Hieu Le

Building and evaluating recommender systems

Metropolia University of Applied Sciences

Bachelor of Engineering

Information Technology

Bachelor’s Thesis

4 May 2019

Author Title Number of Pages Date	Hieu Le Building and evaluating recommender systems 36 pages + 6 appendices 4 May 2019
Degree	Bachelor of Engineering
Degree Programme	Information Technology
Professional Major	Software Engineering
Instructors	Peter Hjort, Senior Lecturer Minna Paananen-Porkka, Senior Lecturer, Language Advisor

Recommender systems play an important role in the lives of people in today's information-rich environment. Businesses and companies utilize recommender systems to make meaningful suggestions to their users, increasing sales, and improving user experiences. Users make use of recommender systems to avoid wasting time finding the right products and contents, and to discover new perspectives and insights about what their likes and dislikes are. In most scenarios, recommender systems are highly likely to bring benefits to both businesses and their users, and thus, should be carefully planned and constructed so that a satisfactory outcome is achieved.

This thesis project is a demonstration of how technical implementations of recommender systems can be done. Five recommender systems were built based on the goodbooks-10k dataset, and their contexts and theories are given. The recommender systems were then evaluated according to an arbitrarily established evaluation strategy. Insights into different characteristics and qualities of the five recommender systems could be drawn from the evaluations.

It is concluded that the process of building and evaluating recommender systems are highly customizable, depending on multiple business-specific factors. Due to the heuristic nature of the building and evaluating processes, it is often the case that there is no one best way to implement such processes. It is advisable that businesses and companies perform careful planning and experimenting in order to reach a satisfactory result.

Keywords	recommender system, evaluation metric, theoretical algorithm, technical implementation, Python

Contents

Glossary

MAP	Mean average precision. The metric that quantifies the capability of a recommender system in recommending accurate and useful items.
CC	Catalog coverage. The metric that shows the fraction of the recommendable items over the item repository of a recommender system.
MNS	Mean novelty score. The metric that quantifies the capability of a recommender system in recommending novel and less popular items.
MDS	Mean diversity score. The metric that quantifies the capability of a recommender system in recommending diverse recommendation lists.
MPS	Mean personalization score. The metric that quantifies the capability of a recommender system in recommending personalized recommendation lists.
IBCF	Item-based collaborative filtering. The recommending method where ratings of multiple users are utilized to calculate the similarities between items.
CSV	Comma-separated values. The delimited text file format where tabular data is stored as plain text and values are separated by commas.

1 Introduction

Recommender systems are tools that help users discover products and content by recommending items that they think users would like. Recommender systems are utilized most notably by merchants to suggest personalized recommended items to their customers [1]. In the e-commerce industry, recommender systems have become an essential part of the business models of companies and businesses that offer a wide variety of products, such as Amazon and eBay [2]. In 2006, Netflix, a movie and TV show streaming provider, announced a competition with a prize of one million US dollars to improve the accuracy of their recommendation algorithm [3]. It is reasonable that Netflix was willing to invest a considerable amount of money into their recommendation algorithm. With 85 thousand titles to choose from, according to [4], Netflix's users would have difficulty hand-picking relevant movies by themselves.

Providers of other types of online content, such as YouTube (an online video sharing platform) and Spotify (a music service), face the same challenge. YouTube hosted billions of uploaded videos on their website [5], whereas Spotify offers more than 50 million songs with their service [6]. In today's information-rich environment, recommender systems play an essential role in human lives. Figure 1 shows an example of how Youtube suggests videos for their users to watch at their main website.

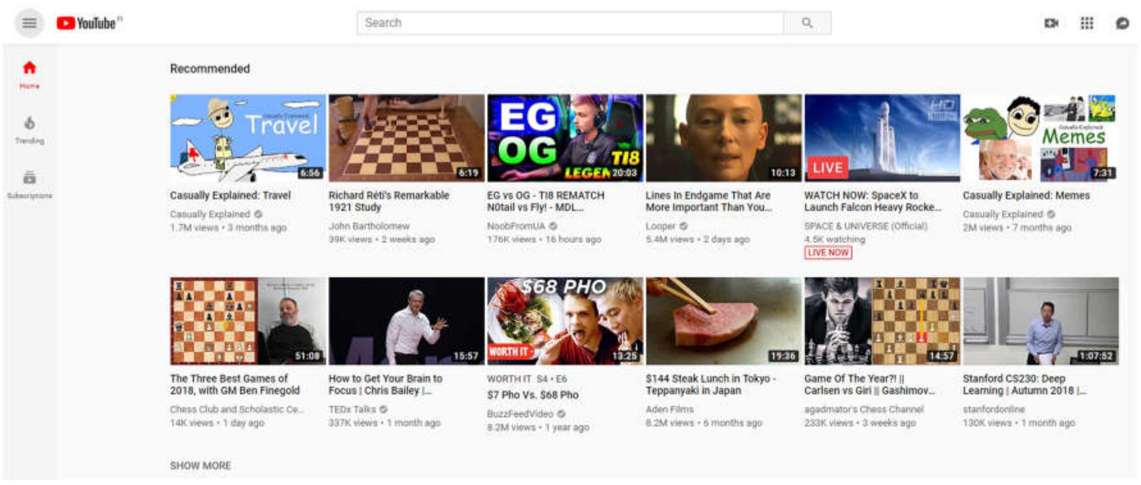

YouTube
Search
Home
Trending
Subscriptions
Recommended
Casually Explained: Travel
Casually Explained
1.7M views • 3 months ago
Richard Réti's Remarkable 1921 Study
John Bartholomew
39K views • 2 weeks ago
EG vs OG - TI8 REMATCH N0tail vs Fly! - MDL...
NoobFromUA
176K views • 16 hours ago
Lines In Endgame That Are More Important Than You...
Looper
5.4M views • 2 days ago
WATCH NOW: SpaceX to Launch Falcon Heavy Rocke...
SPACE & UNIVERSE (Official)
4.5K watching
LIVE NOW
Casually Explained: Memes
Casually Explained
2M views • 7 months ago
The Three Best Games of 2018, with GM Ben Finegold
Chess Club and Scholastic Ce...
14K views • 1 day ago
How to Get Your Brain to Focus | Chris Bailey |...
TEDx Talks
337K views • 1 month ago
WORTH IT S4 • E6
$7 Pho Vs. $68 Pho
BuzzFeedVideo
8.2M views • 1 year ago
$144 Steak Lunch in Tokyo - Teppanyaki in Japan
Aden Films
8.2M views • 6 months ago
Game Of The Year?! || Carlsen vs Giri || Gashimov...
agadmator's Chess Channel
233K views • 3 weeks ago
Stanford CS230: Deep Learning | Autumn 2018 |...
stanfordonline
130K views • 1 month ago
SHOW MORE

Figure 1. Example list of Youtube's video recommendation. Retrieved May 5, 2019, from https://www.youtube.com. Screenshot by author.

As depicted in Figure 1, Youtube’s recommendation algorithm filters billions of videos to get the top 12 videos that it thinks the user might like and watch. This algorithm addresses what is called the top-N recommendation problem, where recommender systems determine the top-N most relevant items to their users [7, p. 3]. This problem is the most common problem that online merchants aim to solve with recommender systems.

Goodbooks-10k is a book dataset recently published on Github [8]. The dataset contains six million ratings by more than fifty thousand users for ten thousand books. As the dataset is new, few research papers have been written with it being the main subject of the recommendation problem. In this thesis, the goodbooks-10k dataset serves as the focus of a study on recommender systems.

The primary goal of this thesis project was to demonstrate how recommender systems can be built and evaluated. Five recommender systems were built based on the goodbooks-10k dataset and aimed at solving the top-N recommendation problem. The chapters in this thesis are organized as follows. Chapter 2 introduces theoretical backgrounds about the top-N recommendation problem, the rating system, the evaluation strategy, and the recommending methods. Technical implementations of the recommender systems are described in Chapter 3. Chapter 4 provides the evaluations of the recommender systems built in Chapter 3. Finally, conclusions and summary are given in Chapter 5.

2 Theoretical background

2.1 Top-N recommendation problem

The top-N recommendation problem can often be reduced to the problem of calculating numerical values that represent a user's likelihood of performing positive actions on certain items that she has not interacted with. The items are then ranked based on the computed numerical values, and the top-N items are determined based on the ranking. The form of the said numerical value varies between different recommender systems. Recommender systems based on different assumptions have different theories on why a specific type of numerical value correlates with users' tendency to act positively on items. In this thesis, the mentioned numerical value is referred to as the ranking score.

In this thesis, an arbitrary value of 10 as N is chosen when solutions to the top-N recommendation problem are constructed. The recommender systems built in this project were evaluated based on the quality of the top 10 items they suggest. Choosing 10 as N is a reasonable choice as a user will usually be distracted if her recommended list is too long.

It is worth noting that the value of N was not a focus in this project since the value of N has little effect on the process of building and evaluating recommender systems. The process laid out in this thesis applies to any value of N that is larger than 0. In this thesis project, many arbitrary decisions were made when building and evaluating the recommender systems, such as choosing the value of N, selecting the metrics in the evaluation strategy, or choosing the values of the parameters used in the algorithms. These decisions are highly business-specific and thus, should be made based on the context and the field of the business in question. When one applies the procedures mentioned in this thesis, she should replace the arbitrary decisions in this project by careful experimentation so that the resulted recommender systems are suitable to her problems and business field.

2.2 Rating system

To make meaningful and personalized recommendations, recommender systems require the presence of a feedback system where users could explicitly state their opinions and preferences. A typical type of feedback system is the rating system, where a user's preference of an item is represented as a numerical value belonging to a discrete set of ordered numbers [7, p. 10]. An example of a rating system is illustrated in Figure 2.

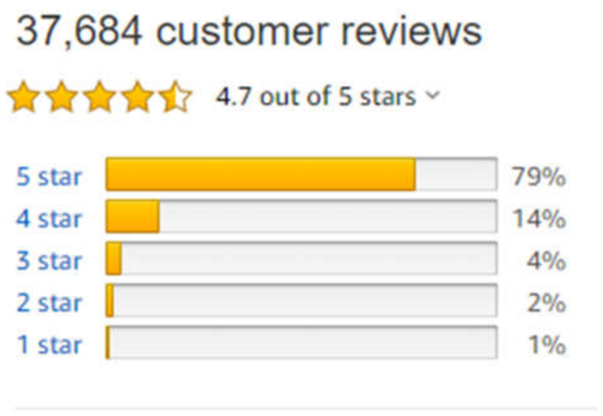


Figure 2. Ratings of the book The Fault in Our Stars on Amazon.com. Retrieved May 6, 2019, from https://www.amazon.com/The-Fault-in-Our-Stars/dp/B006VPAXQY. Screenshot by author.

Figure 2 demonstrates how Amazon's users give their feedback on the e-commerce site. To state their opinions, users can choose from a set of five values, intuitively from 1-star as the least positive response to 5-star as the most positive response. The most common rating scales employed by rating systems are 5-point, 7-point, and 10-point [7, p. 10].

The goodbooks-10k dataset also uses a rating system with a 5-point scale. Users' ratings are stored as numerical values ranging from 1 to 5. In this thesis, the ratings corresponding to a numerical value are referred to as 1-ratings, 2-ratings, and so on.

