences based on these characteristics.
- `books_info.csv`: Here, you'll find details about the books, including the ISBN, Book-Title, Book-Author, Year of Publication, Publisher, and a URL for the book cover. This information is essential for understanding the diversity and scope of books in the dataset.
- `book_ratings.csv`: This file contains the ratings users have given to books, with User-ID, ISBN, and the Book-Rating (ranging from 0 to 10). A rating of 0 indicates an implicit interaction, meaning the user either didn't rate or the rating wasn't recorded.
- `book_history.csv`: Here, you'll find a list of books that each user accessed. This can be useful to look at the books that users accessed, but didn't rate.

The dataset provides a great starting point for building recommendation systems, especially for book-related recommendations. We were able to access it on GitHub and used it as a starting point for our project.

2.1.2 Cleaning the Data

The original Book-Crossing dataset is provided in .dat files with tab-separated values. However, we encountered several formatting issues that required cleaning before the data could be used for model training.

We began with items_info.dat, the file containing book metadata. While it contains 17,384 rows with exactly 9 columns each, reading it into a DataFrame using pandas.read_csv with tab separation and skipping bad lines (on_bad_lines='skip') resulted in only 16,411 valid rows. To investigate, we manually checked the improperly formatted rows and discovered that many contained an HTML-encoded ampersand (&amp) followed by a tab. Since the dataset is tab-separated, this caused the line to be split into more columns than expected.

To fix this, we wrote code to replace '&\t' with a simple '& ', preserving the ampersand but removing the unintended tab-separator. After writing this corrected version to a new file, we were able to parse 17,357 rows successfully—only 27 short of the expected count.

To further clean the remaining problematic rows, we inspected lines that still did not have exactly 9 columns. We observed that these lines contained tabs in arbitrary and incorrect positions, so we manually removed them. The final cleaned dataset, stored in items_info_cleaned_2.dat, contained all rows properly formatted with the expected number of columns. This cleaned dataset was then used to create the enriched dataset using the ISBN-DB API, as described in the next section.

2.2 ISBN-DB API Integration for Enhancing Book-Crossing Dataset

Because the Book-Crossing dataset has previously been used for other projects by other researchers, we wanted to find a way to make our project different. We felt like the dataset was missing some relevant metadata that could be helpful for our algorithms and our approaches. So, to enhance the `books_info` dataset from the Book-Crossing dataset, we utilized the ISBN-DB API to enrich the dataset with additional book-related information.

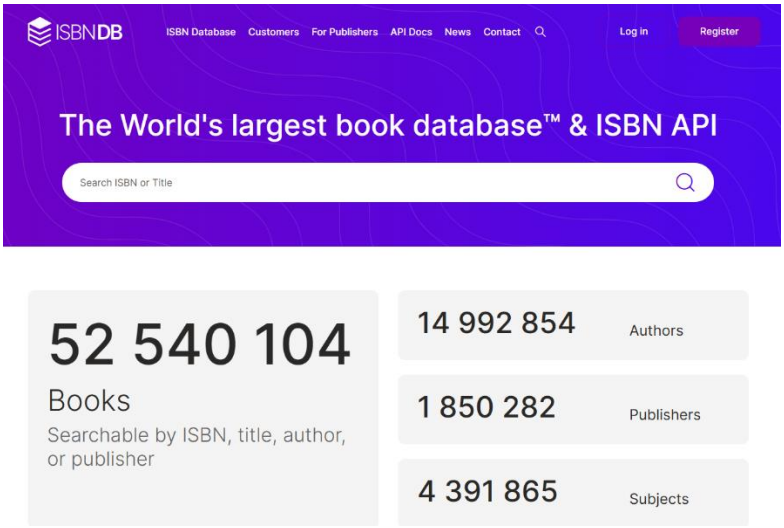

Figure 2.1: ISBN-DB API Website Interface

This next section outlines the process and methodology followed in integrating the ISBN-DB API for data enrichment. For the full code used to do these requests, please see the Appendix A and the Colab file linked in the code section.

2.2.1 API Key and Setup

The ISBN-DB API was accessed using a unique API key provided by the service upon purchase. This key was necessary for making authenticated requests to the API and retrieving book information.

```
# --- CONFIGURATION ---
API_KEY = "59541_7b280637020d60a99abb89a246a1df18"
BASE_URL = "https://api2.isbndb.com/book/"
BATCH_SIZE = 10  # Since the API allows max 10 results per call
DAILY_LIMIT = 2000  # Academic plan limit
SAVE_FILE = "/content/isbn_enriched_data.csv"  # Where enriched data is saved
SKIPPED_FILE = "/content/skipped_isbns.csv"  # Log of invalid ISBNs
```

Figure 2.2: Our personal API Key (see blue arrow) was used to make authenticated requests

2.2.2 Querying the ISBN-DB API

For each book in the `books_info` dataset, a corresponding ISBN (International Standard Book Number) was identified. Using these ISBNs, we made API calls to retrieve detailed information about each book.

The ISBN-DB API supports querying book details via the ISBN, which allows us to access various metadata, including:

- **ISBN10:** The 10-digit identifier for the book in the ISBN system.
- **ISBN13:** The 13-digit identifier for the book in the ISBN system.
- **Title:** The official title of the book.
- **Author:** The primary author(s) associated with the book.
- **Publication Date:** The date the book was published.
- **Publisher:** The publishing company or organization that published the book.
- **Binding:** The material or method used to bind the book (e.g., hardcover, paperback, etc.).
- **Pages:** The total number of pages in the book.
- **List Price:** The recommended retail price set by the publisher or seller.
- **Cover Image:** A visual representation of the book's cover.
- **Language:** The language in which the book is written (e.g., English, French, Spanish).
- **Edition:** Indicates the version or iteration of the book (e.g., 1st edition, revised edition).
- **Format:** The physical or digital format of the book (e.g., eBook, audiobook, print).
- **Synopsis:** A brief description or summary of the book, outlining the plot or main ideas.
- **Subject:** The main topics or themes addressed in the book (e.g., History, Science, Fiction).
- **Weight:** The weight of the book.
- **Dimensions:** The physical size of the book.

Pulling information from the API required a lot of computing power. As we were running this on our personal student laptops, it required us to select which additional metadata we wanted to use to enhance our dataset. After lots of discussion, we selected the ones we thought were the most useful, such as:

- **ISBN13:** We thought it would be interesting to have an alternative identifier for each book.
- **Subjects:** Since we don't have a genre category in the original Book-Crossing book_info dataset, we felt like adding the subjects to the dataset would be quite useful For example, for *Decision in Normandy* by Carlo D'Este, the subjects are "History", "Military" and "World War II". This could be quite useful to use in a case where we are trying to recommend books to a user that is likely interested in historical or military themed books.
- **Language:** We noticed that there are books that are in various languages. This could be interesting to consider in our approach as some people might only read in English or only read in French. While others might read some in English and some in French.
- **Numbers of pages:** We felt like this could be interesting to add to our metadata, as we might be able to detect trends within the users. Having this could help us pick up when some users may only read books that are under 300 pages, while others only read books that are over 500 pages.
- **Synopses:** We felt like adding this to the metadata could allow our approaches to pick up on key words that the subjects didn't identify.

These pieces of information were crucial for enhancing the dataset and providing more context for each book in the Book-Crossing dataset.

2.2.3 Handling API Limitations

The ISBN-DB API imposes certain limits on the number of requests that can be made in a given period. We used an academic subscription plan, which had certain limitations like:

- **Daily Request Limit**: The API allows a maximum number of 2000 requests per day.
- **Results Limit per Request**: Each query can return up to 10 results per request.

To efficiently handle these limitations, we implemented a batching mechanism. The ISBNs from the Book-Crossing dataset were processed in manageable batches, ensuring that we did not exceed the daily request limit. Essentially, every day, we would run the code to process a new batch. It would first verify which ISBNs had already been verified, then it would start querying for the ones that were still missing. This "bookmark" approach allowed us to obtain the information we needed to process all 17360 valid ISBNs.

```
Processing batch 1734/1737...
Batch 1734 completed. Progress saved.
Processing batch 1735/1737...
Batch 1735 completed. Progress saved.
Processing batch 1736/1737...
Batch 1736 completed. Progress saved.
All valid ISBNs processed successfully or daily limit reached.
```

Figure 2.3: Processing the batches and saving the progress until all of them were completed

2.2.4 Data Processing and Enrichment

The data retrieved from the ISBN-DB API was parsed into a structured format for integration with the Book-Crossing dataset. The following steps were followed:

- **Data Extraction**: For each ISBN queried, the relevant data (title, author, genre, and synopsis) was extracted from the API response.
- **Data Cleaning**: The extracted data was cleaned to handle missing or incomplete information, ensuring consistency across the dataset.
- **Data Integration**: The enhanced book details from the ISBN-DB API were then merged with the existing `books_info` dataset, enriching it with new attributes like genre, author(s), and synopsis.

2.2.5 Error Handling and Logging

Not all ISBNs from the Book-Crossing dataset returned valid data from the ISBN-DB API. In such cases, the invalid ISBNs were logged in a separate file (`skipped_isbns.csv`) for further review or reprocessing. This ensured that no data was lost and that we could revisit problematic ISBNs if needed. It also meant that we were avoiding unnecessary retries during the main enrichment process and avoiding wasting precious API calls.

After evaluating the ISBNs that were skipped by the program, we made the decision to remove them as they had little to no impact on the overall picture (only 24 of them were invalid and they were rarely to never accessed by the users in the system).

2.3 Final Enriched Data

After the API data was integrated into the Book-Crossing dataset, the final enriched dataset included additional book attributes that were previously absent.

We were most excited about the "subjects" column, but unfortunately, 56 books had none attributed to them. We decided to fill these in manually by doing some research on these books.

Upon completion, these enhancements provided richer metadata for each book, enabling more effective book recommendations based on various attributes such as genre and author. The enriched dataset serves as a valuable resource for content-based filtering and collaborative filtering tasks in the recommender system.

	Book_ID	ISBN10	Book-Title	Book-Author	Year-Of-Publication	Publisher	isbn13	subjects	synopsis	language	pages
0	1	0060973129	Decision in Normandy	Carlo D'Este	1991	HarperPerennial	9780060973124	History|Military|World War II	An outstanding military history that offers a ...	en	555.0
1	2	0393045218	The Mummies of Urumchi	E. J. W. Barber	1999	W. W. Norton & Company	9780393045215	Arts & Photography|Decorative Arts & Design|Te...	Barber, one the world's leading authorities on...	en	240.0
2	3	0425176428	What If?: The World's Foremost Military Histor...	Robert Cowley	2000	Berkley Publishing Group	9780425176429	History|Historical Study & Educational Resourc...	With its in-depth reflections on the monumenta...	en	416.0
3	4	0452264464	Beloved (Plume Contemporary Fiction)	Toni Morrison	1994	Plume	9780452264465	Literature & Fiction|Genre Fiction|Historical|...	Sethe, an escaped slave living in post-Civil W...	en	288.0
4	5	0609804618	Our Dumb Century: The Onion Presents 100 Years...	The Onion	1999	Three Rivers Press	9780609804612	Humor & Entertainment|Humor|Parodies|Political...	The Onion has quickly become the world's most ...	en	176.0

Figure 2.4: The head of our merged data

3. Methodology and Analysis

In this section, we describe the different approaches we used to build personalized book recommendation systems. We begin by exploring a basic content-based filtering method, followed by an enriched content-based model, collaborative filtering techniques, and finally, a hybrid recommendation approach that combines multiple strategies to improve recommendation quality.

3.1 Content-Based Filtering

Content-based filtering recommends items to a user based on the characteristics of the items (in this case, books), rather than relying on interactions from other users. In this project, books were represented by a combination of features such as title and author. The intuition behind this approach is that if a user liked a certain book, they are likely to enjoy other books that share similar attributes.

Before creating the enriched dataset, we first developed a basic content-based filtering model using only the limited metadata available in the original dataset. For the purposes of this report, we still describe this initial model, as it provides a simple, intuitive example of how content-based filtering works before introducing more complex enhancements.

3.1.1 Initial Model

In the initial model, we selected book title, author, and publisher to represent each book. These features were concatenated into a single string, to provide a unified textual description for each book:

	Book-Title	Features
0	Decision in Normandy	Decision in Normandy Carlo D'Este HarperPerennial
1	The Mummies of Urumchi	The Mummies of Urumchi E. J. W. Barber W. W. N...
2	What If?: The World's Foremost Military Histor...	What If?: The World's Foremost Military Histor...
3	Beloved (Plume Contemporary Fiction)	Beloved (Plume Contemporary Fiction) Toni Morr...
4	Our Dumb Century: The Onion Presents 100 Years...	Our Dumb Century: The Onion Presents 100 Years...

Figure 3.1: Construction of the Combined Textual Feature (Title, Author, Publisher) for Each Book

This is used as input for the TF-IDF (Term Frequency–Inverse Document Frequency) vectorizer. TF-IDF is a common technique in natural language processing that reflects how important a word is to a document relative to a collection of documents. It assigns higher weights to words that are unique to specific books (such as a distinctive author or keyword) and lower weights to common words that appear across many books (such as "the" or "and"), helping to emphasize distinguishing characteristics of a book. It is a sparse matrix where the columns are the words, and the rows are the books, and each entry represents the TF-IDF score of a particular word in a particular book.

Once each book was represented numerically through its TF-IDF vector, we computed cosine similarity between all book pairs. Cosine similarity measures the cosine of the angle between two vectors in a high-dimensional space and gives a score between 0 (completely dissimilar) and 1 (identical). Books with higher cosine similarity scores are considered more alike. Finally, for a given book, the system recommended the top N most similar books based on these scores, excluding the book itself (which has a score of 1) to avoid recommending it back to the user.

Here is the output from our `simple_content_based_filtering()` function, for the first book in the dataset, *Decision in Normandy* by Carlo D'Este:

	Book-Title	Book-Author	Publisher
12986	Decision at Doona	Anne McCaffrey	Del Rey Books
17061	The Eleventh Summer	Carlo Gebler	Simon & Schuster
2756	Due di due (Bestsellers)	Andrea De Carlo	A. Mondadori
12328	Io Francesco	Carlo Carretto	Distribooks Inc
10374	Your Sacred Self: Making the Decision to Be Free	Wayne W. Dyer	HarperTorch

Figure 3.2: Top 5 Recommended Books for *Decision in Normandy* using Initial Model

As we can see, the method finds *Decision at Doona* by Anne McCaffrey as the best recommendation. This is not a surprising result – it seems that the method found this book to be similar because of the common word 'Decision' in both titles. This highlights how the model captures surface-level similarities, even if the books themselves might not be closely related in theme or content. Indeed, *Decision in Normandy* is a historical analysis of the planning and execution of the D-Day invasion during World War II, whereas *Decision at Doona* is a science fiction novel about human colonization on a distant planet.

Although this model successfully produced relevant recommendations based on surface-level similarities, it has limitations:

- It cannot understand the actual topics, genres, or storylines behind the books.
- Books with very generic titles or common authors could dominate the results.
- Publisher is likely not a very important indicator of book similarity, as many publishers release a wide variety of books across different genres and topics.

This basic model, however, served as an important foundation for later improvements, where we integrated richer metadata into a more advanced content-based filtering system. See Appendix B for implementation details for this section.

3.1.2 Enriched Model

In the initial content-based model, we relied on title, author, and publisher to describe a book. However, we found these features to be limited in their ability to capture the actual content of a book, so we enhanced our dataset using the ISBN-DB API described in section 2.2. Specifically, we chose to add subjects and synopsis to our *enriched* model and removing publisher, which we found to be a less meaningful indicator of similarity.

Instead of concatenating all the features (title, author, subjects, and synopsis) into a single string as before, we applied TF-IDF vectorization separately for each one. This approach preserves the unique informational value of each feature. For example, a word appearing in the synopsis might convey genre or plot, while a word in the subjects field might represent a theme.

Next, we defined 15 content-based "strategies", which represent all possible non-empty combinations of the four features (i.e., $2^4 - 1 = 16 - 1 = 15$ combinations):

- Title
- Author
- Subjects
- Synopsis
- Title + Author
- Title + Subjects
- Title + Synopsis
- Author + Subjects
- Author + Synopsis
- Subjects + Synopsis
- Title + Author + Subjects
- Title + Author + Synopsis
- Title + Subjects + Synopsis
- Author + Subjects + Synopsis
- Title + Author + Subjects + Synopsis

Each strategy also defines feature weights. We decided to weigh features equally within each strategy. For example, in the Author + Subjects strategy, the final book representation is based 50% on author and 50% on subjects. More technically, this means that the final TF-IDF matrix is a horizontal stack of the author TF-IDF matrix and the subjects TF-IDF matrix, with each matrix scaled by its respective weight before stacking.

To identify similar books from the TF-IDF matrix, we trained a Nearest Neighbors model using cosine similarity as the distance metric. In our initial implementation, we directly computed pairwise cosine similarities between all books, as in the Initial Model. However, with the addition of new features like subjects and synopsis, the TF-IDF matrices became significantly larger. Since we generated multiple TF-IDF matrices and combined them, memory usage increased substantially. This caused our program to exceed available random-access memory and crash. To address this, we switched to a Nearest Neighbors approach, which computes similarities only when needed (rather than storing all pairwise scores), making it more memory-efficient and scalable.

Since each strategy defines a different combination of features, we trained a separate Nearest Neighbors model for each of the 15 strategies. To ensure language consistency, we did this per language as well, resulting in a total of 15 × 23 = 345 models (since there are 23 unique book languages in the dataset).

To generate recommendations from the trained Nearest Neighbors models, we implemented the `improved_content_filter()` function. This function takes a book title, a strategy name, and a number of desired recommendations as input, and returns the top N most similar books according to the selected strategy.

The function first identifies the language of the input book, since each model is language-specific. It then locates the appropriate Nearest Neighbors model based on the provided strategy and language. Using the model, it retrieves the top N+1 most similar books from the TF-IDF feature matrix—one more than needed, since the most similar book is the book itself with a perfect score of 1.

To avoid recommending duplicate editions or alternate formats of the same book, we apply additional filtering. This is handled by the `normalize_title()` function, which converts titles to lowercase, removes punctuation and text within parentheses, and normalizes whitespace. For example, *Dracula* and *Dracula (Tor Classics)* would normalize to the same base string. We compare normalized titles to exclude from the list of recommendations books that are effectively the same as the original, or that have already appeared in the recommendation list. Finally, the top N filtered recommendations are returned as a DataFrame containing their titles, authors, and publishers.

For example, we see now that recommendations for *Decision in Normandy* using the Title + Author + Subjects + Synopsis strategy are different from the initial model:

	Book-Title	Book-Author	Publisher
2420	Under Blood Red Sun	Graham Salisbury	Bantam Doubleday Dell Publishing Group
6753	The Greatest Generation	TOM BROKAW	Random House
1542	Band of Brothers : E Company, 506th Regiment, ...	Stephen E. Ambrose	Simon & Schuster
9100	Torpedo Junction: U-Boat War Off America's Eas...	Homer Hickam	Dell Publishing Company
5310	The Red Orchestra (Witnesses to War)	Gilles Perrault	Random House Inc

Figure 3.3: Top 5 Recommended Books for *Decision in Normandy* using Enriched Model

We see that for example, the first recommendation is *Under Blood Red Sun* by Grahan Salisbury, is a historical novel set during World War II that focuses on the experience of a Japanese-American boy in Hawaii. This result shows that the enriched model is now capturing deeper thematic relevance rather than surface-level similarities like title overlap in the first recommendation of the Initial Model. See Appendix C for implementation details for this section.

3.2 Collaborative Filtering

3.2.1 Baseline Collaborative Filtering

In this method, we implement a baseline collaborative filtering approach that predicts a user's rating for a book by combining the overall average (global mean) with adjustments (biases) specific to the user and the item. This technique provides a simple yet effective means of accounting for systematic deviations—some users may consistently rate books higher or lower than average, and some books might generally receive better or worse ratings regardless of the user. By incorporating these biases, the model can offer reasonable predictions even in the absence of complex relationships.

The implementation begins with the `predict_baseline` function. The first step checks whether the target user and book exist in the dataset; if either is missing, the function returns a missing value. If both are present, it calculates the global mean of all ratings as the baseline expectation. The next step is to compute the user bias, which is determined by comparing the average rating given by the user to the global mean. Similarly, the item bias is calculated by evaluating the average rating received by the book in comparison to the overall mean. The predicted rating is then the sum of the global mean, the user bias, and the item bias. This calculation effectively adjusts the general trend (the average rating across the board) by considering individual tendencies—a user's inclination to rate more favorably or harshly, and an item's popularity or lack thereof.

Following this, the `recommend_books_baseline` function generates personalized recommendations for a specified user. It first identifies the books the user has not yet rated. Then, it iterates over these unseen books and uses the `predict_baseline` function to estimate the rating for each. The predicted ratings are sorted in descending order, and the top recommendations are selected. To enhance the output, the function loads additional metadata from an external dataset and merges this data with the predicted ratings. This produces a comprehensive list of recommended books that includes not only the title, author, and publisher but also the predicted rating, offering a more complete picture of how much the user might enjoy the book.

Overall, this baseline collaborative filtering method is intuitive and transparent. It tailors predictions to individual user behavior and item characteristics, providing a robust foundation for more complex recommendation algorithms. While simple, this method offers valuable insights into how systematic biases affect rating behavior and serves as an excellent baseline against which other techniques can be measured.

3.2.2 User-Based Collaborative Filtering Using Cosine Similarity

In this method, we implement a user-based collaborative filtering approach to provide book recommendations. The primary idea behind this technique is that users with similar past behavior (as reflected in their ratings for various books) are likely to share similar tastes in the future. By identifying

users with similar preferences, we can recommend books that the target user has not yet rated but might enjoy.

We begin by constructing a user-item matrix where each row represents a user and each column corresponds to a specific book. The cells of this matrix are populated with the ratings provided by users. Any missing ratings are filled with zeros, assuming that an absence of rating implies neutrality (i.e., no preference or opinion).

Next, we calculate the similarity between users using cosine similarity. Cosine similarity measures the orientation of two vectors, representing the rating profiles of two users. A similarity score close to 1 indicates that the users have nearly identical taste profiles, while a score closer to 0 suggests little similarity. The resulting similarity matrix is stored in a data structure, where both dimensions correspond to the users, making it easy to look up how similar any pair of users is.

For the recommendation process, the algorithm proceeds as follows:

- Neighbour Identification: For a given target user, the algorithm identifies other users with the highest similarity scores. The target user is excluded from this set, as their similarity to themselves is trivially perfect. The top N most similar users are selected for further analysis.
- Aggregation of Favorite Books: Once the similar users are identified, the system collects the books they have rated. However, it filters out any books the target user has already rated, ensuring that the recommendations consist only of unseen books.
- Ranking and Recommendation: The remaining books (those rated by similar users but not yet rated by the target user) are grouped by title. The average rating for each book is calculated, and the books are then ranked based on these average ratings. The highest-ranked books are selected as recommendations.

This method is effective and interpretable. By relying on user behavior patterns, it leverages straightforward statistical techniques (like cosine similarity) to provide recommendations based on community preferences. It offers a clear, understandable way to generate personalized recommendations and can easily be extended with more sophisticated models.

In summary, user-based collaborative filtering with cosine similarity identifies users with similar preferences and recommends books that those users have enjoyed but that the target user has not yet experienced. This approach is an excellent baseline for recommender systems, providing personalized recommendations based on the behavior of others with similar tastes. See Appendix E for implementation details for this section.

3.2.3 SVD-Based Latent Factor Model for Collaborative Filtering

In this method, we use Singular Value Decomposition (SVD), implemented via the Surprise library, to build a latent factor model for personalized book recommendations. The core goal of this approach is to uncover hidden patterns in user ratings that represent abstract qualities (such as genre, writing style, or overall popularity), without needing to explicitly define these attributes.

The input to this model is the user-item ratings matrix, constructed from the book_ratings dataset. Even though the book_ratings table looks dense (fully populated with ratings from 1 to 10), it is actually just a list of interactions. The real user-book matrix (if we tried to create it) would be very large and sparse.

We have:

- 1295 users
- 14,684 books
- 62,657 ratings recorded

Then the full user-book matrix would be 1,295 × 14,684 = 19,015,780 cells, but we only have 62,657 known ratings (only 0.3295% of the matrix is filled).

Since the matrix is sparse, we make use of SVD to transform this sparse matrix into two matrices: Users x Features and Features x Books. Multiplied, we get Users x Books.

To prepare the data for model training, the ratings are converted into a Surprise Dataset object, which enables compatibility with the library's recommendation routines. For evaluation, the dataset is split into an 80% training set and a 20% test set, allowing us to assess performance using metrics such as root mean squared error (RMSE).

Next, we instantiate and train the SVD model with 50 latent factors. These latent factors (e.g., genres, wrting styles, book popularity, etc.) are not directly observed; rather, SVD automatically discovers them by decomposing the user-item matrix. These factors can be thought of as abstract dimensions that capture subtle relationships and preferences inherent in the data. Once trained, the model is used to make predictions on the test set, and RMSE is calculated to assess the accuracy of the predictions.

After the model is evaluated, we define helper functions to first determine which books a user has already rated, and then identify those the user has not yet rated. By looping through the unseen books, the model predicts the potential rating for each. These predictions are sorted in descending order, and the top recommendations are selected. See Appendix F for implementation details for this section.

3.2.4 KNN-Based Collaborative Filtering

This method leverages a k-nearest neighbors (KNN) approach within collaborative filtering to predict a user's rating for an unseen book. Like user-based collaborative filtering using cosine similarity, this technique assumes that similar users tend to rate items similarly. However, KNN is a more specific and complex method, which computes user-to-user similarities using cosine similarity, then identifies a set of neighbors for each target user. For any book that the user has not rated, the rating is predicted as a weighted average of the ratings from these neighbors, with more similar users having a greater influence.

The process starts with the construction of a user-item ratings matrix, where each row represents a user's ratings across various books. Missing values are filled with zeros. The cosine similarity between users is then computed, resulting in a similarity matrix that quantifies how closely users' preferences align.