2.3 Evaluation strategy

In order to evaluate a recommender system, a systematic evaluating strategy must be established. This thesis focuses on a few evaluation factors that indicate the quality of recommendations outputted by recommender systems. Each of the following factors has been researched intensively in the recommender system community and is often regarded to affect the experience of users. An improvement in these factors usually translates to an improvement in the performance of recommender systems when real users use them.

2.3.1 Accuracy

Accuracy is an essential factor of most evaluating strategies [7, p. 229]. Generally, in the context of the top-N recommendation problem, accuracy can be understood as the tendency that the users will react positively to the recommended items. In a 5-point rating system, it is safe to assume that if a user gives an item a 5-rating, that item is relevant to the said user, since 5-rating is the highest available rating that a user can give. The accuracy of recommender systems that operate on a 5-point rating system can then be determined by how likely a user will give a 5-rating to a recommended item.

The problem of recommending relevant items has a close connection to the problem of information retrieval, as recommending can be considered as an act of retrieving information, and metrics that measure the accuracy of information retrieval systems can be utilized to measure the accuracy of recommender systems as well. In this thesis, the metric that is used to measure the accuracy of recommender systems is the mean average precision (MAP) metric. According to [7, pp. 244-247], MAP is classified as a utility-based measure, whose overall goal is to quantify the usefulness of recommender systems.

Let x_k be the binary value that denotes whether the $k\text{th}$ item in a ranked recommend list is relevant to the user receiving the recommendations:

$$x_k = \begin{cases} 1 & (\text{if } k\text{th item is relevant}) \\ 0 & (\text{if } k\text{th item is irrelevant}) \end{cases} \tag{1}$$

Therefore, precision at k ($P@k$), which computes the fraction of relevant items in the first k items in the recommended list, can be calculated as follows:

$$P@k = \frac{1}{k}\sum_{i=1}^{k} x_i \tag{2}$$

Average precision (AP) for a recommended list of length N is defined as follows:

$$AP = \frac{1}{\min\{R,N\}}\sum_{i=1}^{N} P@i \tag{3}$$

where R is the total number of relevant items in the repository. In the case that R is less than N, averaging the precision values by R instead of N is more intuitive as otherwise a recommender system would be wrongfully penalized for recommending N items even if it correctly places all R relevant items to the top of the recommended list.

Finally, MAP is computed by averaging the AP calculated across all the users:

$$MAP = \frac{\text{sum total of } AP \text{ across all users}}{\text{number of users}} \tag{4}$$

One advantage that MAP has over other traditional metrics used in academic research, such as the root mean squared error metric, is that it gives greater importance to the order of the recommended list [7, p. 244]. It is intuitive that a recommended list that places relevant items closer to the top is evaluated as more accurate compared to a list that places relevant items closer to the bottom.

2.3.2 Coverage

It is often the case that there is a fraction of the repository of items that a recommender system will not be able ever to suggest even if it possesses excellent accuracy [7, p. 231]. Coverage of a recommender system describes how many unique items it can incorporate into its recommendation algorithm compared to the total number of items in the repository. One of the metrics that can quantify this factor is catalog coverage (CC), which is the fraction of items that are at least recommended to one user:

$$CC = \frac{\text{number of items that are recommended at least once}}{\text{number of items in the repository}} \tag{5}$$

CC is particularly suitable for the evaluation of top-N recommendation lists, while other metrics are more fitted to evaluate a recommender system's ability to predict ratings [7, p. 232].

2.3.3 Novelty

Novelty is considered a desirable characteristic of recommender systems [8, p. 110]. Novelty can typically be understood as a recommender system's ability to suggest less popular items [8, p. 110]. Novel recommendations usually help users take a more-in-depth look into their likes and dislikes that they are not aware of [7, p. 233].

A method of quantifying novelty is through measuring the unexpectedness of the recommended items in the list compared to their global popularity [9, p. 3]. An item's popularity (IP) can be measured by computing the fraction of users who have rated the said item over the total number of users in the system:

$$IP = \frac{\text{number of users who have rated the item}}{\text{number of users}} \tag{6}$$

An item's novel value (INV) is then measurable by taking the log of the inverse IP:

$$INV = -\log_2(IP) \tag{7}$$

The novelty of a recommended list, referred to as the novelty score (NS), is then calculated by averaging the INV of the items recommended in the list:

$$NS = \frac{\text{sum total of } INV \text{ across all items in the list}}{N} \tag{8}$$

The NS calculated for every user can then be averaged to get the mean novelty score (MNS), which quantifies a recommender system's ability to recommend novel items:

$$MNS = \frac{\text{sum total of } NS \text{ across all users}}{\text{number of users}} \tag{8}$$

where higher MNS indicates better ability in suggesting novel recommendations.

2.3.4 Diversity

Essentially, diversity can be understood as how different content-wise items within a single recommended list are from each other [7, p. 234]. Keywords that represent the attributes of items are often utilized to formalize a similarity function, which can be used in measuring diversity as well as recommending items. Keywords are typically weighted differently based on how important they are in describing an item, as a keyword about a book's author might be more descriptive than a keyword stating whether the said book is an e-book or an audio-book.

The similarity function can be formulated based on the vector-space representation of item keywords [7, p. 234]. One of the most common ways of formulating a similarity function is the use of the cosine function [7, p. 151]. Let $\overline{X} = (x_1 \ldots x_d)$ and $\overline{Y} = (y_1 \ldots y_d)$ be a pair of items, where x_i and y_i denote the weight of the ith keyword in the two items. The cosine similarity function is then defined as follows:

$$sim(\overline{X}, \overline{Y}) = cosine(\overline{X}, \overline{Y}) = \frac{\sum_{i=1}^{d} x_i y_i}{\sqrt{\sum_{i=1}^{d} x_i^2} \sqrt{\sum_{i=1}^{d} y_i^2}} \qquad (10)$$

The results retrieved from applying the similarity function pair-wise between items in a recommended list can be averaged to get the quantification of the diversity of a recommended list, referred to as the average similarity (AS) of a recommended list L of length N:

$$AS = 2 \frac{\sum_{i \in L} \sum_{j \in L\, (i \neq j)} sim(i,j)}{N(N-1)} \qquad (11)$$

As AS always lies in the range $[0, 1]$, diversity score (DS) of a recommended list can be calculated as follows:

$$DS = 10 \cdot (1 - \mathrm{AS}) \qquad (12)$$

The average value of DS across all users in the system is referred to as the mean diversity score (MDS), which measures the diversity of a recommender system:

$$MDS = \frac{\text{sum total of } DS \text{ across all users}}{\text{number of users}} \qquad (13)$$

where MDS lies in the range $[0, 10]$ and higher MDS denotes a higher capability of recommending a diverse list of items.

2.3.5 Personalization

Personalization has a close relation to diversity. A personalized recommender system usually suggests different items to different users [9, p. 3]. A metric that can quantify this factor is the inter-list distance (ILD), which is calculated between two lists of recommended items:

$$ILD = 1 - \frac{\text{the number of common items in two lists}}{N} \tag{14}$$

Personalization score (PS) for a user u's recommended list is determined by the average ILD computed between her and all other users, multiplied by 10:

$$PS(u) = 10\frac{\sum ILD\ between\ u\ and\ other\ users}{\text{the number of users - 1}} \tag{15}$$

Finally, the mean personalization score (MPS), which measures the personalization quality of a recommender system, is calculated by averaging the PS measured across all users in the system:

$$MPS = \frac{\text{sum total of } PS \text{ across all users}}{\text{number of users}} \tag{16}$$

where MPS lies in the range $[0, 10]$ and higher MPS implies greater capacity of recommending personalized recommendations.

2.4 Recommendation methods

Different recommender systems have different assumptions and methods in solving the recommendation problem. This thesis project focuses on five basic approaches to constructing a recommended list of items. Each approach's assumptions and characteristics are described below.

2.4.1 Random-based recommender system

A random-based recommender system assumes that users prefer arbitrary recommendations that disregard both the items' characteristics and their past interactions with the rating system. While the assumption is arguably not sound, this type of recommender system can provide highly surprising recommendations that are usually not expected by users due to its random nature.

Another strong point of the random-based approach is the ease of implementing a recommender system that follows the said approach. In this project, a random-based recommended list for a user consists of N random items chosen from a set of items that the said user has not interacted with, referred to as a recommendable set. The computation and complexity of the aforementioned process are trivial compared to the process of building the other four types of recommender systems.

A possible alternative to produce random-based recommendations is to generate a ranking score value that lies in a specific range, such as $[0, 1]$, for each item in the recommendable set. The items are then ranked based on the generated score, and the top N items are selected based on the ranking. A possible ranking score formula for an item i is as follows:

$$score(i) = \text{a randomized number in the range } [0, 1] \tag{17}$$

While this option requires more computational resources to operate, the generated score can be used together with other types of numerical values computed by other types of recommender systems to form a hybrid recommender system, which is further explained in a later section.

2.4.2 Popularity-based recommender system

A popularity-based recommender system assumes that users prefer popular items. The notion of popularity can be interpreted in different ways, and as a result, different methods of producing recommendations can be formed.

A possible interpretation of popularity is the number of ratings an item has. A popular item is intuitively known by many users and consequently is rated by many users. This interpretation assumes that the higher the number of ratings an item has, the more likely it will be reacted positively by users.

In this project, the interpretation of popularity was decided as the number of 5-ratings an item has, and a higher number of 5-ratings denotes greater popularity. An advantage of this approach, compared to the previous interpretation, is that this method takes ratings into account when constructing recommendations. It is often the case that the more resources a recommender system can utilize, the more useful recommendations it can suggest. Therefore, there is a high chance that recommending items by their numbers of 5-ratings leads to better results than that produced by recommending based on numbers of ratings. The formula for the ranking score is as follows:

$$score(i) = \text{number of 5-ratings item } i \text{ has} \tag{18}$$

While it is relatively easy to implement a popularity-based recommender system, the resulted recommendations are usually not novel nor personalized. Recommending only popular items directly diminishes the novelty of the recommendations and restricting the pool of items to only popular ones results in greater overlaps between recommended lists suggested to different users. Another downside of a popularity-based recommender system is its poor coverage, which is also a consequence of having a limited collection of items to recommend.

2.4.3 Content-based recommender system

Fundamentally, content-based recommender systems are systems that utilize items' descriptions to make recommendations to users [7, p.139]. The assumption of content-based systems is that if a user has reacted positively to items possessing certain characteristics, she must also be interested in similar items which are described as having the same attributes. The most common representations of items' characteristics are keywords and the keywords' weights, which denote how important the keywords are in describing the items.

The most elementary approach, which is also the approach the system built in this project follows, to building a content-based system is the determining of the top-k lists of similar items [7, p. 151]. This approach requires a content-centric similarity function, for which Equation 10 is a good candidate. Given an item, this method returns the top k most similar items and their similarity weight, which are leveraged to make recommendations.