When it comes time to estimate the rating for a particular user-book combination, the algorithm first checks whether the target book has been rated by any of the similar users. If ratings are available, the algorithm filters the neighbors to include only those who have rated the book. The next step involves selecting the top K most similar users, based on their cosine similarity scores. The final prediction is a weighted average of the neighbors' ratings, where the weights are the similarity scores. This ensures that more similar users contribute more to the prediction, making it more personalized and reliable.

After generating predictions for all unseen books, the method sorts the predictions and selects the top N books with the highest predicted ratings.

This KNN-based method is particularly useful for tasks that require personalization based on direct user similarity. By focusing on the most similar users, it often provides more tailored recommendations that resonate with individual preferences, making it a valuable tool within collaborative filtering.

3.2.5 LightFM-based Collaborative Filtering (Content-Aware)

These methods use the LightFM library, a Python toolkit for building recommender systems. Although technically a hybrid recommender, it still is largely collaborative, and therefore we classify it as one of our collaborative methods. It works by learning latent representations of users and items from interaction data, while incorporating content features (title, author, subjects, and synopsis) to improve performance in sparse or cold-start settings.

3.2.5.1 LightFM with Book Ratings

In this method, user ratings are converted into binary interactions, indicating whether a user liked a book based on a threshold (*1 = liked* if rating is greater than 5, *0 = disliked* otherwise). This binary simplification aligns with LightFM's focus on implicit feedback, where the emphasis is on the presence or absence of interaction rather than fine-grained rating differences. The system builds an internal mapping of users, books, and features derived from the book metadata, enabling the model to combine collaborative signals from user ratings with content-based signals from descriptive book features.

The core of the model is training a LightFM model with the Weighted Approximate-Rank Pairwise (WARP) loss function. This loss function optimizes the ranking quality instead of prediction accuracy. Instead of predicting a rating (like 1-10), WARP ranks books based on likelihood of interest. This approach is well-suited for implicit feedback scenarios, where the goal is to learn user preferences based on interactions rather than explicit ratings. During training, the model learns latent representations for both users and books, capturing underlying patterns that drive user preferences.

Once the model is trained, it predicts scores for all books that a user has not yet interacted with. These scores are derived from the interaction between the user's latent factors and the book features, leading to a ranking of books that are most likely to be of interest to the user. The system retrieves detailed metadata for the top recommendations. See Appendix H for implementation details for this section.

3.2.5.2 LightFM with Book History

This method is very similar to the Book Ratings approach, but it skips the step of converting ratings into implicit feedback. In the book history dataset, an interaction is already recorded as a 1 if a user accessed a book. We adopted the heuristic that accessing a book indicates a positive preference. Accordingly, we treat 1 as "liked" and 0 (no access) as "not liked." While this is a simplifying assumption that does not always reflect a user's true opinion, it still provides a useful input for training the LightFM model.

The rest of the implementation follows the same structure as the LightFM with Book Ratings method, with construction of a user-item interaction matrix using access events as positive feedback, building item features from book metadata (title, author, subjects, and synopsis), and training the LightFM model using the WARP loss function to learn latent relationships for personalized ranking. See Appendix I for implementation details for this section.

3.3 Cold-Start Problem and Serendipity-Based Recommendations

The Cold-Start Problem refers to the challenge faced by recommender systems when there is insufficient data to make personalized recommendations, particularly when dealing with new users or new items. In our system, we implemented a solution to this problem that not only addresses the lack of user data but also helps introduce a novelty and serendipity aspect into our recommendations. The core idea is to provide recommendations based on age and location (two demographic factors), allowing the system to make educated suggestions even when no prior ratings or preferences are available.

At first glance, this method was designed to address the Cold-Start Problem for new users who have never rated a book. We imagined scenarios where users either forget which books they have read or are simply starting fresh. In such cases, they can provide basic demographic information (e.g., age and location), and the system will return a list of books that are highly rated by users within the same age group and location. This approach leverages the wisdom of other users in similar demographic groups to make suggestions. Though these recommendations might not be perfect, they can serve as a starting point for these users.

However, as we refined this method, we realized that it could serve a dual purpose: not just solving the cold-start problem, but also introducing serendipity into the recommendation system. We recognize that a key aspect of a good recommender system is the ability to suggest books that are somewhat different from what a user typically reads, to encourage discovery and exploration. By using the cold-start function based on age and location, we can "randomly" suggest books that users from similar demographics have rated highly, but which might fall slightly outside their usual preferences. These are educated suggestions, not completely random choices, and they are likely to provide interesting and diverse recommendations that still align with the user's general tastes.

For example, rather than suggesting *Diary of a Wimpy Kid* to an older user who typically reads history books, the system could suggest books that have been popular among users in the same age group and location but from different genres, offering something fresh but still relevant. This adds a level of serendipity to the system, ensuring that recommendations are both personalized and diverse, helping users explore beyond their usual reading habits.

3.3.1 Cold-Start Code Breakdown and Thought Process:

Here is an overview of the code that was implemented as a solution to the Cold-Start problem:

- User Data Preprocessing: We start by loading and preprocessing the demographic data (age and location). The location strings are standardized (converted to lowercase and stripped of extra whitespace), and we ensure that the age column contains valid numeric data.
- Cold-Start Recommendation Function: The function `recommend_books_cold_start` takes a new user's age and location as inputs and identifies similar users based on those demographic factors. The function filters users within an acceptable age threshold (for example, if the user is 23, the system can be set to look in the 21-25-year-old age range) and matches users by location. If no users match both criteria, the system falls back to recommending based on age alone.
- Retrieving Book Ratings: After identifying similar users, the function filters the book ratings dataset to include only ratings from those users. If no ratings are found from similar users, it defaults to using the overall book ratings. We do this as a back-up so that the user can have some recommendations of where to start their reading journey, no matter what.

- Aggregating Ratings and Sorting: The function calculates the average rating for each book based on the similar users' ratings and sorts the books by the highest average rating. The top N books are then selected as the recommendations.
- Recommendations: Finally, the function provides the full set of recommendations for this user.

In summary, by using the cold-start approach based on age and location, we can offer recommendations that feel personalized, even for new users with no prior ratings. Additionally, when used in our main recommender system, we call this method the `AGE-LOCATION` method. Using this method introduces novelty and serendipity by suggesting books that are both relevant to the user's demographic and diverse enough to encourage discovery. It allows the system to recommend books that the user might not have thought of themselves, helping to diversify their reading habits while maintaining a sense of personalization. See Appendix J for implementation details for this section.

3.4 Hybrid Recommender

Before introducing our full hybrid recommendation system, we first developed a simpler comparison matrix to evaluate the consistency of content-based strategies. That is, we created a matrix that tracks which books were recommended based on a single input book, and how many of the 15 content-based strategies selected each one. Figure 4 below shows an example of this matrix.

	Book-Title	title	author	subjects	synopsis	title_author	title_subjects	title_synopsis	author_subjects	author_synopsis	subjects_synopsis	title_author_subject	title_author_synopsis	title_subjects_synopsis	author_subjects_synopsis	title_author_subjects_synopsis	# Content-Based Strategies	% Content-Based Strategies
10	Under Blood Red Sun	0	0	1	0	0	1	0	1	0	1	1	0	1	1	1	8	53.33
3	Band Of Brothers : E Company, 506Th Regiment, ...	1	0	0	0	1	1	1	0	0	0	1	1	1	0	1	8	53.33
13	The Greatest Generation	0	0	1	0	0	1	0	1	0	1	1	0	1	1	1	8	53.33
12	The Red Orchestra (Witnesses To War)	0	0	1	0	0	1	0	1	0	1	1	0	0	1	1	7	46.67
11	Battle Of Midway Island (The Great Battles Of ...	0	0	1	0	0	1	0	1	0	1	1	0	1	1	0	7	46.67
14	Torpedo Junction: U-Boat War Off America'S Eas...	0	0	1	0	0	0	0	1	0	1	0	0	1	1	1	6	40.00
1	Your Sacred Self: Making The Decision To Be Free	1	0	0	0	1	0	1	0	0	0	0	1	0	0	0	4	26.67
0	Decision At Doona	1	0	0	0	1	0	1	0	0	0	0	1	0	0	0	4	26.67
5	The Eleventh Summer	0	1	0	0	1	0	0	0	1	0	0	1	0	0	0	4	26.67
2	Decision In Philadelphia: The Constitutional C...	1	0	0	0	1	0	1	0	0	0	0	1	0	0	0	4	26.67
15	For Her Own Good : 150 Years Of The Experts' A...	0	0	0	1	0	0	1	0	1	0	0	0	0	0	0	3	20.00
16	Trevor'S Place: The Story Of The Boy Who Bring...	0	0	0	1	0	0	0	0	1	0	0	0	0	0	0	2	13.33

Figure 3.4: Content-Based Strategy Overlap Matrix for *Decision in Normandy*

The recommended books are listed in the first column and the next 15 columns are each content-based strategy. For each recommended book, an entry under a strategy is marked as 1 if that strategy recommended the book, and 0 if it did not. Note that we take the top 10 books per strategy. The last two columns contain the total count and percentage of strategies that recommended the book. The table is sorted in descending order of strategy overlap, so books that were recommended by the most strategies appear at the top. *Under Blood Red Sun* is recommended by 8 content-based strategies, which indicates that this is potentially a good recommendation! It is important to note however that many of the strategies are covaried (e.g. Title + Subjects and Author + Subjects but contain subjects), and therefore more likely to recommend the same book due to the same underlying feature. See Appendix K for implementation details for this matrix.

This matrix is a helpful in showing recommendations based on a single **book**, but ultimately what we would like to do is give recommendations to a **user**. Therefore, we created a hybrid matrix that combines the content-based and collaborative filtering techniques we implemented. See Appendix L for implementation details for this matrix. However, because our content-based filtering approach generates recommendations using individual books as input, we begin by selecting the user's top N rated books (we used N = 3 in our experiments). For each of these books, we apply the 15 content-based strategies, creating 15 × 3 = 45 columns. Then we have 7 more columns, one for each of the collaborative filtering methods – SVD, KNN, Baseline, Age-Location, LightFM-Ratings, LightFM-History and Similarity.

As summary statistics, we have:

- Two columns for the total count and percentage of content-based strategies that recommended the book
- Two columns for the total count and percentage of collaborative strategies that recommended the book
- Two columns for the total count and percentage of all strategies (content-based and collaborative) that recommended the book.
- One column for the **Final Score**, which we designed with careful consideration, in a manner that we deemed effective to determine the final ordering of the book recommendations

Here is what these columns look like for user 1 in our dataset:

# Content-Based Strategies	% Content-Based Strategies	# Collaborative Filtering Strategies	% Collaborative Filtering Strategies	# Total Filtering Strategies	% Total Filtering Strategies	Final Score
0	0.00	3	42.86	3	5.77	0.4250
8	17.78	1	14.29	9	17.31	0.3672
8	17.78	1	14.29	9	17.31	0.3672
0	0.00	2	28.57	2	3.85	0.2833
0	0.00	2	28.57	2	3.85	0.2833
0	0.00	2	28.57	2	3.85	0.2833
0	0.00	2	28.57	2	3.85	0.2833
0	0.00	2	28.57	2	3.85	0.2833
0	0.00	2	28.57	2	3.85	0.2833
0	0.00	2	28.57	2	3.85	0.2833

Figure 3.5: Summary Columns for User 1

We see that the second recommended book is recommended by 8 content-based strategies, which is 17.78% of all content-based strategies. It is also recommended by 1 collaborative strategy, which is 14.29% of all collaborative strategies. Together, the book is recommended by 9 strategies, which is 17.31% off all strategies. The final score is 0.4044, which is computed in the following manner:

- 42.5% Content-Based Score: This is the proportion of content-based strategies that recommended the book. So, in the case of the second book in the example above this is (8/45) × 0.425, where 45 is the total number of strategy-book pairs tested (15 strategies × 3 input books).
- 42.5% Collaborative Filtering Score: This is the **normalized** proportion of collaborative filtering methods that recommended the book. We chose to do a normalized proportion, since collaborative filtering strategies overlap less frequently. For example, for user 1, the highest overlap is 3 out of 7 collaborative strategies. Therefore, we consider this a meaningful overlap among collaborative methods and assign it a full score of 3 out of 3 for the collaborative component. So, for the final score it would contribute is (3/3) × 0.425 = 0.425. If a book is recommended by 1 collaborative strategy, this would contribute (1/3) × 0.425 = 0.1417 to the final score.
- 15% Hybrid Boost: We believed that if a book is recommended by a content-based and collaborative method, it should get a boost in score so that it would rank higher in the list of recommendations. Therefore, we added 0.15 to the book's score if it is recommended by at least one content-based strategy and at least one collaborative strategy. In the example above, there are only two books that are recommended by both types of strategies – the second and third book in the list. These two books had a final score of 0.3672, slightly below the top-scoring book, which had a Final Score of 0.4250, since it is recommended by 3 collaborative strategies.

Overall, this ranking seems to work reasonably well. The first recommendation is recommended by three collaborative methods and the next two are recommended by eight content-based methods and one collaborative method. The next fourteen recommendations are recommended two collaborative methods and no content-based methods. The next 34 recommendations are recommended by one collaborative method and no content-based methods, and the following recommendation is recommended by 15 content-based methods and no collaborative methods.