There are many ways one can utilize the top k most similar item lists in the process of producing recommendations. A simple method that this project focuses on is to find the top-k list for all items that have been given a 5-rating by a user. Items that do not appear in any list are then considered not relevant and removed from the pool of recommendable items. The ranking score for a relevant item is then calculated by summing up its available similarity weight across lists, meaning that for the lists that do not contain the said item its similarity weight is 0:

$$score(i) = \text{aggregated available similarity weight of } i \text{ across lists} \tag{19}$$

One advantage this approach has is its simplicity and low computational complexity. Because most of the irrelevant items are removed after applying the k-nearest neighbor algorithm, the calculation is only required to operate on a limited set of items, whereas the process in [7, p. 152] requires predictions to be made for all items in the testing set.

Content-based recommender systems are often useful in cold-start scenarios, where other users' ratings are of little use [7, p. 140], due to the fact that these systems do not require other users' ratings to make recommendations. However, recommendations produced by content-based systems are typically not as diverse as that of other systems [7, p. 15]. The use of a content-centric similarity function for ranking the items results in recommended lists consisting of highly similar items content-wise, reducing the diversity of the recommendations.

2.4.4 Item-based collaborative filtering recommender system

A collaborative filtering recommender system makes use of multiple users' ratings to suggest recommendations [7, p. 8]. An item-based collaborative filtering (IBCF) recommender system assumes that item A is similar to item B if they receive similar ratings and if a user reacts positively to item A, she will like item B as well.

The process of building an IBCF system in this thesis is similar to that of building the content-based system, with the only difference in choosing the similarity function. Instead of a content-centric similarity function, the IBCF systems utilize a collaborative filtering similarity function, which leverages items' ratings in determining similarity.

Among available similarity functions, the adjusted cosine similarity has been reported to offer better performance [10, p. 291], with the reason being that it takes users' rating scales into consideration [10, p. 288]. Let $R_{u,i}$ be the rating that user u gives to item i, the adjusted cosine similarity that measures the similarity between item i and item j is defined as follows:

$$sim(i,j) = \frac{\sum_{u \in U}(R_{u,i} - \overline{R}_u)(R_{u,j} - \overline{R}_u)}{\sqrt{\sum_{u \in U}(R_{u,i} - \overline{R}_u)^2}\sqrt{\sum_{u \in U}(R_{u,j} - \overline{R}_u)^2}} \quad (29)$$

where $\overline{R}_u$ is the average of user u's ratings and U is the set of users that have rated both item i and item j.

IBCF systems boast good recommendation capacity, as their algorithms suggest highly correlated items [1, p. 79]. A critical disadvantage of IBCF systems is their computationally intensive operations [7, p. 44]. The number of users in U is often many times larger than the number of items' characteristics and consequently, the time it takes to compute the collaborative filtering similarity between every pair of items is many times slower than that of computing the content-centric similarity. Amazon addressed this problem by only computing the similarity between items that are bought together [1, p. 79]. This thesis provides a workaround that still allows the computation of similarity for every pair, but comes with a different possible downside, which is further detailed in Chapter 3.

2.4.5 Hybrid recommender system

Generally, a hybrid recommender system is a combination of multiple other recommender systems with the aim to achieve the best of all worlds [7, p. 19]. A hybrid system's assumption is a combination of different assumptions coming from the systems it integrates. As a result, a hybrid system has access to more resources, based on which it can provide more robust recommendations [7, p. 199]. However, an obvious downside to implementing a hybrid system is the extra computational complexity added for operating multiple recommender systems simultaneously.

A common implementation of the hybrid approach is the weighted hybrid system, where the scores of several systems are combined into a single score using a set of weights, which are often determined through trial and error [7, p. 201]. In this thesis, a combination of a content-based recommender system and an item-based collaborative filtering recommender system is experimented with different values of weights. It is worth noting that the main focus of the experimentation is not about optimizing the system, but more about examine the effects a hybrid system has over the evaluations of two stand-alone recommender systems. The formula for the ranking score calculated by the hybrid system is as follows:

$$score(i) = score_{ctb}(i) + w \cdot score_{ibcf}(i) \qquad (21)$$

where $score_{ctb}$ and $score_{ibcf}$ are respectively the ranking scores outputted by the content-based recommender system and the item-based collaborative filtering recommender system. Equation 21 only uses a single weight w as there are only two systems being integrated and w can be set to a fraction if one wants to give greater importance to the output of the content-based recommender system.

3 Implementation

3.1 Technology

3.1.1 Python

Python is a high-level and general-purpose interpreted programming language. The Python programming language is widely used in the field of scientific computing, due to its suitability for developing algorithms and performing exploratory data analysis [11].

Some of Python's key features are its beginner-friendliness, high-level data structures, and elegant syntax, making it an appealing choice for fast development requirements [12]. Python is highly popular, consistently making into the top 5 most popular programming languages in various ranking indices, such as the TIOBE Index [13], the PYPL Index [14] and the RedMonk Programming Language Rankings [15]. Its applications span across many different areas, such as web development, database programming, graphical user interfaces for desktop, and game development [16].

The version of Python used in this thesis project is 3.7.3, which is also the latest version released. All coding works of this project were written in Python.

3.1.2 Jupyter Notebook

Jupyter Notebook is a web-based application that serves as a development environment, whose important use cases include rapid experimenting, performing analysis, and communicating ideas and outcomes [17]. In this thesis project, the procedures of experimenting and building recommender systems were done on Jupyter Notebook.

3.1.3 Pandas

Pandas is often considered the most popular data analysis and manipulation tool in Python. Pandas is a Python library that provides high-level data structures suitable for performing data-related work. Pandas is widely used in the fields of finance, statistics, and various engineering areas. [18.] Pandas is a core tool in this project. Tabular data, such as the book dataset and the rating dataset, were stored, analyzed, and processed using Pandas.

3.1.4 NumPy and SciPy

NumPy is an essential library for scientific computation in Python. This package offers effective implementations of multi-dimensional arrays and matrices, as well as advanced mathematical functions that operate on those data structures. [19.]

SciPy is a collection of mathematical algorithms and common numerical routines in science and engineering. Its functions are built on top of NumPy’s data structures. [20.]

The scientific procedures in this project were operated using either NumPy or SciPy.

3.1.5 Scikit-learn

Scikit-learn is a Python machine learning library based on NumPy and SciPy [11, 21]. It offers advanced implementations of popular machine learning algorithms, as well as a user-friendly interface [11]. In this project, Scikit-learn was used for tasks that are in the machine learning area.

3.2 Dataset

Goodbooks-10k is a relatively new public book dataset. The dataset includes six million ratings for ten thousand books with the most ratings, as well as other data that are useful for recommending items, such as book metadata and tag data [8]. The dataset is divided into comma-separated values (CSV) files that store the sub-datasets:

- `ratings.csv` contains rating entries of the unique combinations of `user_id` and `book_id`. The `rating` column is of the categorical range $[1, 2, 3, 4, 5]$.
- `books.csv` contains the metadata, such as title and authors, of books.
- `tags.csv` contains the list of tags and their identifier.
- `book_tags.csv` contains the number of user-voted counts corresponding to unique combinations of `book_id` and `tag_id`. A count value implies the importance and descriptiveness of a corresponding tag to a corresponding book.

3.3 Testing methodology

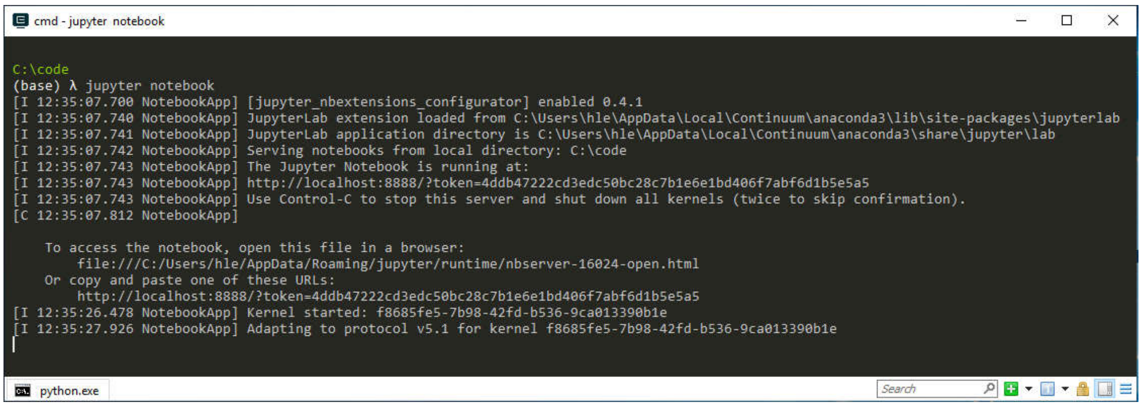
To apply the evaluation strategy in Section 2.3, a testing set that is completely unknown to the recommender systems must be defined. The rating dataset downloaded from [8] was partitioned into a training set and a testing set. As the recommender systems were tested on the ability to recommend items that have high chances of receiving 5-ratings, the testing set only consists of 5-ratings. The recommender systems were then trained on the training set and evaluated on the testing set.

It is highly likely that there is no official statement about what proportion of the dataset should be used for testing, due to the heuristic nature of the machine learning field. In this thesis project, an arbitrary value of one-third was used as the proportion of 5-ratings to be used for testing. However, outside of this project, this proportion value should be experimented based on the dataset and the business field that the recommender systems operate on. It is reasonable to assume that there is no one proportion value that works well in every situation, and thus, the value should be derived through trial and error.