However, this final score design is not ideal. While first three recommendations are recommended for unique reasons, the following fourteen are all recommended for the same reason (two overlapping collaborative methods) and have an identical score of 0.2833. This results in a long tie, making the ranking within that group arbitrary. Preferably, we would see greater variability among the top-ranked recommendations, not only in their scores but also in the types of strategies that supported them. That said, it is at least a positive sign that the specific pairs of collaborative methods vary across those fourteen books – we see pairs of SVD and KNN, Age-Location and LightFM-History, and LightFM-Ratings and LightFM-History.

We encounter the same issue with the next 34 recommendations, which are also tied at a score of 0.1417. Perhaps this number could be reduced by recommending less books per strategy. We tested with 10 books per strategy, but there is a trade-off since lowering this number could also reduce the likelihood of overlap between strategies.

Nevertheless, the matrix displays the reason for each book recommendation (i.e., which strategies recommended it), so if a user would like to delve deeper into why a certain book was recommended to them, they can.

3.5 Recommender for Friends

To personalize our project, we wanted to be able to recommend books not only users in the dataset, but also for our own friends. To do this, we asked eight friends for about 10 books they like and about 5 books they dislike, rating each one on a scale from 1 to 10 (as the ratings are in the dataset). Then, we would be able to give them book recommendations using the same approach we used for dataset users.

However, due to time constraints, we only implemented a friend recommender with content-based strategies. Supporting friend input required several modifications to our code—for instance, we had to match each book provided by a friend to the dataset, accounting for formatting differences in titles and filtering out any books that didn't exist in the dataset. So we created a version of the matrix shown earlier (Figure 3.4), tailored for friends. For example, for one of our friends, their top-rated books that were found in the dataset were *Dracula* by Bram Stoker, *The Godfather* by Mario Puzo, and *Frankenstein* by Mary Shelley. The first three recommended books for this friend, were *The Sicilian* by Mario Puzo, *Omerta* by Mario Puzo, and *Fools Die* by Mario Puza, with 31.11%, 26.67%, and 22.22% of content-based strategies recommending the books, respectively.

We see that in this case, the top three recommended books are all by the same author of *The Godfather*. They are also similar in themes. The Sicilian is a companion novel to The Godfather – it is set in the same universe during a similar time period. Omerta is another mafia-centered novel, and Fools Die explores comparable themes of power, corruption, and morality. Therefore, while these recommendations make sense from a content-similarity perspective, they could be criticized for lacking variability and serendipity. See Appendix M for implementation details for this section.

4. Challenges and Limitations

Despite achieving promising results, this project faced several challenges and limitations.

4.1 Interpretability of Recommendations

Many of the recommendation algorithms used (like LightFM for example) are implemented as pre-packaged functions in libraries. While convenient, these models often operate as "black boxes," making it difficult to interpret why a particular book was recommended to a user. With more time, we would have done some more research to make these implementations more explainable for us and for potential users.

4.2 Evaluation Constraints Due to Medium

In music or video recommendation systems, feedback can be gathered rather quickly. Listening to a song or watching a short video takes only a few minutes. Books, however, require a significant time investment. As a result, it was impractical within a single semester to gather meaningful, large-scale feedback on the quality of the recommendations we generated. This limited our ability to validate model effectiveness through user satisfaction with our friends.

While we initially tried to use typical metrics to evaluate our models, these quantitative measures don't fully capture the effectiveness of book recommendations. Such metrics assess whether a recommended book appears in a list of known user-rated items, but they don't reflect whether the user would actually enjoy or finish the book. In domains like literature, where preferences are deeply personal and context-dependent, user satisfaction can't be easily inferred from past ratings or similarity scores alone. This made it challenging to determine how successful our system truly was in promoting meaningful reading experiences, especially in the absence of qualitative or long-term feedback.

4.3 Data Sparsity and Incompleteness

The original Book-Crossing dataset contained a large number of missing values and sparse interactions. Many users rated only a handful of books, and many books had few or no ratings. While we addressed this by enriching metadata via the ISBN-DB API, the sparsity still affected the performance of collaborative models and limited the data available for new or niche books.

4.4 Cold-Start Problems

Although we tried to mitigate cold-start issues for new users, completely resolving this problem remains difficult. New users without prior ratings still pose a challenge, especially for models heavily reliant on interaction history. Likewise, books with highly unique subjects or limited metadata remain difficult to recommend effectively without the existing interactions.

5. Future Work

Several promising directions remain to be explored in order to enhance the performance, usability, and reliability of the recommender system developed in this project.

5.1 Incorporating Numerical Metadata for Content-Based Filtering

Currently, the content-based filtering pipeline primarily relies on textual data vectorized using TF-IDF. To incorporate numerical metadata such as the year of publication, future iterations will need to transform this data into a format compatible with vector-based similarity measures. One approach may involve binning years into categories (e.g., classic, contemporary, modern) or applying normalization followed by concatenation with TF-IDF features to reflect temporal trends in user preferences.

5.2 Development of a User-Interactive Interface

To transition from a prototype into a usable product, a user-facing web application should be developed. This would allow users to explore and interact with the recommendation engine in real-time, select filters, input preferences, and receive tailored recommendations outside of the current Colab-based environment.

5.3 Incorporating User Feedback for Adaptive Recommendations

User feedback is essential for refining the accuracy of recommendations. Many users may or may not rate books they haven't finished or select titles they later abandon. A future system should differentiate between intent to read and actual engagement, possibly using implicit feedback signals like time spent

reading, bookmark behavior, or completion status. Incorporating these dimensions could enable the model to better reflect user satisfaction.

5.4 Rating Quality and Validation

A major challenge in recommendation systems is determining whether available ratings genuinely reflect a reader's experience. Unlike platforms such as Amazon, where product reviews typically follow confirmed purchases, datasets like Book-Crossing offer ratings with no guarantee the user read or completed the book. This raises questions about the reliability of such data. Future work could explore validating rating quality by incorporating additional indicators such as review sentiment, user completion status, or engagement history. Distinguishing between surface-level ratings and those based on meaningful interaction could enhance the model's accuracy and trustworthiness.

5.5 Accounting for Nuance in User Ratings

Ratings do not always tell the full story. Users might dislike a book not because of the topic but due to poor execution. On the contrary, they may appreciate the concept even if the narrative style did not suit them. Future systems could aim to extract these nuances using natural language processing (NLP) on more detailed written reviews, allowing recommendations to account for both thematic preference and execution quality.

5.6 Accounting for Nuance Beyond Clean Data

Even after thorough data cleaning, important subtleties can be lost that significantly impact recommendation quality. Clean datasets often retain only structured, surface-level information (such as titles, authors, or average ratings), while discarding context-specific or subjective aspects of user experience. For example, two users may rate a book similarly but for completely different reasons: one may have enjoyed the storyline but disliked the pacing, while the other appreciated the author's style but found the plot weak. Future work should explore techniques for capturing these nuanced signals, potentially through natural language processing on user reviews, deeper metadata analysis, or user behavior modeling. Recognizing and integrating these layers of complexity could lead to more personalized and context-aware recommendations.

6. Conclusion

In this project, we designed and implemented a book recommendation system using the Book-Crossing dataset. We began by cleaning and enriching the dataset with external metadata from the ISBN-DB API, adding valuable features such as subjects and synopses. These enhancements allowed us to move beyond surface-level attributes like title and author and enable more sophisticated content-based filtering approach.

To ultimately build a hybrid recommendation system, we first created content-based and collaborative filtering methods. For content-based, we tested 15 feature combinations, which we found to be decently effective at identifying content-similar books, but lacked variability and serendipity. We then created 7 unique collaborative strategies, which we again found to work decently well, but were less explainable, as we were mainly using built-in Python libraries. With all these different strategies, we designed a final

scoring formula to rank recommendations, balancing content-based and collaborative input, as well as giving a boost to recommendations that contain both.

While there remain areas for future refinement, we believe that we designed a decent hybrid model that successfully balances content-based and collaborative filtering strategies. In an age of overwhelming content choice, we hope that systems like this can help readers more easily discover books tailored to their taste.

References

Aggarwal, C. C. (2016). *Recommender Systems: The textbook*. Springer.

Collections. Python Documentation. https://docs.python.org/3/library/collections.html

Cosine_similarity. scikit. https://scikit-learn.org/stable/modules/generated/sklearn.metrics.pairwise.cosine_similarity.html

Decision in normandy. Goodreads. https://www.goodreads.com/book/show/53514493

Doshi, S. (2019, February 20). *Brief on Recommender Systems*. Medium. https://medium.com/data-science/brief-on-recommender-systems-b86a1068a4dd

Dracula. Goodreads. https://www.goodreads.com/book/show/17245.Dracula

Felden, C., & Chamoni, P. (2007). Recommender Systems Based on an Active Data warehouse with Text Documents. *2007 40th Annual Hawaii International Conference on System Sciences (HICSS'07)*. https://doi.org/10.1109/hicss.2007.460

Felfernig, A., Boratto, L., Stettinger, M., & Tkalčič, M. (2018). *Group Recommender Systems: An introduction*. Springer.

Fools die. Goodreads. https://www.goodreads.com/book/show/22035.Fools_Die

Frankenstein: The 1818 text. Goodreads. https://www.goodreads.com/book/show/35031085-frankenstein

The godfather (the godfather, #1). Goodreads. https://www.goodreads.com/book/show/22034.The_Godfather

Hstack. hstack - SciPy v1.15.2 Manual. https://docs.scipy.org/doc/scipy/reference/generated/scipy.sparse.hstack.html

Hu, Y., Koren, Y., & Volinsky, C. (2008). Collaborative Filtering for Implicit Feedback Datasets. *2008 Eighth IEEE International Conference on Data Mining*. https://doi.org/10.1109/icdm.2008.22

ISBNdb API Documentation V2. ISBNdb. https://isbndb.com/apidocs/v2

Lyst. *LightFM Documentation*. Welcome to LightFM's documentation! - LightFM 1.16 documentation. https://making.lyst.com/lightfm/docs/home.html#:~:text=LightFM%20is%20a%20Python%20implementation,the%20traditional%20matrix%20factorization%20algorithms.

Nearestneighbors. scikit. https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.NearestNeighbors.html

Omerta (the godfather). Goodreads. https://www.goodreads.com/book/show/22028.Omerta

Pazos Arias, J. J., Fernández Vilas, A., & Dìaz Redondo, R. P. (2012). *Recommender Systems for the Social Web* (Vol. 32). Springer.

Prabowo, D. S. (2023, March 7). *Hybrid Recommendation System Using LightFM*. Medium. https://medium.com/@dikosaktiprabowo/hybrid-recommendation-system-using-lightfm-e10dd6b42923

Ricci, F., Rokach, L., Shapira, B., & Kantor, P. B. (2011). *Recommender Systems Handbook*. Springer.

Ronquillo, A. (2024, February 28). *Python's Requests Library (guide)*. Real Python. https://realpython.com/python-requests/

The Sicilian (the godfather #2). Goodreads. https://www.goodreads.com/book/show/22026.The_Sicilian

Surprise Documentation. Welcome to Surprise' documentation! - Surprise 1 documentation. https://surprise.readthedocs.io/en/stable/

Symeonidis, P., & Zioupos, A. (2016). *Matrix and Tensor Factorization Techniques for Recommender Systems*. Springer.

Symeonidis, P., Ntempos, D., & Manolopoulos, Y. (2014). *Recommender Systems for Location-based Social Networks*. Springer.

TfidfVectorizer. scikit. https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.TfidfVectorizer.html

TruncatedSVD. scikit. https://scikit-learn.org/stable/modules/generated/sklearn.decomposition.TruncatedSVD.html

Under the blood-red sun. Goodreads. https://www.goodreads.com/book/show/286469.Under_the_Blood_Red_Sun

Yildirim, M. (2024, November 30). *Building a Recommendation System with LightFM*. Medium. https://medium.com/@murattyldrm7/building-a-recommendation-system-with-lightfm-35394c8d90fb

Yoo, K.-H., Gretzel, U., & Zanker, M. (2013). *Persuasive Recommender Systems: Conceptual Background and Implications* . Springer.