3.4 Procedures

3.4.1 Starting Jupyter Notebook and importing Python libraries

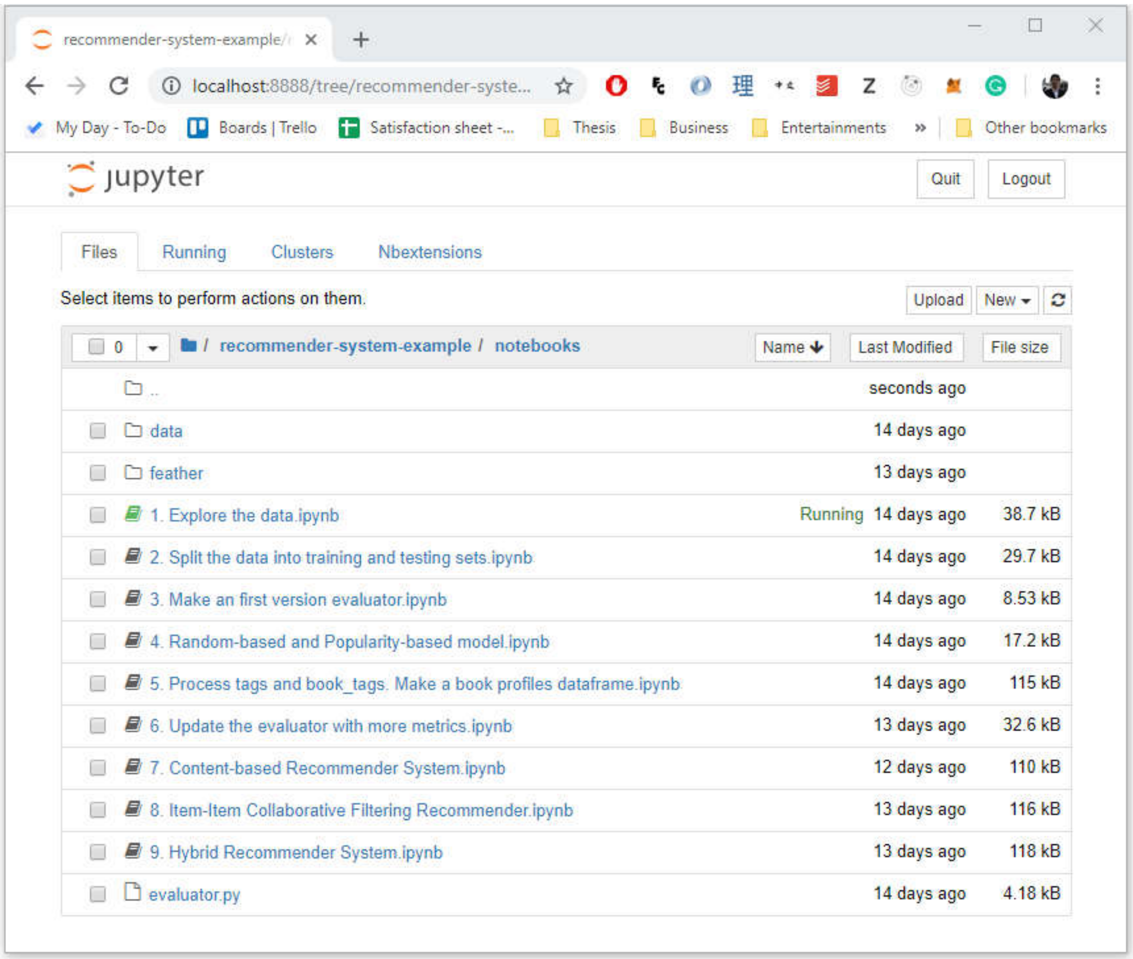
After being installed, Jupyter Notebook could be started from the command line, shown in Figure 3.



```
C:\code
(base) λ jupyter notebook
[I 12:35:07.700 NotebookApp] [jupyter_nbextensions_configurator] enabled 0.4.1
[I 12:35:07.740 NotebookApp] JupyterLab extension loaded from C:\Users\hle\AppData\Local\Continuum\anaconda3\lib\site-packages\jupyterlab
[I 12:35:07.741 NotebookApp] JupyterLab application directory is C:\Users\hle\AppData\Local\Continuum\anaconda3\share\jupyter\lab
[I 12:35:07.742 NotebookApp] Serving notebooks from local directory: C:\code
[I 12:35:07.743 NotebookApp] The Jupyter Notebook is running at:
[I 12:35:07.743 NotebookApp] http://localhost:8888/?token=4ddb47222cd3edc50bc28c7b1e6e1bd406f7abf6d1b5e5a5
[I 12:35:07.743 NotebookApp] Use Control-C to stop this server and shut down all kernels (twice to skip confirmation).
[C 12:35:07.812 NotebookApp]

    To access the notebook, open this file in a browser:
        file:///C:/Users/hle/AppData/Roaming/jupyter/runtime/nbserver-16024-open.html
    Or copy and paste one of these URLs:
        http://localhost:8888/?token=4ddb47222cd3edc50bc28c7b1e6e1bd406f7abf6d1b5e5a5
[I 12:35:26.478 NotebookApp] Kernel started: f8685fe5-7b98-42fd-b536-9ca013390b1e
[I 12:35:27.926 NotebookApp] Adapting to protocol v5.1 for kernel f8685fe5-7b98-42fd-b536-9ca013390b1e
```

Figure 3. Command line when Jupyter Notebook was started.

The web application was then launched and is accessible using any web browsers. The link to the application was displayed in the command line. In Figure 3, the link starting with `http://localhost:8888/?token=` was the link used to access the application. The current directory in the command line then appeared on the user interface of Jupyter Notebook. The application could then be navigated by selecting the directory in-browser. Figure 4 shows the user interface after it had been navigated to the directory of this project.



Select items to perform actions on them.

0 / recommender-system-example / notebooks

Name	Status	Last Modified	File size
..		seconds ago	
data		14 days ago	
feather		13 days ago	
1. Explore the data.ipynb	Running	14 days ago	38.7 kB
2. Split the data into training and testing sets.ipynb		14 days ago	29.7 kB
3. Make an first version evaluator.ipynb		14 days ago	8.53 kB
4. Random-based and Popularity-based model.ipynb		14 days ago	17.2 kB
5. Process tags and book_tags. Make a book profiles dataframe.ipynb		14 days ago	115 kB
6. Update the evaluator with more metrics.ipynb		13 days ago	32.6 kB
7. Content-based Recommender System.ipynb		12 days ago	110 kB
8. Item-Item Collaborative Filtering Recommender.ipynb		13 days ago	116 kB
9. Hybrid Recommender System.ipynb		13 days ago	118 kB
evaluator.py		14 days ago	4.18 kB

Figure 4. Jupyter Notebook running on Google Chrome.

A new coding session, referred to as a notebook, could then be created through the user interface, and saved with under the `.ipynb` file extension. As Figure 4 illustrated, there were nine notebooks available in the project. An example of a blank notebook is depicted in Figure 5.

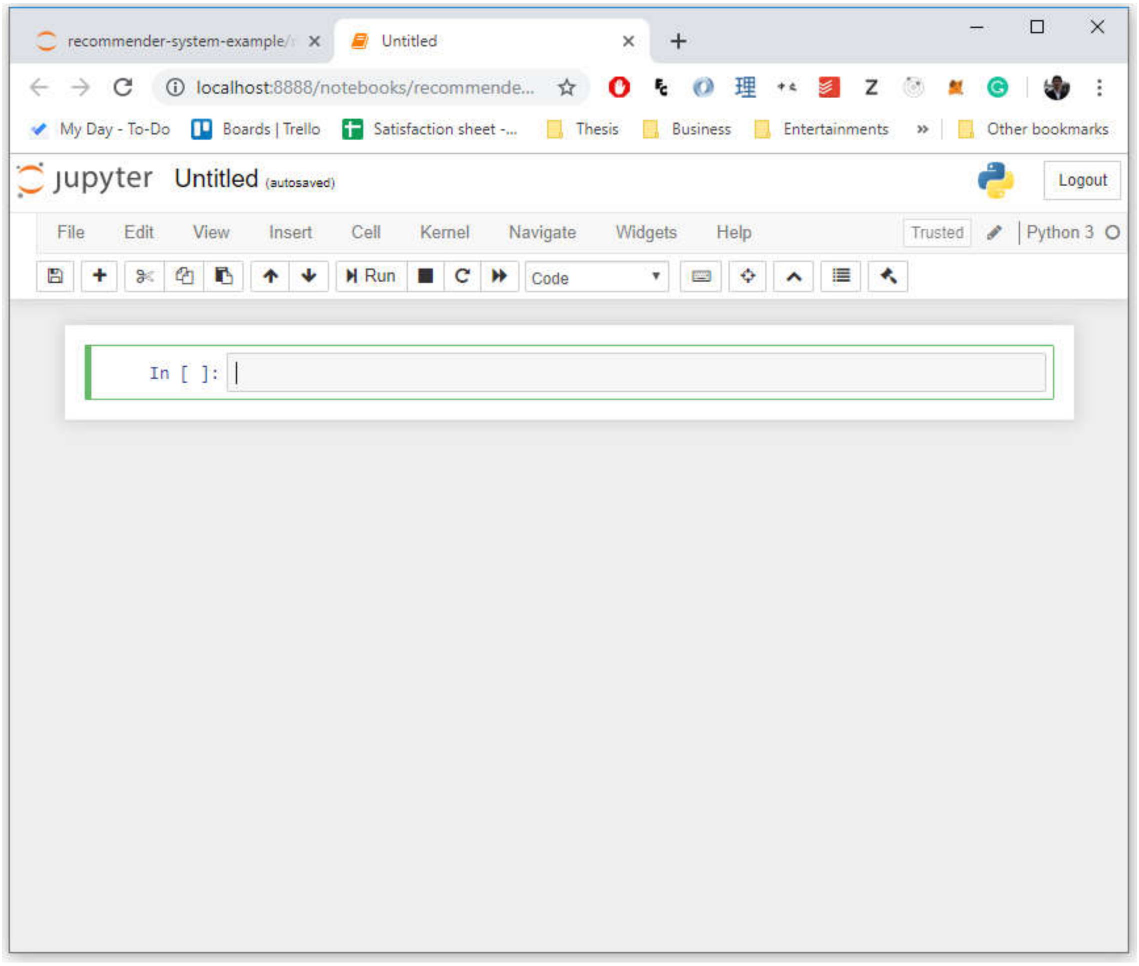
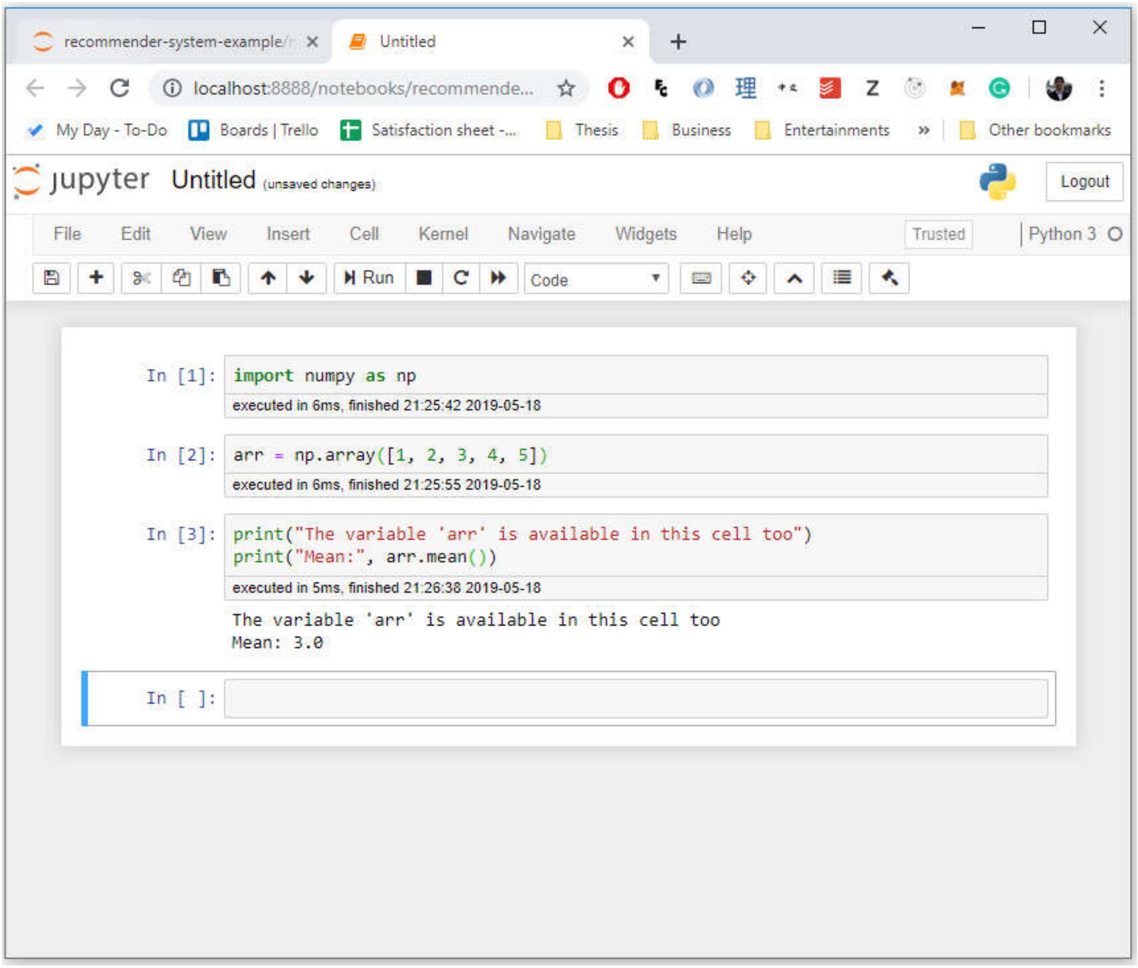


Figure 5. An empty notebook.