Colab Links

Below, you will find the links towards our ipynb files on colab:

1. Data Enrichment Using the ISBN-DB API: https://colab.research.google.com/drive/1UhAJEFeUeYRU0GXZ7nZjScWzFq3au8-m#scrollTo=w7VrC2CJeE96
2. Overall Recommender System: https://colab.research.google.com/drive/1EJU96XtA43ODLRe0czGycJqdeWKTjw2F#scrollTo=jVXfw57LhMJZ

Appendix

Note: We included code that we deemed relevant to this report. To see all code, refer to section 9 (Colab Links)

Appendix A: Code used for API calls to enrich our dataset

```
import pandas as pd
import requests
import time
import os
import re

# --- CONFIGURATION ---
API_KEY = "59541_7b280637020d60a99abb89a246a1df18"
BASE_URL = "https://api2.isbndb.com/book/"
BATCH_SIZE = 10  # Since the API allows max 10 results per call
DAILY_LIMIT = 2000  # Academic plan limit
SAVE_FILE = "/content/isbn_enriched_data.csv"  # Where enriched data is saved
SKIPPED_FILE = "/content/skipped_isbns.csv"  # Log of invalid ISBNs

headers = {"Authorization": API_KEY}

# --- Load dataset ---
url = 'https://raw.githubusercontent.com/tamaraamicic/Honours-Project-Datasets/refs/heads/main/items_info_cleaned_2.dat'
items_info_cleaned_2_df = pd.read_csv(url, sep='\t', on_bad_lines='skip', index_col=False)

# Extract unique ISBNs
isbn_list = items_info_cleaned_2_df['ISBN'].astype(str).unique().tolist()

# --- Function to validate ISBNs ---
def is_valid_isbn(isbn):
    """Checks if an ISBN is valid (either 10 or 13 digits, allowing 'X' at the end for ISBN-10)."""
    return bool(re.match(r'^\d{9}[\dX]$|^\d{13}$', isbn))

# Filter valid and invalid ISBNs
valid_isbns = [isbn for isbn in isbn_list if is_valid_isbn(isbn)] # [:10] IF TRYING FOR FIRST 10
invalid_isbns = [isbn for isbn in isbn_list if not is_valid_isbn(isbn)]

# Save invalid ISBNs for reference
pd.DataFrame({'Invalid_ISBNs': invalid_isbns}).to_csv(SKIPPED_FILE, index=False)
```

```
print(f"Total valid ISBNs to process: {len(valid_isbns)}")
print(f"{len(invalid_isbns)} invalid ISBNs logged in {SKIPPED_FILE}.")

# --- Resume from saved progress ---
if os.path.exists(SAVE_FILE):
    enriched_df = pd.read_csv(SAVE_FILE)
    processed_isbns = set(enriched_df['isbn10'].astype(str)) | set(enriched_df['isbn13'].astype(str))
    print(f"Resuming from saved progress. {len(processed_isbns)} books already processed.")
else:
    enriched_df = pd.DataFrame(columns=['isbn10', 'isbn13', 'subjects', 'synopsis', 'language', 'pages'])
    processed_isbns = set()

# --- Function to fetch book data ---
def fetch_book_data(isbn):
    try:
        url = f"{BASE_URL}{isbn}"
        response = requests.get(url, headers=headers)

        if response.status_code == 200:
            data = response.json().get('book', {})
            return {
                'isbn10': data.get('isbn10', ''),
                'isbn13': data.get('isbn13', ''),
                'subjects': "|".join(data.get('subjects', [])),  # Convert list to string
                'synopsis': data.get('synopsis', ''),
                'language': data.get('language', ''),
                'pages': data.get('pages', '')
            }
        else:
            print(f"Error fetching {isbn}: {response.status_code} - {response.json()}")
            return None
    except Exception as e:
        print(f"Exception for {isbn}: {e}")
        return None

# --- Process in batches (Respecting API Limits) ---
total_calls = 0  # Track API calls

for i in range(0, len(valid_isbns), BATCH_SIZE):
    if total_calls >= DAILY_LIMIT:
        print("Daily API limit reached. Stopping for today.")
        break

    batch_isbns = valid_isbns[i:i + BATCH_SIZE]
    batch_results = []

    print(f"Processing batch {i // BATCH_SIZE + 1}/{len(valid_isbns) // BATCH_SIZE + 1}...")

    for isbn in batch_isbns:
        if isbn in processed_isbns:  # Skip already processed ISBNs
            continue

        book_info = fetch_book_data(isbn)
        if book_info:
            batch_results.append(book_info)
            processed_isbns.add(book_info['isbn10'])
```

```
            processed_isbns.add(book_info['isbn13'])
            total_calls += 1

        if total_calls >= DAILY_LIMIT:
            print("Daily API limit reached. Stopping for today.")
            break  # Stop processing if limit is reached

        time.sleep(1)  # Prevent hitting API rate limits

    # Convert batch results to DataFrame and append to CSV
    if batch_results:
        batch_df = pd.DataFrame(batch_results)
        enriched_df = pd.concat([enriched_df, batch_df], ignore_index=True)
        enriched_df.to_csv(SAVE_FILE, index=False)  # Save progress

    print(f"Batch {i // BATCH_SIZE + 1} completed. Progress saved.")

print("All valid ISBNs processed successfully or daily limit reached.")
```

Appendix B: Initial Model

```
items_info_cleaned_2_df["Features"] = (
    items_info_cleaned_2_df["Book-Title"] + " " +
    items_info_cleaned_2_df["Book-Author"] + " " +
    items_info_cleaned_2_df["Publisher"]
)
items_info_cleaned_2_df[["Book-Title", "Features"]].head()
```

```
from sklearn.feature_extraction.text import TfidfVectorizer

tfidf_vectorizer = TfidfVectorizer(stop_words="english") # initializing tf-idf vectorizer.
stop_words="english" removes common english stopwords
tfidf_matrix = tfidf_vectorizer.fit_transform(items_info_cleaned_2_df["Features"]) # applies TF-IDF (term
frequency/inverse document frequency) to convert the text into a matrix of numbers. each cell is the TF-
IDF score (how important that word is for that book)

# rows = books, columns = words
print(tfidf_matrix.shape)
```

```
# just to see what it looks like
tfidf_df = pd.DataFrame(tfidf_matrix.toarray(), columns=tfidf_vectorizer.get_feature_names_out())
tfidf_df.head(10)
```

```
from sklearn.metrics.pairwise import cosine_similarity

cosine_sim = cosine_similarity(tfidf_matrix)

print(cosine_sim.shape)
```

```
# just to see what it looks like
cosine_sim_df = pd.DataFrame(cosine_sim, index=items_info_cleaned_2_df.index,
columns=items_info_cleaned_2_df.index)
```

```
cosine_sim_df.iloc[:10, :10]
```

```
book_indices = pd.Series(items_info_cleaned_2_df.index, index=items_info_cleaned_2_df["Book-Title"]).drop_duplicates() # mapping book titles to their corresponding index in the dataframe
book_indices
```

```
def simple_content_based_filtering(title, num_recommendations):
    matches = items_info_cleaned_2_df[items_info_cleaned_2_df['Book-Title'] == title]

    if matches.empty:
        raise ValueError(f"'{title}' not found in the dataset.")

    if len(matches) > 1:
        print(f"Multiple entries found for '{title}'. Using the first one:")
        display(matches[['Book-Title', 'Book-Author', 'Publisher']].iloc[0])

    index = matches.index[0] # gets index of first match

    #index = book_indices[title] # get the given book
    sim_scores = list(enumerate(cosine_sim[index])) # similarity score between the given book and all the others
    sim_scores = sorted(sim_scores, key=lambda x: x[1], reverse=True)[1:num_recommendations+1] # get most similar. exclude itself
    book_indices_list = [i[0] for i in sim_scores] # get indices of recommended books

    return items_info_cleaned_2_df.iloc[book_indices_list][["Book-Title", "Book-Author", "Publisher"]] # get relevant book details for display

simple_content_based_filtering("Decision in Normandy", 5)
```

Appendix C: Enriched Model

```
import pandas as pd
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.neighbors import NearestNeighbors
from scipy.sparse import hstack
from sklearn.decomposition import TruncatedSVD

enriched_smaller_df = pd.read_csv('https://raw.githubusercontent.com/tamaraamicic/Honours-Project-Datasets/refs/heads/main/enriched_books_dataset_smaller.csv', sep=',', on_bad_lines='skip', index_col=False)

# list of textual features to weight and combine later
text_features = ['Book-Title', 'Book-Author', 'subjects', 'synopsis']

# for storage of all of the individual tfidf matrices
tfidf_matrices = {}
vectorizers = {}

# apply tfidf to each textual feature independently
for feature in text_features:
    tfidf = TfidfVectorizer(stop_words='english')
    tfidf_matrix = tfidf.fit_transform(enriched_smaller_df[feature].astype(str))
    tfidf_matrices[feature] = tfidf_matrix
    vectorizers[feature] = tfidf  # store in case needed later for unseen data
```

```
# different strategies to try out (defining weight combinations)
strategies = {
    "title": {'Book-Title': 1.0},
    "author": {'Book-Author': 1.0},
    "subjects": {'subjects': 1.0},
    "synopsis": {'synopsis': 1.0},
    "title_author": {'Book-Title': 0.5, 'Book-Author': 0.5},
    "title_subjects": {'Book-Title': 0.5, 'subjects': 0.5},
    "title_synopsis": {'Book-Title': 0.5, 'synopsis': 0.5},
    "author_subjects": {'Book-Author': 0.5, 'subjects': 0.5},
    "author_synopsis": {'Book-Author': 0.5, 'synopsis': 0.5},
    "subjects_synopsis": {'subjects': 0.5, 'synopsis': 0.5},
    "title_author_subject": {'Book-Title': 0.33, 'Book-Author': 0.33, 'subjects': 0.34},
    "title_author_synopsis": {'Book-Title': 0.33, 'Book-Author': 0.33, 'synopsis': 0.34},
    "title_subjects_synopsis": {'Book-Title': 0.33, 'subjects': 0.33, 'synopsis': 0.34},
    "author_subjects_synopsis": {'Book-Author': 0.33, 'subjects': 0.33, 'synopsis': 0.34},
    "title_author_subjects_synopsis": {'Book-Title': 0.25, 'Book-Author': 0.25, 'subjects': 0.25,
'synopsis': 0.25}
}

similarity_models = {}

# get all unique languages, so that we can build one model per strategy per language
languages = enriched_smaller_df['language'].dropna().unique() # there are 23 languages in the dataset. so
we create 15*23 = 345 models

for strategy_name, weights in strategies.items():
    for lang in languages:
        # subset the dataframe to books in that language
        lang_df = enriched_smaller_df[enriched_smaller_df['language'] == lang]
        if lang_df.empty:
            continue  # skip if no books in this language

        # combining tfidf matrices according to weight. essentially just concatenating them horizontally
        matrices_to_stack = []
        for feature, weight in weights.items():
            tfidf = vectorizers[feature]
            matrix = tfidf.transform(lang_df[feature].astype(str)) * weight
            matrices_to_stack.append(matrix)

        combined_matrix = hstack(matrices_to_stack)

        # fit nearest neighbors
        nn_model = NearestNeighbors(metric='cosine', algorithm='brute', n_neighbors=6) # get top 6, then
drop the first (the book itself, because it is perfectly similar to itself) to get top 5 books
        nn_model.fit(combined_matrix)

        # dictionary with keys strategy and language to save the model, combined matrix, and df to be used
later
        similarity_models[(strategy_name, lang)] = {
            "model": nn_model,
            "combined_matrix": combined_matrix,
            "df": lang_df.reset_index(drop=True)  # store index-matching df
        }
```

```
import re
```

```
def normalize_title(title):
    title = title.lower().strip()
    title = re.sub(r'\([^)]*\)', '', title) # remove text in parentheses
    title = re.sub(r'[^a-z0-9 ]+', '', title) # remove punctuation (anything that is not a letter a-z, a
digit 0-9, or a space)
    title = re.sub(r'\s+', ' ', title) # normalize whitespace (turns tabs, newlines, multiple spaces, etc.
into just one space)
    return title.strip()

def improved_content_filter(title, num_recommendations, strategy_name):
    try:
        # find the book and language
        original_row = enriched_smaller_df[enriched_smaller_df['Book-Title'] == title].iloc[0]
        target_language = original_row['language']
    except IndexError:
        raise ValueError(f"Book titled '{title}' not found in dataset.")

    # find the appropriate model to use
    key = (strategy_name, target_language)
    if key not in similarity_models:
        raise ValueError(f"No model found for strategy '{strategy_name}' and language
'{target_language}'.")

    model_info = similarity_models[key]
    model = model_info["model"]
    matrix = model_info["combined_matrix"]
    lang_df = model_info["df"]

    # get the index in the language-filtered DataFrame
    local_index = lang_df[lang_df['Book-Title'] == title].index[0]

    # get top N+1 neighbors and skip the book itself
    distances, indices = model.kneighbors(matrix[local_index].reshape(1, -1), n_neighbors=50 + 1)  #
larger pool
    top_indices = indices[0][1:]  # skip self

    recs = lang_df.iloc[top_indices][['Book-Title', 'Book-Author', 'Publisher']].copy()

    # filter out editions/duplicates of the base book
    base_norm = normalize_title(title)
    seen_titles = set()
    filtered_recs = []

    for _, row in recs.iterrows():
        rec_title = row['Book-Title']
        norm_rec = normalize_title(rec_title)

        if base_norm in norm_rec:
            continue
        if norm_rec in seen_titles:
            continue

        seen_titles.add(norm_rec)
        filtered_recs.append(row)

        if len(filtered_recs) == num_recommendations:
```

```
            break

    return pd.DataFrame(filtered_recs)
```