The block that appears in Figure 5 is referred to as a cell, and a notebook usually has many cells. Each cell is written and executed separately, but a variable that is defined and modified in one cell can be reused in another cell, thus, allowing rapid experimenting without having to re-run the whole notebook again. Figure 6 shows how codes in a notebook are written and executed.


recommender-system-example/r
Untitled
localhost:8888/notebooks/recommende...
My Day - To-Do
Boards | Trello
Satisfaction sheet -...
Thesis
Business
Entertainments
Other bookmarks
jupyter Untitled (unsaved changes)
Logout
File Edit View Insert Cell Kernel Navigate Widgets Help
Trusted
Python 3
Run
Code

```
In [1]: import numpy as np
        executed in 6ms, finished 21:25:42 2019-05-18

In [2]: arr = np.array([1, 2, 3, 4, 5])
        executed in 6ms, finished 21:25:55 2019-05-18

In [3]: print("The variable 'arr' is available in this cell too")
        print("Mean:", arr.mean())
        executed in 5ms, finished 21:26:38 2019-05-18

        The variable 'arr' is available in this cell too
        Mean: 3.0

In [ ]:
```

Figure 6. Example codes executed in Jupyter Notebook.

The first cell in Figure 6 is an example of importing a Python library into the notebook. Because not all notebooks needed to use all the libraries mentioned in Section 3.1, in this thesis, importing statements are only mentioned when it is necessary. Each block of codes without outputs mentioned in this thesis was written in the same cell, and is shown as a listing instead of a figure:

```
import pandas as pd
import numpy as np
```

Listing 1. Importing statements for Pandas and NumPy.

Listing 1 and Figure 6 show how Pandas and NumPy were imported into the notebook with aliases `pd` and `np`, respectively. It is conventional in the scientific community that Pandas and NumPy are imported under those aliases.

3.4.2 Analyzing the dataset

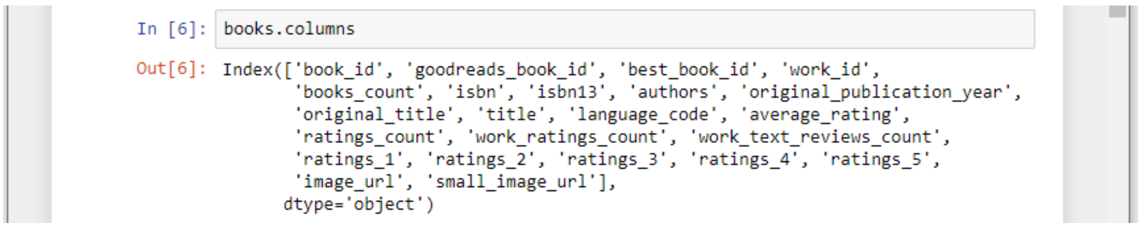
The goodbooks-10k dataset was stored under the `data` directory shown in Figure 4. Listing 2 shows how Pandas loaded CSV files into the notebook.

```
books = pd.read_csv('./data/books.csv')
ratings = pd.read_csv('./data/ratings.csv')
tags = pd.read_csv('./data/tags.csv')
book_tags = pd.read_csv('./data/book_tags.csv')
```

Listing 2. Importing the sub-datasets into the notebook using Pandas.

Before using the data, it is often helpful to perform some simple data analysis methods to discover useful information, such as what the data looks like or which columns are usable. In this thesis, not all analysis codes are shown due to the experimental nature of data analyzing, and only a few meaningful analysis parts are mentioned.

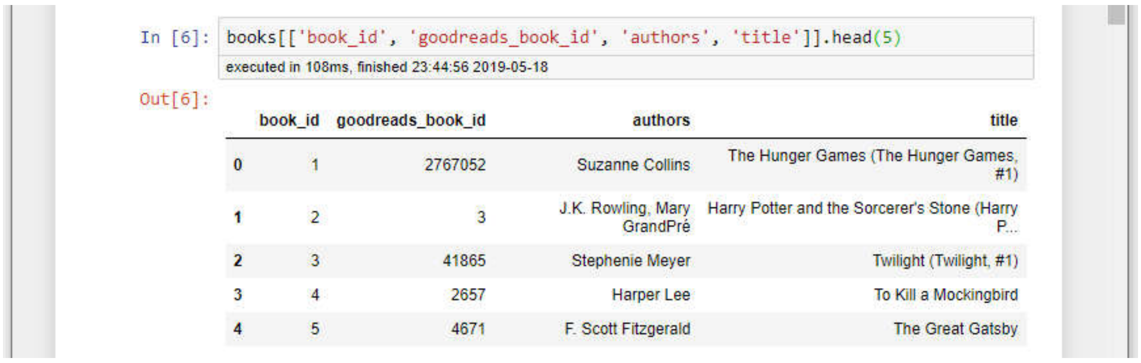
The `books` dataset contains ten thousand books. Figure 7 shows the columns of the `books` dataset.



```
In [6]: books.columns

Out[6]: Index(['book_id', 'goodreads_book_id', 'best_book_id', 'work_id',
               'books_count', 'isbn', 'isbn13', 'authors', 'original_publication_year',
               'original_title', 'title', 'language_code', 'average_rating',
               'ratings_count', 'work_ratings_count', 'work_text_reviews_count',
               'ratings_1', 'ratings_2', 'ratings_3', 'ratings_4', 'ratings_5',
               'image_url', 'small_image_url'],
              dtype='object')
```

Figure 7. List of columns of the `books` dataset.

There were only a few columns depicted in Figure 7 that the project makes use of, namely `book_id`, `goodreads_book_id`, `authors`, and `title`. The rating-related data in the `books` dataset are not derived from the `ratings` dataset and thus were ignored to avoid inconsistency. Since none of the four columns above had a missing entry, there was no need to remove invalid rows. Figure 8 shows the first five books in the `ratings` dataset.

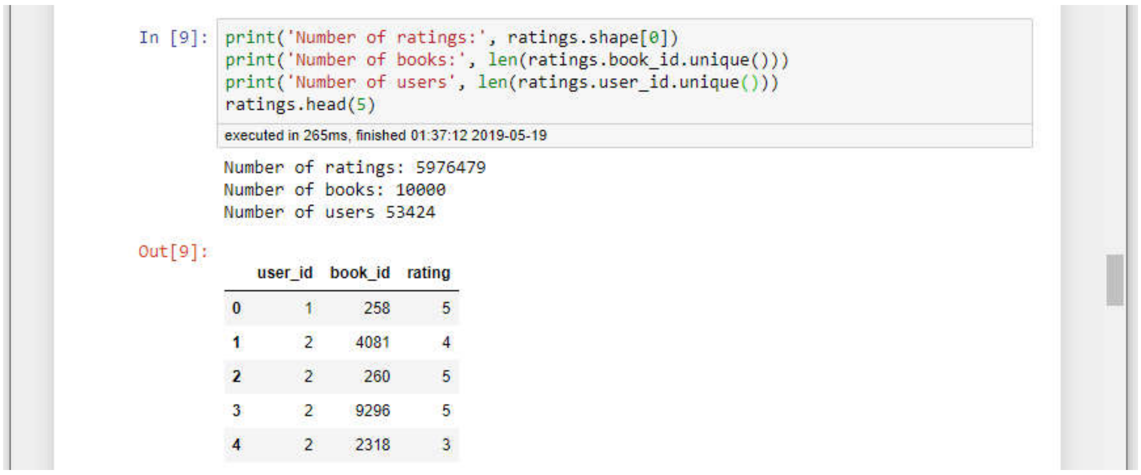

```
In [6]: books[['book_id', 'goodreads_book_id', 'authors', 'title']].head(5)
executed in 108ms, finished 23:44:56 2019-05-18
Out[6]:
```

	book_id	goodreads_book_id	authors	title
0	1	2767052	Suzanne Collins	The Hunger Games (The Hunger Games, #1)
1	2	3	J.K. Rowling, Mary GrandPré	Harry Potter and the Sorcerer's Stone (Harry P...
2	3	41865	Stephenie Meyer	Twilight (Twilight, #1)
3	4	2657	Harper Lee	To Kill a Mockingbird
4	5	4671	F. Scott Fitzgerald	The Great Gatsby

Figure 8. First five books listed in the books dataset.

It is worth noting in Figure 8 that the authors column was of type string, which means that it is highly likely that the column needed to be pre-processed into another structure that better represents multiple authors.

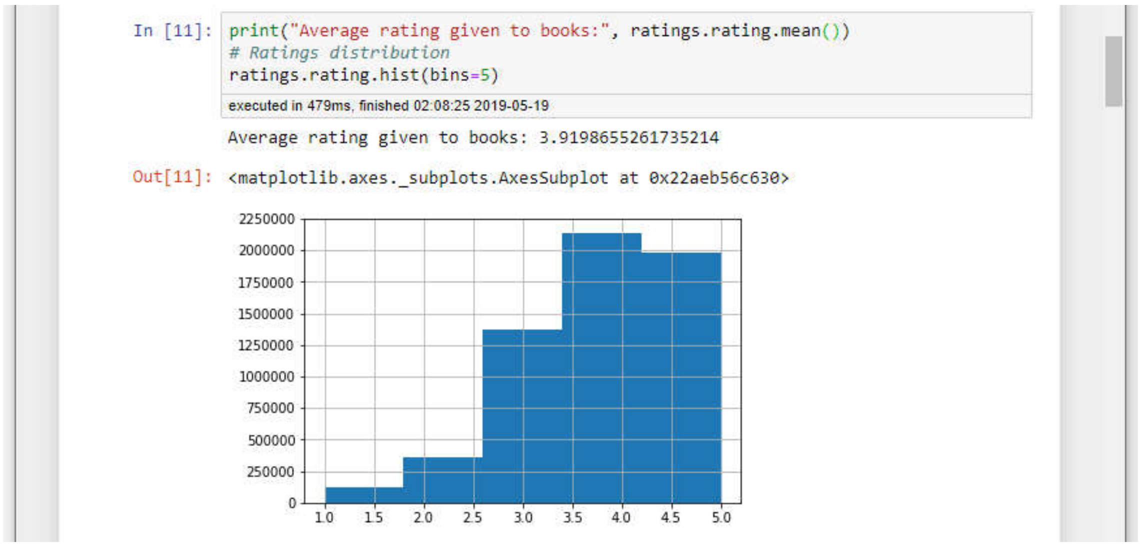
The ratings dataset contains almost 6 million rating entries. It could be found from the ratings dataset that there are about 53 thousand users. One reason why this project chose to implement an item-based collaborative filtering system instead of a user-based one (a system that finds similarities between users instead of items) was that the number of users is five times greater than the number of books. Finding pairwise user similarities is then 25 times costlier than computing item similarities. The first five rating entries can be seen in Figure 9.


```
In [9]: print('Number of ratings:', ratings.shape[0])
        print('Number of books:', len(ratings.book_id.unique()))
        print('Number of users', len(ratings.user_id.unique()))
        ratings.head(5)
executed in 265ms, finished 01:37:12 2019-05-19

        Number of ratings: 5976479
        Number of books: 10000
        Number of users 53424
Out[9]:
```

	user_id	book_id	rating
0	1	258	5
1	2	4081	4
2	2	260	5
3	2	9296	5
4	2	2318	3

Figure 9. First five rating entries and statistics about the number of ratings, books, and users.