Appendix D: Baseline Collaborative Filtering

```
def predict_baseline(user_id, book_id):
    """
    Predict rating using Baseline CF: global mean + user bias + item bias.
    Returns np.nan if user or item is missing.
    """
    if book_id not in ratings_pivot.columns or user_id not in ratings_pivot.index:
        return np.nan

    # Global average rating (μ)
    global_mean = book_ratings_df['rating'].mean()

    # User bias (b_u)
    user_avg = ratings_pivot.loc[user_id].mean()
    user_bias = user_avg - global_mean

    # Item bias (b_i)
    item_avg = ratings_pivot[book_id].mean()
    item_bias = item_avg - global_mean

    # Predicted rating
    return global_mean + user_bias + item_bias

def recommend_books_baseline(user_id, n=10):
    """
    Recommend top-N books for a user based on Baseline CF.
    """
    unseen_books = get_unrated_books(user_id)
    predictions = []

    for book in unseen_books:
        pred = predict_baseline(user_id, book)
        if not np.isnan(pred):
            predictions.append((book, pred))

    predictions = sorted(predictions, key=lambda x: x[1], reverse=True)[:n]
    pred_dict = {book: rating for book, rating in predictions}

    # Load book metadata
    items_info_cleaned_2_df = pd.read_csv(
        'https://raw.githubusercontent.com/tamaraamicic/Honours-Project-
Datasets/refs/heads/main/items_info_cleaned_2.dat',
        sep='\t',
        on_bad_lines='skip',
        index_col=False
    )

    # Get recommended book details
    recommended_books =
items_info_cleaned_2_df[items_info_cleaned_2_df['Book_ID'].isin(pred_dict.keys())].copy()
    recommended_books['Predicted Rating'] = recommended_books['Book_ID'].map(pred_dict)
```

```
    return recommended_books[['Book-Title', 'Book-Author', 'Publisher', 'Predicted Rating']]

# Test it with a user
user_id = 1
recommended_baseline_books = recommend_books_baseline(user_id, n=10)
print(recommended_baseline_books)
```

Appendix E: User-Based Collaborative Filtering Using Cosine Similarity

```
import pandas as pd
from surprise import Dataset, Reader
from surprise.model_selection import train_test_split
from sklearn.metrics.pairwise import cosine_similarity
import numpy as np

book_ratings_df = pd.read_csv('https://raw.githubusercontent.com/tamaraamicic/Honours-Project-
Datasets/refs/heads/main/book_ratings.dat', sep='\t', on_bad_lines='skip', index_col=False)

user_item_matrix = book_ratings_df.pivot(index='user', columns='item', values='rating').fillna(0)

user_similarity = cosine_similarity(user_item_matrix)
user_similarity_df = pd.DataFrame(user_similarity, index=user_item_matrix.index,
columns=user_item_matrix.index)

def get_similar_users(user_id, n=5):
    similar_users = user_similarity_df[user_id].sort_values(ascending=False)[1:n+1]  # skips itself by
skipping the first value at index 0 (perfect similarity of 1)
    return similar_users.index.tolist()

def recommend_books_user_based(user_id, n=10):
    similar_users = get_similar_users(user_id, n=10)  # find top 10 similar users according to similarity
score

    similar_users_books = book_ratings_df[book_ratings_df['user'].isin(similar_users)] # books rated by
similar users
    rated_books = set(book_ratings_df[book_ratings_df['user'] == user_id]['item'])  # books rated by
target user
    unseen_books = similar_users_books[~similar_users_books['item'].isin(rated_books)]  # books rated by
similar users NOT seen by target user

    recommended_books = unseen_books.groupby('item')['rating'].mean().sort_values(ascending=False).head(n)
# lists top n recommendations, printing the average rating (of ratings from the similar users)

    # convert to dictionary with book metadata
    pred_dict = recommended_books.to_dict()
    top_book_ids = recommended_books.index.tolist()
    items_info_cleaned_2_df = pd.read_csv('https://raw.githubusercontent.com/tamaraamicic/Honours-Project-
Datasets/refs/heads/main/items_info_cleaned_2.dat', sep='\t', on_bad_lines='skip', index_col=False)
    recommended_books_df =
items_info_cleaned_2_df[items_info_cleaned_2_df['Book_ID'].isin(pred_dict.keys())].copy()
    recommended_books_df['Predicted Rating'] = recommended_books_df['Book_ID'].map(pred_dict)

    # sorting
    recommended_books_df['SortOrder'] = recommended_books_df['Book_ID'].apply(lambda x:
top_book_ids.index(x))
    recommended_books_df = recommended_books_df.sort_values('SortOrder').drop(columns='SortOrder')
```

```
    return recommended_books_df[['Book-Title', 'Book-Author', 'Publisher', 'Predicted Rating']]

user_id = 1
recommended_books = recommend_books_user_based(user_id)
recommended_books
```

Appendix F: SVD-Based Latent Factor Model for Collaborative Filtering

```
import pandas as pd
from surprise import SVD, Dataset, Reader
from surprise.model_selection import train_test_split
from surprise import accuracy

# https://medium.com/@mariomg85/leveraging-surprise-library-for-recommender-systems-in-python-915e699f7a99
# https://surprise.readthedocs.io/en/stable/reader.html

book_ratings_df = pd.read_csv('https://raw.githubusercontent.com/tamaraamicic/Honours-Project-
Datasets/refs/heads/main/book_ratings.dat', sep='\t', on_bad_lines='skip', index_col=False)

reader = Reader(rating_scale=(1, 10)) # defining that the rating scale is from 1-10
data = Dataset.load_from_df(book_ratings_df[['user', 'item', 'rating']], reader) # format for Surprise
trainset, testset = train_test_split(data, test_size=0.2, random_state=42) # 80% train, 20% test

svd_model = SVD(n_factors=50, random_state=42) # training. we define 50 latent factors -- factors that are
hidden, not directly observed. for example genres, writing styles, book popularity, but we don't
explicitly define them. instead, SVD discovers these factors automatically
                                                # i tried changing this around though and it didn't change
the value much...
svd_model.fit(trainset)
predictions = svd_model.test(testset) # using trained SVD model to make
rmse = accuracy.rmse(predictions) # root MSE is preferred over MSE in recommender systems
```

```
items_info_cleaned_2_df = pd.read_csv('https://raw.githubusercontent.com/tamaraamicic/Honours-Project-
Datasets/refs/heads/main/items_info_cleaned_2.dat', sep='\t', on_bad_lines='skip', index_col=False,)

all_books = set(items_info_cleaned_2_df['Book_ID'])

def get_user_rated_books(user_id): # books that a user has rated
    return set(book_ratings_df[book_ratings_df['user'] == user_id]['item'])

def get_unrated_books(user_id): # books that a user hasn't rated
    rated_books = get_user_rated_books(user_id)
    return list(all_books - rated_books)

def predict_ratings_for_user(user_id): # predicts user rating for all books that the user hasn't rated
    unseen_books = get_unrated_books(user_id)
    predictions = [(book, svd_model.predict(user_id, book).est) for book in unseen_books]
    return sorted(predictions, key=lambda x: x[1], reverse=True)  # sorts books from highest to lowest
rating

# Function to recommend top-N books
def recommend_books_SVD(user_id, n=10):
    top_books = predict_ratings_for_user(user_id)[:n]  # gets top N books

    book_ids = []
    predicted_scores = {}
```

```
    # to get the predicted rating
    for book_id, predicted_rating in top_books:
        book_ids.append(book_id)
        predicted_scores[book_id] = predicted_rating

    recommended_books_SVD =
items_info_cleaned_2_df[items_info_cleaned_2_df['Book_ID'].isin(book_ids)].copy()
    recommended_books_SVD.loc[:, 'Predicted Rating'] =
recommended_books_SVD['Book_ID'].map(predicted_scores) # adding the predicted rating

    return recommended_books_SVD[['Book-Title', 'Book-Author', 'Publisher', 'Predicted Rating']]


user_id = 1
recommended_books_SVD = recommend_books_SVD(user_id)
recommended_books_SVD
```

Appendix G: KNN-Based Collaborative Filtering

```
import pandas as pd
import numpy as np
from sklearn.metrics.pairwise import cosine_similarity

# -------------------------------
# 1. Load the Ratings Data
# -------------------------------
book_ratings_df = pd.read_csv(
    'https://raw.githubusercontent.com/tamaraamicic/Honours-Project-
Datasets/refs/heads/main/book_ratings.dat',
    sep='\t',
    on_bad_lines='skip',
    index_col=False
)

# -------------------------------
# 2. Build the User-Item Ratings Matrix
# -------------------------------
# Pivot the ratings table so that rows are users and columns are items (books)
ratings_pivot = book_ratings_df.pivot(index='user', columns='item', values='rating')
# For computing cosine similarity, fill missing values with 0.
ratings_filled = ratings_pivot.fillna(0)

# -------------------------------
# 3. Compute User Similarity Matrix
# -------------------------------
# Use cosine similarity from scikit-learn
user_similarity = cosine_similarity(ratings_filled)
# Create a DataFrame for easier lookup; its rows/columns are the user IDs
user_similarity_df = pd.DataFrame(user_similarity, index=ratings_filled.index,
columns=ratings_filled.index)

# Create a mapping from user ID to index
user_id_to_index = {user: idx for idx, user in enumerate(ratings_filled.index)}

# -------------------------------
# 4. Define Prediction Functions
# -------------------------------
```

```python
def predict_rating(user_id, book_id, k=5):
    """
    Predict rating for a given user and book using weighted average of top k similar users.
    """
    # If the book is not in the ratings pivot, we cannot predict (return NaN)
    if book_id not in ratings_pivot.columns:
        return np.nan

    # Get the target user's row index and similarity scores
    user_idx = user_id  # The index here is the user id itself from the pivot table
    # Get similarities between the target user and all other users (excluding self by setting to 0)
    sims = user_similarity_df.loc[user_id].copy()
    sims[user_id] = 0.0  # zero-out self-similarity

    # Consider only users that rated the book (i.e. non-zero rating in the original pivot)
    # ratings_filled for the given book returns a Series with index = user id
    book_ratings = ratings_filled[book_id]
    rated_mask = book_ratings > 0
    # If no other user has rated the book, return NaN (or you could use the user's average)
    if rated_mask.sum() == 0:
        return np.nan

    # Filter the similarities and corresponding ratings to those users who rated the book
    sims = sims[rated_mask]
    neighbor_ratings = book_ratings[rated_mask]

    # If k is specified, pick the top k similar users
    if k is not None and len(sims) > k:
        top_k_idx = sims.nlargest(k).index
        sims = sims.loc[top_k_idx]
        neighbor_ratings = neighbor_ratings.loc[top_k_idx]

    # Avoid division by zero; if sum of similarities is 0, return the neighbor average
    sim_sum = sims.sum()
    if sim_sum == 0:
        return neighbor_ratings.mean()
    predicted = np.dot(sims, neighbor_ratings) / sim_sum
    return predicted


def get_user_rated_books(user_id):
    """
    Return a set of book IDs that the user has rated.
    """
    return set(book_ratings_df[book_ratings_df['user'] == user_id]['item'])


def get_unrated_books(user_id):
    """
    Return a list of book IDs that the user hasn't rated.
    """
    # Load the complete set of book IDs from the items info dataset
    items_info_cleaned_2_df = pd.read_csv(
        'https://raw.githubusercontent.com/tamaraamicic/Honours-Project-
Datasets/refs/heads/main/items_info_cleaned_2.dat',
        sep='\t',
        on_bad_lines='skip',
        index_col=False
```

```
    )
    all_books = set(items_info_cleaned_2_df['Book_ID'])
    rated_books = get_user_rated_books(user_id)
    return list(all_books - rated_books)


def predict_ratings_for_user(user_id, k=5):
    """
    Predict the ratings for all unseen books for a given user.
    """
    unseen_books = get_unrated_books(user_id)
    predictions = []
    for book in unseen_books:
        pred = predict_rating(user_id, book, k)
        # Only include predictions if we have a value (i.e., at least one similar neighbor rated the book)
        if not np.isnan(pred):
            predictions.append((book, pred))
    # Sort predictions from highest to lowest predicted rating
    return sorted(predictions, key=lambda x: x[1], reverse=True)


def recommend_books_KNN_custom(user_id, n=10, k=5):
    """
    Recommend top-N books for a given user using the KNN-based CF.
    """
    # Predict ratings for unseen books
    preds = predict_ratings_for_user(user_id, k=k)
    top_preds = preds[:n]

    # Build a dictionary of book: predicted rating
    pred_dict = {book: rating for book, rating in top_preds}

    # Load the items info dataset to get book details
    items_info_cleaned_2_df = pd.read_csv(
        'https://raw.githubusercontent.com/tamaraamicic/Honours-Project-
Datasets/refs/heads/main/items_info_cleaned_2.dat',
        sep='\t',
        on_bad_lines='skip',
        index_col=False
    )

    # Filter items dataset for the recommended books and map the predicted ratings
    recommended_books =
items_info_cleaned_2_df[items_info_cleaned_2_df['Book_ID'].isin(list(pred_dict.keys()))].copy()
    recommended_books.loc[:, 'Predicted Rating'] = recommended_books['Book_ID'].map(pred_dict)

    return recommended_books[['Book-Title', 'Book-Author', 'Publisher', 'Predicted Rating']]

# -------------------------------
# 5. Get Recommendations for a Specific User
# -------------------------------
user_id = 1
recommended_books_custom = recommend_books_KNN_custom(user_id, n=10, k=5)
print(recommended_books_custom)
```