Figure 9 illustrates how simple statistics such as the number of books and users could be acquired using Pandas. As Figure 9 depicts, a rating entry consists of a user_id, a book_id, and a rating. From the example entries, it is safe to state that the user with user_id 1 likes the book with book_id 258, whereas the user with user_id 2 does not enjoy the book with book_id 2318 as much.

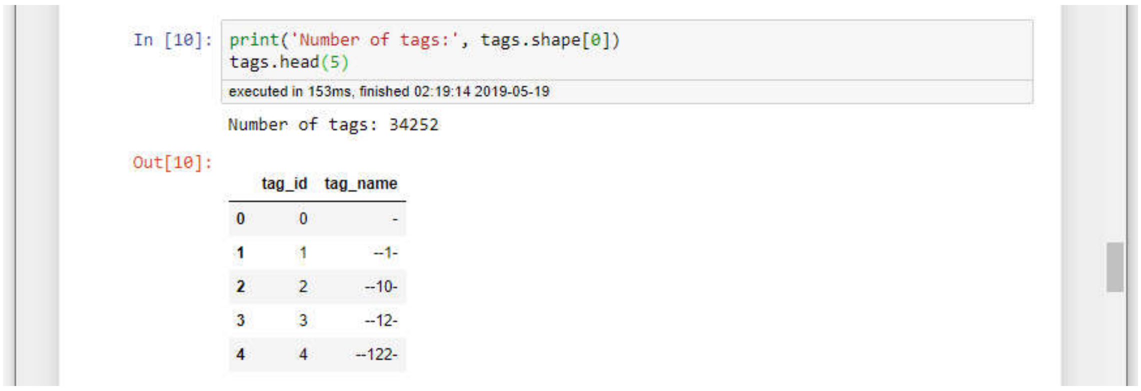


```
In [11]: print("Average rating given to books:", ratings.rating.mean())
         # Ratings distribution
         ratings.rating.hist(bins=5)
         executed in 479ms, finished 02:08:25 2019-05-19

         Average rating given to books: 3.9198655261735214

Out[11]: <matplotlib.axes._subplots.AxesSubplot at 0x22aeb56c630>
```

Figure 10. Distribution of the ratings in the dataset.

It can be observed from Figure 10 that the ratings users give in this dataset is at the higher end, which was expected as this dataset contains only popular books. It then becomes more justified that an item was considered relevant only if a user gives it a 5-rating. It is worth noting that this is not the case for every dataset. Different datasets with different distributions have different requirements on how items are considered relevant.

Figure 11 shows the first five tag entries in the tags dataset.



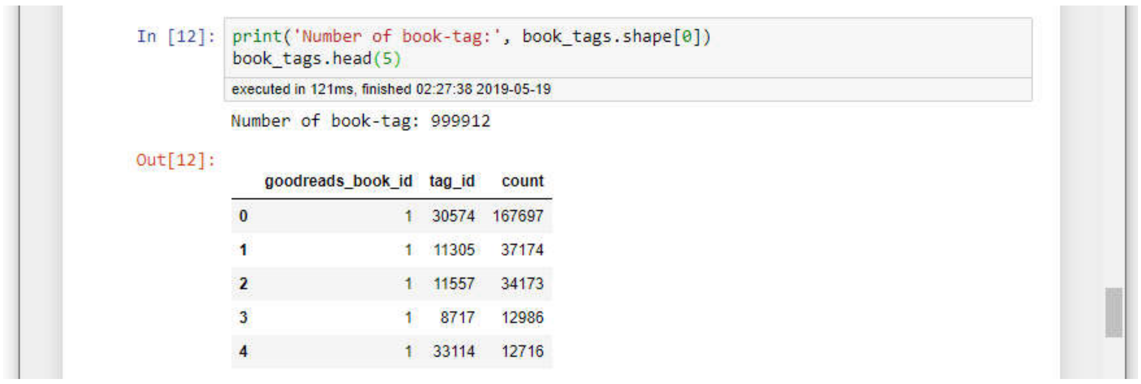
```
In [10]: print('Number of tags:', tags.shape[0])
         tags.head(5)
         executed in 153ms, finished 02:19:14 2019-05-19

         Number of tags: 34252

Out[10]:
```

	tag_id	tag_name
0	0	-
1	1	--1-
2	2	--10-
3	3	--12-
4	4	--122-

Figure 11. First five tag entries.

It can be safely concluded from Figure 11 that the `tags` dataset needs to be filtered. Tag names such as `-1-` and `-10-` do not make much intuitive sense, and it is reasonable to remove them from the dataset.

Figure 12 shows the first five book-tag entries.



```
In [12]: print('Number of book-tag:', book_tags.shape[0])
         book_tags.head(5)
executed in 121ms, finished 02:27:38 2019-05-19
Number of book-tag: 999912
Out[12]:
```

	goodreads_book_id	tag_id	count
0	1	30574	167697
1	1	11305	37174
2	1	11557	34173
3	1	8717	12986
4	1	33114	12716

Figure 12. First five book-tag entries.

It is worth noting in Figure 12 that the `book_tags` dataset uses the `goodreads_book_id` as the book identifier instead of `book_id`, which is why column `goodreads_book_id` of the `books` dataset was considered usable previously.

3.4.3 Partitioning the data into a training set and a testing set

The partitioning of data is done according to Section 3.3. A helper function `train_test_split` was imported from Scikit-learn to simplify the process. Because one-third of the 5-ratings in the dataset was extracted for testing, it is reasonable to remove users with two or less 5-ratings in their profiles, since in those cases there are not enough 5-ratings to divide to the testing set, rendering the evaluation process meaningless.

```
only_five_ratings = ratings[ratings.rating==5]
user_five_rating_counts = only_five_ratings.groupby('user_id').rating.count()
kept_users = user_five_rating_counts[user_five_rating_counts>2].index.tolist()
kept_ratings = ratings[ratings.user_id.isin(kept_users)]
```

Listing 3. Process of removing users with two or less 5-ratings

As shown in Listing 3, the `ratings` dataset was filtered to `kept_ratings`, which contains only ratings for users who have three or more 5-ratings.

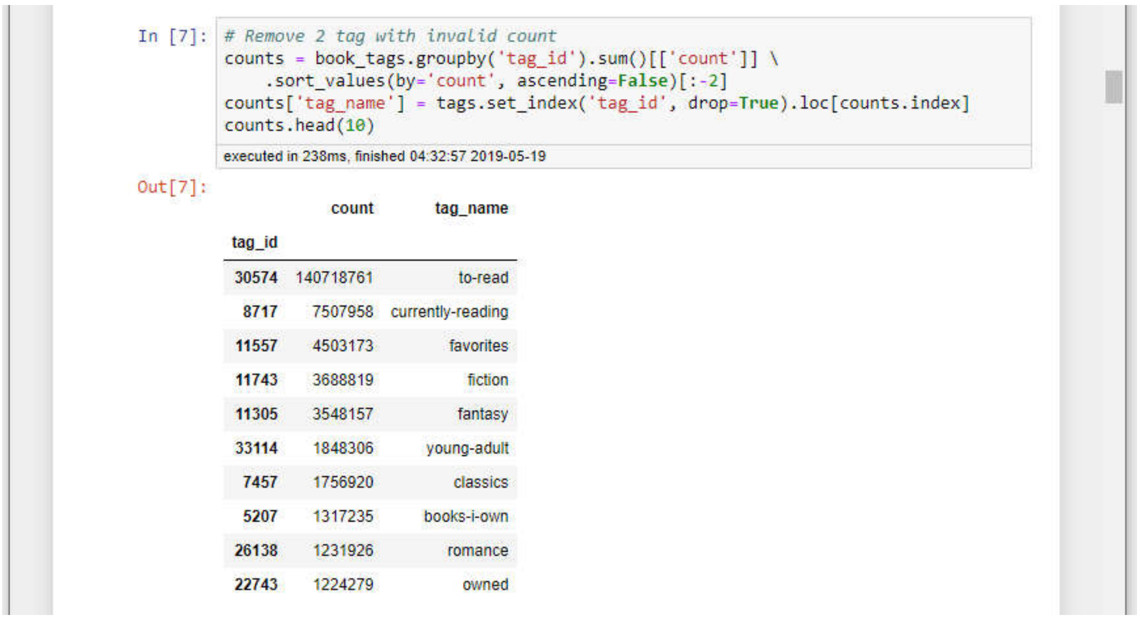
```
kept_4_and_less_ratings = kept_ratings[kept_ratings.rating<5]
kept_5_ratings = kept_ratings[kept_ratings.rating==5]
training_5_ratings, testing_ratings = train_test_split(
      kept_5_ratings,
      test_size=1/3,
      random_state=42,
      stratify=kept_5_ratings[['user_id']])
training_ratings = pd.concat([kept_4_and_less_ratings, training_5_ratings])
```

Listing 4. Routine of partitioning the `ratings` dataset into `training_ratings` and `testing_ratings`.

The `random_state` parameter in Listing 4 is to ensure the deterministic nature of the project, for it would be troubling to reach a conclusion if randomness was involved in the experimentation. The partitioning was stratified on users to ensure that there was no case that a user does not have a testing 5-rating.

3.4.4 Pre-processing tag-related data and making a content-centric similarity function

In this thesis project, tags and authors were considered content-centric keywords. As mentioned in Section 3.4.2, the `tags` dataset was filtered based on the ranking of the aggregated count each tag has. Figure 13 illustrates how the ranking of tags is determined.


In [7]:
```
# Remove 2 tag with invalid count
counts = book_tags.groupby('tag_id').sum()[['count']] \
    .sort_values(by='count', ascending=False)[:-2]
counts['tag_name'] = tags.set_index('tag_id', drop=True).loc[counts.index]
counts.head(10)
```
executed in 238ms, finished 04:32:57 2019-05-19

Out[7]:

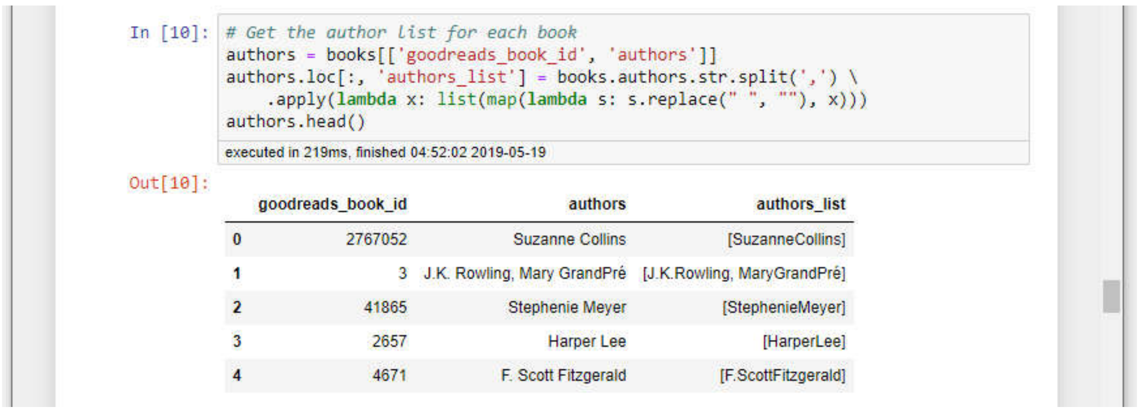
tag_id	count	tag_name
30574	140718761	to-read
8717	7507958	currently-reading
11557	4503173	favorites
11743	3688819	fiction
11305	3548157	fantasy
33114	1848306	young-adult
7457	1756920	classics
5207	1317235	books-i-own
26138	1231926	romance
22743	1224279	owned

Figure 13. Ranking of tags.

It can be observed from Figure 13 that popular tags have more recognizable names than the example tags shown in Section 3.4.2. An arbitrary number of 300 was chosen as the number of most popular tags to be kept for building the similarity function.

However, tags such as `to-read`, `currently-reading`, or `books-i-own` hardly have any descriptive value in portraying qualities of books. From the 300 tags kept, 80 tags were then manually removed based on intuitive common sense. The remaining 220 tags were then used to filter the `book_tags` dataset, and the counts were normalized. The processed `book_tags` dataset was then combined with the author data to form the content-centric keyword dataset for building the similarity function.

Before the union of the book-tag data and author data is proceeded, the `authors` column must be pre-processed into another suitable data structure, as mentioned in Section 3.4.2.



In [10]:
```
# Get the author list for each book
authors = books[['goodreads_book_id', 'authors']]
authors.loc[:, 'authors_list'] = books.authors.str.split(',') \
    .apply(lambda x: list(map(lambda s: s.replace(" ", ""), x)))
authors.head()
```
executed in 219ms, finished 04:52:02 2019-05-19

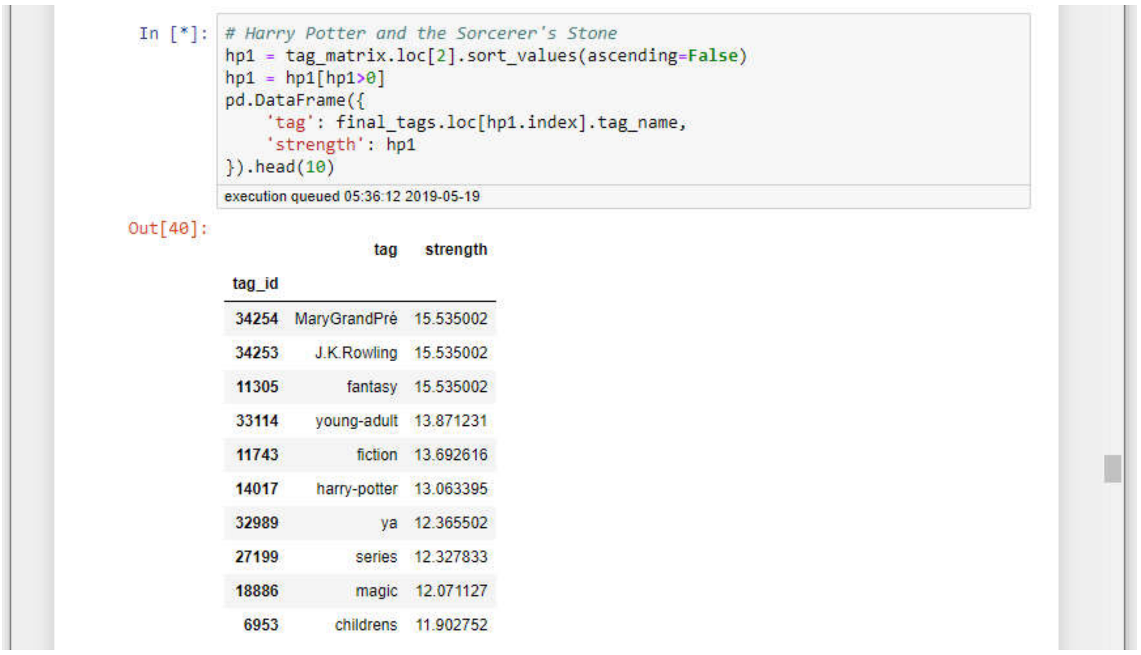
Out[10]:

	goodreads_book_id	authors	authors_list
0	2767052	Suzanne Collins	[SuzanneCollins]
1	3	J.K. Rowling, Mary GrandPré	[J.K.Rowling, MaryGrandPré]
2	41865	Stephenie Meyer	[StephenieMeyer]
3	2657	Harper Lee	[HarperLee]
4	4671	F. Scott Fitzgerald	[F.ScottFitzgerald]

Figure 14. Routine of reformatting the `authors` column to type array.

As Figure 14 depicts, the new column `authors_list` is more suitable for comparing and processing than the old `authors` column.

The book authors were then concatenated into the `tags` dataset, with their normalized counts equal to the highest normalized counts of their books. It is reasonable to assume that multiple authors are considered to have the same importance, and that author tags are given the highest priority in computing similarities.



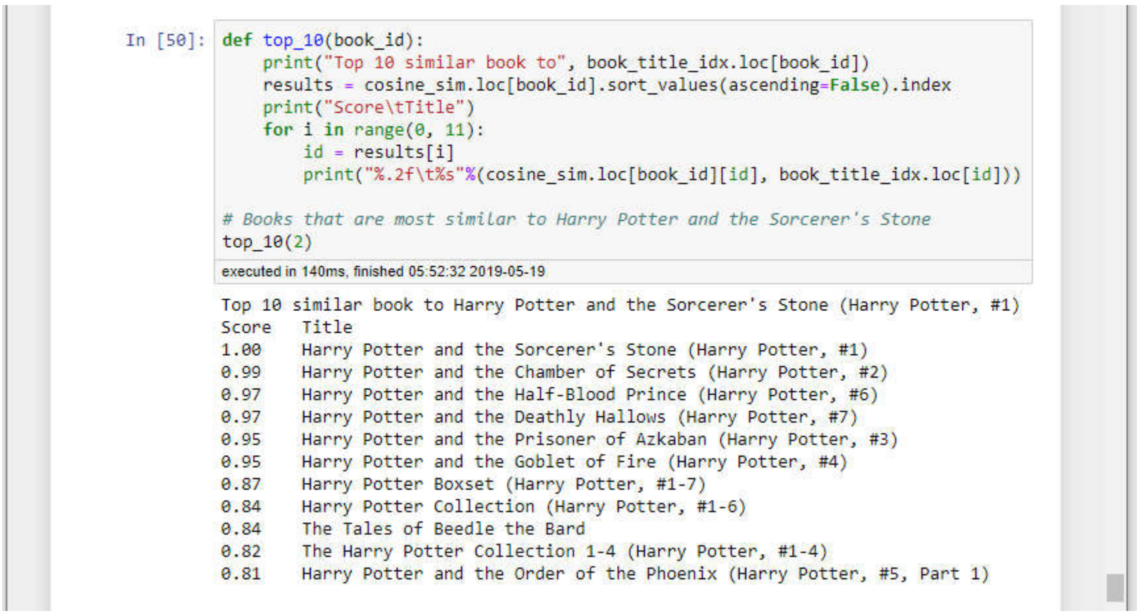
```
In [*]: # Harry Potter and the Sorcerer's Stone
        hp1 = tag_matrix.loc[2].sort_values(ascending=False)
        hp1 = hp1[hp1>0]
        pd.DataFrame({
            'tag': final_tags.loc[hp1.index].tag_name,
            'strength': hp1
        }).head(10)
        execution queued 05:36:12 2019-05-19
Out[40]:
```

tag_id	tag	strength
34254	MaryGrandPré	15.535002
34253	J.K.Rowling	15.535002
11305	fantasy	15.535002
33114	young-adult	13.871231
11743	fiction	13.692616
14017	harry-potter	13.063395
32989	ya	12.365502
27199	series	12.327833
18886	magic	12.071127
6953	childrens	11.902752

Figure 15. An example tag list after being processed and normalized.

Figure 15 illustrates the result tag list of the book Harry Potter and the Sorcerer’s Stone. It can be observed from the figure that the tags of the two authors are given equal importance to the fantasy tag, which is the most important tag excluding the author tags.

The similarity function was then formed by using the cosine function defined in Equation 10. Figure 16 shows the top 10 most similar books to the book Harry Potter and the Sorcerer’s Stone.



```
In [50]: def top_10(book_id):
             print("Top 10 similar book to", book_title_idx.loc[book_id])
             results = cosine_sim.loc[book_id].sort_values(ascending=False).index
             print("Score\tTitle")
             for i in range(0, 11):
                 id = results[i]
                 print("%.2f\t%s"%(cosine_sim.loc[book_id][id], book_title_idx.loc[id]))

         # Books that are most similar to Harry Potter and the Sorcerer's Stone
         top_10(2)
         executed in 140ms, finished 05:52:32 2019-05-19
```

```
Top 10 similar book to Harry Potter and the Sorcerer's Stone (Harry Potter, #1)
Score   Title
1.00    Harry Potter and the Sorcerer's Stone (Harry Potter, #1)
0.99    Harry Potter and the Chamber of Secrets (Harry Potter, #2)
0.97    Harry Potter and the Half-Blood Prince (Harry Potter, #6)
0.97    Harry Potter and the Deathly Hallows (Harry Potter, #7)
0.95    Harry Potter and the Prisoner of Azkaban (Harry Potter, #3)
0.95    Harry Potter and the Goblet of Fire (Harry Potter, #4)
0.87    Harry Potter Boxset (Harry Potter, #1-7)
0.84    Harry Potter Collection (Harry Potter, #1-6)
0.84    The Tales of Beedle the Bard
0.82    The Harry Potter Collection 1-4 (Harry Potter, #1-4)
0.81    Harry Potter and the Order of the Phoenix (Harry Potter, #5, Part 1)
```

Figure 16. Top 10 most similar books to the book Harry Potter and the Sorcerer’s Stone.

As Figure 16 illustrates, the results of the similarity function are intuitively reasonable. It is safe to conclude from Figure 16 that the similarity function worked as intended since it is normal that books of the same franchise (Harry Potter) are similar content-wise.

3.4.5 Building an evaluator

The evaluator was then built according to the evaluation strategy in Section 2.3. As the full version of the evaluator code is long, it is made available in the appendix section. The evaluator is a Python class called `Evaluator` that has a method to evaluate recommender systems and a method to print the results of the evaluation. The constructor of the evaluator takes N, the training set, the testing set, the content-centric similarity function, and the books’ novelty scores as the arguments.

```
evl = Evaluator(N=10,
                training_ratings=training_ratings,
                testing_ratings=testing_ratings,
                book_sim=book_sim,
                novelty_scores=novelty_scores)
```

Listing 5. Initialization of the evaluator.