Appendix H: LightFM with Book Ratings

```
import pandas as pd
import numpy as np
from lightfm import LightFM
from lightfm.data import Dataset
from lightfm.evaluation import precision_at_k

book_ratings_df = pd.read_csv('https://raw.githubusercontent.com/tamaraamicic/Honours-Project-Datasets/refs/heads/main/book_ratings.dat', sep='\t', on_bad_lines='skip', index_col=False)
enriched_smaller_df = pd.read_csv('https://raw.githubusercontent.com/tamaraamicic/Honours-Project-Datasets/refs/heads/main/enriched_books_dataset_smaller.csv', sep=',', on_bad_lines='skip', index_col=False)

# only keep ratings for books that exist in the enriched dataset
valid_books = set(enriched_smaller_df['Book_ID'])
book_ratings_df = book_ratings_df[book_ratings_df['item'].isin(valid_books)].copy()

# converting ratings into binary interactions (1 = liked, 0 = not liked) because LightFM works best with implicit feedback (interaction vs. no interaction)
book_ratings_df['rating'] = book_ratings_df['rating'].apply(lambda x: 1 if x > 5 else 0)

lightfm_ratings_dataset = Dataset() # LightFM dataset object

# creating list of words (that comprise of the book title, author, subjects, and synopsis)
# LightFM will learn that books with similar words in their title, author, subjects, or synopsis are related
enriched_smaller_df['item_features'] = (
    enriched_smaller_df['Book-Title'].astype(str) + " " +
    enriched_smaller_df['Book-Author'].astype(str) + " " +
    enriched_smaller_df['subjects'].astype(str) + " " +
    enriched_smaller_df['synopsis'].astype(str)
).apply(lambda x: x.split())  # split into list of words

# Ensure we pass the same features to fit() and build_item_features()
unique_features = set(enriched_smaller_df['item_features'].explode().unique()) # getting all the unique words among all book titles, authors, subject, and synposis

# fits/registersr dataset with users, items, and item features. this creates internal mappings so that LightFM can process these entities correctly later
lightfm_ratings_dataset.fit(
    users=book_ratings_df['user'].unique(),
    items=enriched_smaller_df['Book_ID'].unique(), # using all books (not just rated ones) so that books that have not been rated could also be recommended
    item_features=unique_features
)

# building a sparse user-book interaction matrix internally
# rows = users
# columns = books
# values = 1 (liked) or 0 (not liked)
(lightfm_ratings_interactions, lightfm_ratings_weights) = lightfm_ratings_dataset.build_interactions([
    (row['user'], row['item'], row['rating'])
    for _, row in book_ratings_df.iterrows()
])

# internally creates a sparse item feature matrix
```

```
# rows = book ids
# columns = features
# values = 1 if the book has the feature, 0 if not
# but instead of storing a huge matrix with mostly zeros, LightFM only stores non-zero values for
efficiency
lightfm_ratings_item_features = lightfm_ratings_dataset.build_item_features([
    (row['Book_ID'], row['item_features'])
    for _, row in enriched_smaller_df.iterrows()
])
```

```
# Train a hybrid model (both collaborative & content-based filtering)
lightfm_ratings_model = LightFM(loss='warp') # weighted approximate-rank pairwise loss function
                            # optimizes for ranking quality instead of prediction accuracy
                            # instead of predicting a rating (like 1-10), WARP ranks books based on
likelihood of interest.
                            # works better for implicit feedback
                            # prioritizes learning what books should be ranked higher for each user

# learns patterns from both user ratings & book metadata.
lightfm_ratings_model.fit(lightfm_ratings_interactions, item_features=lightfm_ratings_item_features,
epochs=10, num_threads=4) # more epochs = better learning, but too many can cause overfitting
                                                                                # 4 cpu threads to speed up
training
```

```
# we do hybrid, even though this is one of our collaborative methods, bc the book content helps the
LightFM model work better (for cold start).
# the model is still largely collaborative
def recommend_books_hybrid_df(model, dataset, item_features_to_use, user_id, n=10):
    item_ids = np.arange(len(dataset.mapping()[2])) # get all book indices

    scores = model.predict(user_id, item_ids, item_features=item_features_to_use) # predicting scores

    top_items = np.argsort(-scores)[:n*3]  # sorts books highest to lowest, gets top 3*n (get more than
needed, in case we remove duplicates later)

    # LightFM internally uses numeric indices for books. this converts the internal book indices back to
original book ids
    item_map = {v: k for k, v in dataset.mapping()[2].items()} # dataset.mapping()[2] is item (book)
mapping
    recommended_book_ids = [item_map[i] for i in top_items]

    # getting book details from enriched_smaller_df
    recommended_books_df =
enriched_smaller_df[enriched_smaller_df['Book_ID'].isin(recommended_book_ids)].copy()

    # add predicted ratings
    predicted_scores = {item_map[i]: scores[i] for i in top_items}
    recommended_books_df.loc[:, 'Predicted Rating'] =
recommended_books_df['Book_ID'].map(predicted_scores)

    recommended_books_df = recommended_books_df.sort_values('Predicted Rating', ascending=False)

    # drop duplicates and get top n
    recommended_books_df = recommended_books_df.drop_duplicates(subset=['Book-Title', 'Book-Author'])
    recommended_books_df = recommended_books_df.head(n)
```

```
    return recommended_books_df[['Book-Title', 'Book-Author', 'Publisher', 'Predicted Rating']]

user_id = 1
recommended_books_df = recommend_books_hybrid_df(lightfm_ratings_model, lightfm_ratings_dataset,
lightfm_ratings_item_features, user_id)
recommended_books_df
```

Appendix I: LightFM with Book History

```
import pandas as pd
import numpy as np
from lightfm import LightFM
from lightfm.data import Dataset

# Load datasets
book_history_df = pd.read_csv('https://raw.githubusercontent.com/tamaraamicic/Honours-Project-
Datasets/refs/heads/main/book_history.dat', sep='\t', on_bad_lines='skip', index_col=False)
enriched_smaller_df = pd.read_csv('https://raw.githubusercontent.com/tamaraamicic/Honours-Project-
Datasets/refs/heads/main/enriched_books_dataset_smaller.csv', sep=',', on_bad_lines='skip',
index_col=False)

# only keep ratings for books that exist in the enriched dataset
valid_books = set(enriched_smaller_df['Book_ID'])
book_history_df = book_history_df[book_history_df['item'].isin(valid_books)].copy()


# creating list of words (that comprise of the book title, author, subjects and synopsis)
# LightFM will learn that books with similar words in their title, author, subjects or synopsis are
related
# could extend this when we combine with the new dataset
enriched_smaller_df['item_features'] = (
    enriched_smaller_df['Book-Title'].astype(str) + " " +
    enriched_smaller_df['Book-Author'].astype(str) + " " +
    enriched_smaller_df['subjects'].astype(str) + " " +
    enriched_smaller_df['synopsis'].astype(str)
).apply(lambda x: x.split())

# Ensure we pass the same features to fit() and build_item_features()
unique_features = set(enriched_smaller_df['item_features'].explode().unique()) # getting all the unique
words among all book titles and authors

lightfm_history_dataset = Dataset() # LightFM dataset object

# fits/registersr dataset with users, items, and item features. this creates internal mappings so that
LightFM can process these entities correctly later
lightfm_history_dataset.fit(
    users=book_history_df['user'].unique(),
    items=book_history_df['item'].unique(),
    item_features=unique_features
)

# building a sparse user-book interaction matrix internally
# rows = users
# columns = books
# values = 1 (liked)
(lightfm_history_interactions, lightfm_history_weights) = lightfm_history_dataset.build_interactions([
```

```
    (row['user'], row['item'], 1) for _, row in book_history_df.iterrows() # always 1 since all
interactions are access events
])

# internally creates a sparse item feature matrix
# rows = book ids
# columns = features
# values = 1 if the book has the feature, 0 if not
# but instead of storing a huge matrix with mostly zeros, LightFM only stores non-zero values for
efficiency
lightfm_history_item_features = lightfm_history_dataset.build_item_features([
    (row['Book_ID'], row['item_features']) for _, row in enriched_smaller_df.iterrows()
])
```

```
# Train a hybrid model (both collaborative & content-based filtering)
lightfm_history_model = LightFM(loss='warp') # weighted approximate-rank pairwise loss function
                            # optimizes for ranking quality instead of prediction accuracy
                            # instead of predicting a rating (like 1-10), WARP ranks books based on
likelihood of interest.
                            # works better for implicit feedback
                            # prioritizes learning what books should be ranked higher for each user

# learns patterns from both user ratings & book metadata.
lightfm_history_model.fit(lightfm_history_interactions, item_features=lightfm_history_item_features,
epochs=10, num_threads=4) # more epochs = better learning, but too many can cause overfitting
                                                                                # 4 cpu threads to speed up
training
```

```
user_id = 1
recommended_books_df = recommend_books_hybrid_df(lightfm_history_model, lightfm_history_dataset,
lightfm_history_item_features, user_id) # reusing the same function as before but calling it with the new
model (lightfm_history_model)
recommended_books_df
```

Appendix J: Cold-Start

```
import pandas as pd
import numpy as np

# ------------------------------
# 1. Load and Preprocess the User Info Data
# ------------------------------
user_info_url = 'https://raw.githubusercontent.com/tamaraamicic/Honours-Project-
Datasets/refs/heads/main/users_info.dat'
user_info_df = pd.read_csv(user_info_url, sep='\t', on_bad_lines='skip', index_col=False)

# Standardize location strings to lower-case and remove any leading/trailing whitespace
user_info_df['Location'] = user_info_df['Location'].str.lower().str.strip()
# Ensure Age is numeric (if necessary, convert or handle missing ages)
user_info_df['Age'] = pd.to_numeric(user_info_df['Age'], errors='coerce')

# ------------------------------
# 2. Define the Cold-Start Recommendation Function
# ------------------------------
def recommend_books_cold_start(new_age, new_location, n=10, age_threshold=5):
```

```
    """
    Recommend books for a new user based solely on demographic data (Age and Location).

    Parameters:
      new_age (int): Age of the new user.
      new_location (str): Location string of the new user (e.g., "minneapolis, minnesota, usa").
      n (int): Number of recommendations desired.
      age_threshold (int): Acceptable difference in age for considering users as similar.

    Returns:
      DataFrame: A dataframe containing the top-N recommended books with their title, author, publisher,
                 and the average rating from similar users.
    """
    # Convert new user's location to lower-case for matching
    new_location_lower = new_location.lower().strip()

    # Filter users within the age threshold
    similar_users = user_info_df[ (user_info_df['Age'].notnull()) &
                                  (abs(user_info_df['Age'] - new_age) <= age_threshold) ]

    # Further filter by location - here we keep users whose Location string contains the new user's
location.
    # This is a simple heuristic; we might use more sophisticated location matching.
    similar_users = similar_users[ similar_users['Location'].str.contains(new_location_lower, na=False) ]

    # If no users match location criteria, relax the condition to use age only.
    if similar_users.empty:
        similar_users = user_info_df[ (user_info_df['Age'].notnull()) &
                                      (abs(user_info_df['Age'] - new_age) <= age_threshold) ]

    if similar_users.empty:
        print("No similar users found based on the provided demographics.")
        return pd.DataFrame()  # or you could return overall popular books

    similar_user_ids = similar_users['User-ID'].unique()

    # -------------------------------
    # 3. Load the Book Ratings Data and Filter by Similar Users
    # -------------------------------
    book_ratings_url = 'https://raw.githubusercontent.com/tamaraamicic/Honours-Project-
Datasets/refs/heads/main/book_ratings.dat'
    book_ratings_df = pd.read_csv(book_ratings_url, sep='\t', on_bad_lines='skip', index_col=False)

    # Filter ratings to those from similar users only.
    similar_ratings = book_ratings_df[ book_ratings_df['user'].isin(similar_user_ids) ]

    # In case there are no ratings from similar users, fall back on overall averages.
    if similar_ratings.empty:
        print("No ratings found from similar users. Falling back to overall average ratings.")
        similar_ratings = book_ratings_df.copy()

    # -------------------------------
    # 4. Aggregate Ratings and Get Top Books
    # -------------------------------
    # Compute the average rating per book among similar users
    book_avg_ratings = similar_ratings.groupby('item')['rating'].mean().reset_index()
```

```
    # Sort by the average rating (highest first) and pick top-N recommendations
    top_books = book_avg_ratings.sort_values('rating', ascending=False).head(n)

    # -------------------------------
    # 5. Join with Book Metadata
    # -------------------------------
    items_info_url = 'https://raw.githubusercontent.com/tamaraamicic/Honours-Project-
Datasets/refs/heads/main/items_info_cleaned_2.dat'
    items_info_df = pd.read_csv(items_info_url, sep='\t', on_bad_lines='skip', index_col=False)

    # Merge to get details like Book Title, Book Author, and Publisher for when we print it out
    recommendations = items_info_df.merge(top_books, left_on='Book_ID', right_on='item')
    recommendations = recommendations.rename(columns={'rating': 'Average Rating'})

    return recommendations[['Book-Title', 'Book-Author', 'Publisher', 'Average Rating']]

# -------------------------------
# 6. Example Usage of the Cold-Start Function
# -------------------------------
# For a new user (with no prior ratings), provide age and location.
new_user_age = 25
new_user_location = "minneapolis, minnesota, usa"

recommended_books_cold_start = recommend_books_cold_start(new_user_age, new_user_location, n=10,
age_threshold=5)
print(recommended_books_cold_start)
```

Appendix K: Hybrid Recommender – Book-Based

```
def compare_strategies_binary_matrix(title, num_recommendations, strategy_names):
    from collections import defaultdict

    # dictionary mapping book titles to the strategies that recommended them
    book_to_strategies = defaultdict(set)

    for strategy in strategy_names:
        recs = improved_content_filter(title, num_recommendations, strategy)
        for book in recs['Book-Title']:
            book_to_strategies[book.strip().lower()].add(strategy)

    # get all unique book titles
    unique_books = list(book_to_strategies.keys())

    # create binary matrix
    binary_data = []
    total_strategies = len(strategy_names)

    for book in unique_books:
        row = {
            "Book-Title": book.title(),
        }
        count = 0
        for strategy in strategy_names:
            is_rec = int(strategy in book_to_strategies[book])
            row[strategy] = is_rec
            count += is_rec
```

```
        row["# Content-Based Strategies"] = count
        row["% Content-Based Strategies"] = round((count / total_strategies) * 100, 2)
        binary_data.append(row)

    return pd.DataFrame(binary_data).sort_values("# Content-Based Strategies", ascending=False)

strategy_names = [
    "title",
    "author",
    "subjects",
    "synopsis",
    "title_author",
    "title_subjects",
    "title_synopsis",
    "author_subjects",
    "author_synopsis",
    "subjects_synopsis",
    "title_author_subject",
    "title_author_synopsis",
    "title_subjects_synopsis",
    "author_subjects_synopsis",
    "title_author_subjects_synopsis"
]
comparison_binary_df = compare_strategies_binary_matrix("Decision in Normandy", 5, strategy_names)
comparison_binary_df
```

Appendix L: Hybrid Recommender – User-Based

```
from collections import defaultdict
import pandas as pd

book_ratings_df = pd.read_csv('https://raw.githubusercontent.com/tamaraamicic/Honours-Project-Datasets/refs/heads/main/book_ratings.dat', sep='\t', on_bad_lines='skip', index_col=False)

def compare_user_top_books_binary_matrix(user_id, num_top_books=3, num_recommendations=10):
    content_strategy_names = [
        "title",
        "author",
        "subjects",
        "synopsis",
        "title_author",
        "title_subjects",
        "title_synopsis",
        "author_subjects",
        "author_synopsis",
        "subjects_synopsis",
        "title_author_subject",
        "title_author_synopsis",
        "title_subjects_synopsis",
        "author_subjects_synopsis",
        "title_author_subjects_synopsis"
    ]

    # Here, Age-Location uses the cold-start implementation
    collaborative_strategy_names = ["SVD", "KNN", "BASELINE", "AGE-LOCATION", "LIGHTFM-RATINGS", "LIGHTFM-HISTORY", "SIMILARITY"]  # Add more here and add "SVD" back here
```

```
# get user's top N rated books
top_rated_books = (
    book_ratings_df[book_ratings_df['user'] == user_id]
    .sort_values(by='rating', ascending=False)
    .head(num_top_books)
)

# merge with enriched dataset to get book titles
top_books_info = top_rated_books.merge(
    enriched_smaller_df,
    left_on='item',
    right_on='Book_ID',
    how='inner'
)[['Book-Title']].drop_duplicates().reset_index(drop=True)

# run recommendations and track which strategies recommend each book
content_based_book_to_strategies = defaultdict(set)

for book_title in top_books_info['Book-Title']:
    for strategy in content_strategy_names:
        try:
            recs = improved_content_filter(book_title, num_recommendations, strategy)
            for rec_book in recs['Book-Title']:
                content_based_book_to_strategies[rec_book.strip().lower()].add((book_title, strategy))
        except Exception as e:
            print(f"Error for book '{book_title}' with strategy '{strategy}': {e}")


collaborative_filtering_book_to_strategies = defaultdict(set)

# SVD
try:
    svd_recs_df = recommend_books_SVD(user_id, num_recommendations)
    for title in svd_recs_df['Book-Title']:
        cleaned = title.strip().lower()
        collaborative_filtering_book_to_strategies[cleaned].add("SVD")
except Exception as e:
    print(f"Error generating SVD recommendations: {e}")

# KNN
try:
    knn_recs_df = recommend_books_KNN_custom(user_id, num_recommendations, k = 5)
    for title in knn_recs_df['Book-Title']:
        cleaned = title.strip().lower()
        collaborative_filtering_book_to_strategies[cleaned].add("KNN")
except Exception as e:
    print(f"Error generating KNN recommendations: {e}")

# Baseline
try:
    base_recs_df = recommend_books_baseline(user_id, num_recommendations)
    for title in base_recs_df['Book-Title']:
        cleaned = title.strip().lower()
        collaborative_filtering_book_to_strategies[cleaned].add("BASELINE")
except Exception as e:
    print(f"Error generating Baseline recommendations: {e}")

# LightFM (based on ratings)
```

```
    try:
        lightfm_recs_df = recommend_books_hybrid_df(lightfm_ratings_model, lightfm_ratings_dataset,
lightfm_ratings_item_features, user_id, n=num_recommendations)
        for title in lightfm_recs_df['Book-Title']:
            cleaned = title.strip().lower()
            collaborative_filtering_book_to_strategies[cleaned].add("LIGHTFM-RATINGS")
    except Exception as e:
        print(f"Error generating LightFM (ratings) recommendations: {e}")

    # LightFM (based on access history)
    try:
        lightfm_access_recs_df = recommend_books_hybrid_df(lightfm_history_model, lightfm_history_dataset,
lightfm_history_item_features, user_id, n=num_recommendations)
        for title in lightfm_access_recs_df['Book-Title']:
            cleaned = title.strip().lower()
            collaborative_filtering_book_to_strategies[cleaned].add("LIGHTFM-HISTORY")
    except Exception as e:
        print(f"Error generating LightFM (history) recommendations: {e}")

    # Similarity
    try:
        #similar_recs_df = get_similar_users(user_id, n=num_recommendations)
        similar_recs_df = recommend_books_user_based(user_id, n=num_recommendations)

        for title in similar_recs_df['Book-Title']:
            cleaned = title.strip().lower()
            collaborative_filtering_book_to_strategies[cleaned].add("SIMILARITY")
    except Exception as e:
        print(f"Error generating Similarity recommendations: {e}")

    # Cold-Start
    # ADD COLD START for the novelty component?? recommended_books_cold_start =
recommend_books_cold_start(new_user_age, new_user_location, n=5, age_threshold=5) - would need to add the
users_info to this??
    # recommend_books_cold_start(new_age, new_location, n=5, age_threshold=5)
    try:
        new_user_age, new_user_location = get_user_demographics(user_id)
        cold_start_recs_df = recommend_books_cold_start(new_user_age, new_user_location,
n=num_recommendations, age_threshold=5)
        for title in cold_start_recs_df['Book-Title']:
            cleaned = title.strip().lower()
            collaborative_filtering_book_to_strategies[cleaned].add("AGE-LOCATION")
    except Exception as e:
        print(f"Error generating Cold-Start recommendations: {e}")

    total_combinations = len(top_books_info) * len(content_strategy_names)

    # finds the maximum number of collaborative filtering strategies that recommend any single book
    max_collaborative_overlap = max((len(methods) for methods in
collaborative_filtering_book_to_strategies.values()), default=1)

    binary_data = []

    all_books_set = set(content_based_book_to_strategies.keys()) |
set(collaborative_filtering_book_to_strategies.keys())
    unique_books = list(all_books_set)

    for book in unique_books:
```

```
        row = {"Book-Title": book.title()}
        count_content = 0
        for base_book in top_books_info['Book-Title']:
            for strategy in content_strategy_names:
                key = (base_book, strategy)
                is_rec = int(key in content_based_book_to_strategies[book])
                row[f"{base_book} | {strategy}"] = is_rec
                count_content += is_rec

        count_collaborative = 0
        for strategy in collaborative_strategy_names:
            is_rec = int(strategy in collaborative_filtering_book_to_strategies.get(book, []))
            row[strategy] = is_rec
            count_collaborative += is_rec

        row["# Content-Based Strategies"] = count_content
        row["% Content-Based Strategies"] = round((count_content / total_combinations) * 100, 2)

        row["# Collaborative Filtering Strategies"] = count_collaborative
        row["% Collaborative Filtering Strategies"] = round((count_collaborative /
len(collaborative_strategy_names)) * 100, 2)

        total_count = count_content + count_collaborative
        total_possible = total_combinations + len(collaborative_strategy_names)

        row["# Total Filtering Strategies"] = total_count
        row["% Total Filtering Strategies"] = round((total_count / total_possible) * 100, 2)

        content_score = (count_content / total_combinations) * 0.425
        collaborative_score = (count_collaborative / max_collaborative_overlap) * 0.425 # normalized
collaborative part, because overlap is more rare than for content. so if we see a book appear in 3/7 of
the methods, that is very good. so we count this as 3/3 for the collaborative component. so for final
score it would contribute 3/3*0.425 = 0.425
        hybrid_bonus = 0.15 if (count_content > 0 and count_collaborative > 0) else 0 # add 0.2 to the
score if the book if found by content-based and collaborative methods
        row["Final Score"] = round(content_score + collaborative_score + hybrid_bonus, 4)

        binary_data.append(row)

    return pd.DataFrame(binary_data).sort_values("Final Score", ascending=False)
```

```
from IPython.display import display, HTML

user_comparison_df = compare_user_top_books_binary_matrix(user_id=1, num_top_books=3,
num_recommendations=10)
display(HTML(user_comparison_df.to_html(notebook=True)))
```

Appendix M: Recommender for Friends

```
friend_ratings = {
    "Friend 1": {
        "Handmaid's Tale": 8, # likes
        "Shutter Island": 9, # likes
        "Rum Punch": 7, # likes
        "L.A. Confidential": 8, # likes
```

```
        "Dracula": 10, # likes
        "Crime and Punishment": 7, # likes
        "Misery": 8, # likes
        "Catcher In The Rye": 9, # likes
        "The Godfather": 9, # likes
        "Frankenstein": 9, # likes
        "Shake Hands With The Devil": 6, # likes
        "The Great Gatsby": 6, # likes
        "Lord Of The Flies": 9, # likes
        "War Of The Worlds": 7, # likes
        "The Road": 10, # likes
        "The Giver": 4, # dislikes
        "The Tommyknockers": 3, # dislikes
        "Fifty Shades of Grey": 2, # dislikes
        "Fahrenheit 451": 4, # dislikes
        "The Da Vinci Code": 4, # dislikes
    },
    "Friend 2": {
        "A Little Life": 8, # likes
    ...
}
```

```
from collections import defaultdict
import pandas as pd

def compare_friend_top_books_binary_matrix(friend_name, friend_ratings_dict, num_top_books=3,
num_recommendations=10):
    strategy_names = [
        "title",
        "author",
        "subjects",
        "synopsis",
        "title_author",
        "title_subjects",
        "title_synopsis",
        "author_subjects",
        "author_synopsis",
        "subjects_synopsis",
        "title_author_subject",
        "title_author_synopsis",
        "title_subjects_synopsis",
        "author_subjects_synopsis",
        "title_author_subjects_synopsis"
    ]
    # filtering friend ratings to include only books rated >= 5 and present in dataset
    friend_books = pd.Series(friend_ratings_dict[friend_name])
    enriched_titles = set(enriched_smaller_df['Book-Title'])

    filtered_books = friend_books[(friend_books >= 5) & (friend_books.index.isin(enriched_titles))]

    if filtered_books.empty:
        print(f"No rated books >= 5 from {friend_name} were found in the dataset.")
        return pd.DataFrame()

    # get top N from the filtered set
    top_books = filtered_books.sort_values(ascending=False).head(num_top_books).index.tolist()
```

```
    print(f"Using these top books for {friend_name}: {top_books}")

    # run recommendations and track which strategies recommend each book
    book_to_strategies = defaultdict(set)

    # get book recommendations for each book with different strategies
    for book_title in top_books:
        for strategy in strategy_names:
            try:
                recs = improved_content_filter(book_title, num_recommendations, strategy)
                for rec_book in recs['Book-Title']:
                    book_to_strategies[rec_book.strip().lower()].add((book_title, strategy))
            except Exception as e:
                print(f"Error for book '{book_title}' with strategy '{strategy}': {e}")

    # build comparison binary matrix
    binary_data = []
    unique_books = list(book_to_strategies.keys())
    total_combinations = len(top_books) * len(strategy_names)

    for book in unique_books:
        row = {"Book-Title": book.title()}
        count = 0
        for base_book in top_books:
            for strategy in strategy_names:
                key = (base_book, strategy)
                is_rec = int(key in book_to_strategies[book])
                row[f"{base_book} | {strategy}"] = is_rec
                count += is_rec
        row["# Content-Based Strategies"] = count
        row["% Content-Based Strategies"] = round((count / total_combinations) * 100, 2)
        binary_data.append(row)

    return pd.DataFrame(binary_data).sort_values("# Content-Based Strategies", ascending=False)
```

```
from IPython.display import display, HTML

friend_df = compare_friend_top_books_binary_matrix("Friend 1", friend_ratings)

if not friend_df.empty:
    display(HTML(friend_df.to_html(notebook=True)))
```