It can be observed from Listing 5 that the evaluator is flexible in term of what value of N to use. Although the testing in this thesis was only done with $N = 10$, typically different values of N are used for a better view over the performance of tested recommender systems.

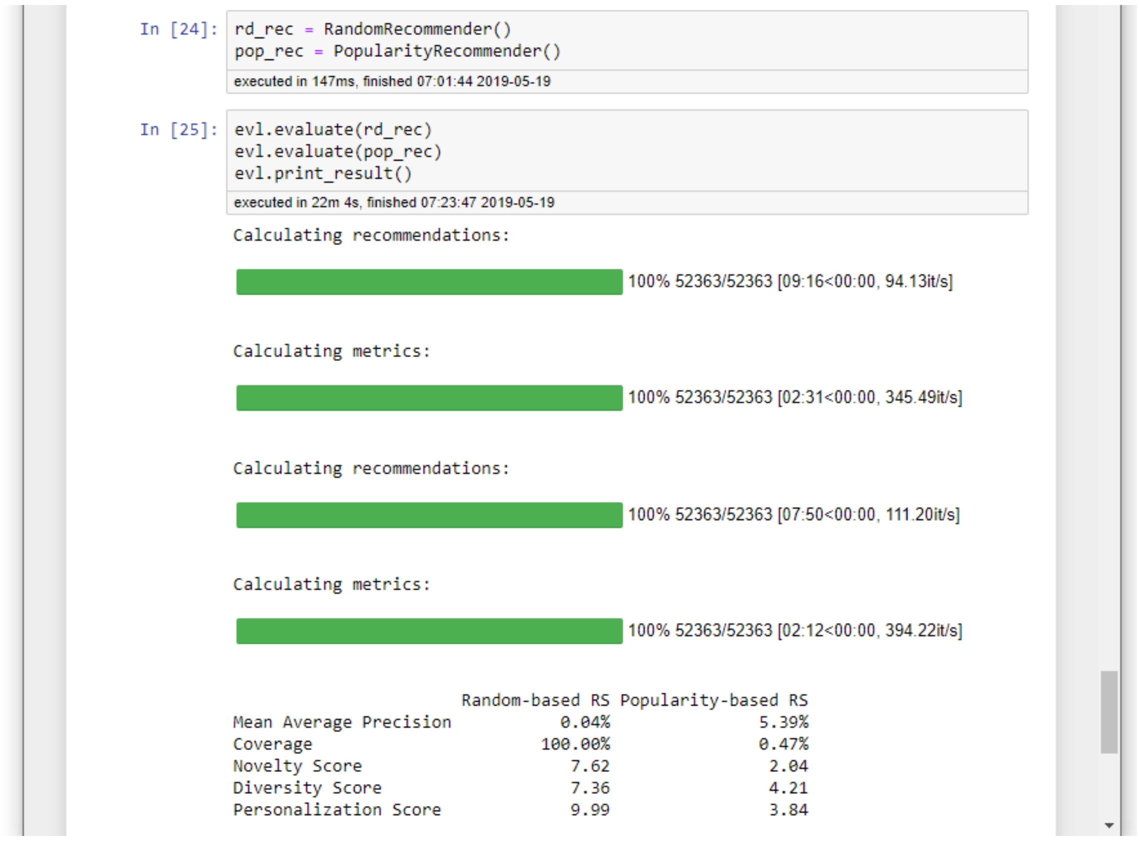
```
n_users = len(training_ratings.user_id.unique())
rating_count = training_ratings.groupby('book_id').count()[['rating']]
rating_count.loc[:, 'novelty_score'] = np.log2(n_users / rating_count.rating)
novelty_scores = rating_count[['novelty_score']]
```

Listing 6. Routine to create the novelty score dataset.

Listing 6 shows how Equation 7 was implemented in the project. From the dataset, the minimum and maximum values of novelty scores were determined as 1.48 and 12.68, respectively. For consistency, the NS metric was cropped to the range $[0, 10]$, which is also the range for DS and PS.

It is also worth noting that due to computational resource limitation, in this project, the pair-wise PS, defined in Equation 15, was only computed with an arbitrary value of 3000 as the number of users. A set of 3000 random users was chosen, and PS was calculated pair-wise between those 3000 users only. Because MPS is calculated by averaging PS across users, it is usually accepted that the user set is subsampled.

Figure 17 illustrates an example of an evaluating run using the evaluator.



```
In [24]: rd_rec = RandomRecommender()
         pop_rec = PopularityRecommender()
executed in 147ms, finished 07:01:44 2019-05-19

In [25]: evl.evaluate(rd_rec)
         evl.evaluate(pop_rec)
         evl.print_result()
executed in 22m 4s, finished 07:23:47 2019-05-19

Calculating recommendations:
100% 52363/52363 [09:16<00:00, 94.13it/s]

Calculating metrics:
100% 52363/52363 [02:31<00:00, 345.49it/s]

Calculating recommendations:
100% 52363/52363 [07:50<00:00, 111.20it/s]

Calculating metrics:
100% 52363/52363 [02:12<00:00, 394.22it/s]

                       Random-based RS Popularity-based RS
Mean Average Precision           0.04%               5.39%
Coverage                       100.00%               0.47%
Novelty Score                     7.62                2.04
Diversity Score                   7.36                4.21
Personalization Score             9.99                3.84
```

Figure 17. Evaluation of the random-based and popularity-based recommender system.

As depicted in Figure 17, the evaluator's methods were called to evaluate the random-based and popularity-based recommender systems. Then the recommender systems computed the recommendations for every user, and the evaluator computed the evaluation metrics.

3.4.6 Implementing recommender systems

Except for a deviation in the process of building the item-based collaborative filtering (IBCF) recommender system, the building of the five recommender systems was entirely based on the procedures and formulas of Section 2.4. The codes of building the recommender systems are available in the appendix section.

As mentioned in Section 2.4.4, the calculating of similarities between ten thousand items pair-wise following Equation 20 requires powerful computation resources, which this project did not possess. A simple workaround was attempted at solving this problem. By utilizing the sparse matrix data structure from SciPy and the `cosine_similarity` function from Scikit-learn, the similarity matrix is then calculable within a reasonable timeframe.

Firstly, the adjusted ratings for computing the adjusted cosine similarity were calculated. Then the adjusted ratings were transformed into a sparse rating matrix.

```
from scipy.sparse import csr_matrix
from pandas.api.types import CategoricalDtype

user_c = CategoricalDtype(sorted(training_ratings.user_id.unique()), or-
dered=True)
book_c = CategoricalDtype(sorted(training_ratings.book_id.unique()), or-
dered=True)

row = training_ratings.user_id.astype(user_c).cat.codes
col = training_ratings.book_id.astype(book_c).cat.codes
sparse_matrix = csr_matrix((training_ratings["adjusted_rating"],(row, col)), \
                    shape=(user_c.categories.size, book_c.categories.size))
```

Listing 7. Computation of the sparse adjusted rating matrix

In Listing 7, the sparse matrix proves to be a more efficient data structure, as the number of ratings is only about 5.2 million (because the testing set is excluded) compared to the dimension 52363x10000 of the matrix. The `cosine_simlarity` function was then applied directly to the sparse matrix. Because this function is optimized for sparse data structure, the computation is fast and does not take as much random-access memory compared to the dense data structure version.

However, by not following the original formula, this approach arises a new issue. The implementation of function `cosine_similarity` considers missing entries in the sparse matrix as 0. This imputation implies that the algorithm replaces every missing rating by the mean ratings of users. The consequence of this replacement is that the similarity values computed are skewed. To be precise, the algorithm is biased towards popular items, meaning that it is more likely to recommend popular items, which are usually not novel. It is unclear whether this bias is beneficial or detrimental, but it was expected that the novelty and diversity evaluation of the IBCF system built in this project is lower than that of the theory-based IBCF system.

4 Evaluations

The evaluations of the recommender systems are as follows:

	MAP	CC	MNS	MDS	MPS
Random-based	0.04%	100%	7.62	7.36	9.99
Popularity-based	5.39%	0.47%	2.04	4.21	3.84
Content-based	7.18%	80.10%	6.31	2.47	9.88
Item-based Collaborative Filtering (IBCF)	22.64%	85.54%	4.84	4.49	9.58
Hybrid ($w = 1$)	10.79%	81.78%	6.16	2.62	9.88

Table 1. Evaluations of the five recommender systems.

Table 1 illustrates the unique characteristics of different recommender systems built in this thesis project. The random-based system has remarkably low accuracy, but in exchange, it excels in all other areas of the evaluation. The popularity-based recommender system possesses a mediocre accuracy, and its recommendations are neither novel nor personalized. The content-based system also has an average accuracy, but its high tendency to recommend less popular items and high catalog coverage make it an available choice for businesses, especially in the scenario of cold-start. The IBCF system exhibits a far superior accuracy compared to the previous three, while its other areas are balanced as well. The representative hybrid system with weight 1 is a balance of the content-based system and the IBCF system, with evaluations settling in the middle of those from the two recommender systems.

5 Conclusions

From the numbers in Chapter 4, it is reasonable to state that the IBCF system is the most performant system on the goodbooks-10k dataset. With its exceptional accuracy, it is not surprising that the system is a target of interest for both academic research and practical implementation.

However, the focus of this thesis project was on the process of building and evaluating recommender systems. The evaluations in Chapter 4 are only valid in the context of this thesis and do not confirm which recommender systems are more useful. It is advisable that companies build and evaluate multiple recommender systems to determine which ones are the most suitable systems for their requirements.

It can be safely concluded that the process of building and evaluating recommender systems is highly customizable, meaning that the systems rely on many unique factors, such as the dataset, the business area, the users' expectation or the computational constraints. Decisions then must be made on the process of building the systems so that the systems resulted is satisfactory. This thesis project is an example of how the process of building recommender systems can be performed. The arbitrary values chosen in this thesis, such as $N = 10$ or $number\ of\ tags = 300$, represent decisions that need to be made. It is advisable that these decisions be made carefully after extensive experimentation.

It is also worth noting the technical implementations of recommender systems can meet unexpected computational constraints. There is not always a direct way from an algorithm to an implementation of the said algorithm. Two examples in this thesis are Equation 15 and Equation 20. For datasets with millions of users, which is common, the two formulas are highly likely to be unusable. It is then up to the business to find workarounds or alternative solutions. It is advisable for businesses to plan the technical implementations along with the chosen theoretical algorithms.

References

1 Linden G, Smith B, York J. Amazon.com Recommendations: Item-to-Item Collaborative Filtering. IEEE Internet Computing 2003;January-February:76-80.

2 Schafer JB, Konstan J, Riedl J. Recommender Systems in E-Commerce. EC '99 Proceedings of the 1st ACM conference on Electronic commerce 1999:158-166.

3 Netflix. The Netflix Prize Competition [online]. URL: https://www.netflix-prize.com/index.html. Accessed 20 May 2019.

4 Bennett J, Lanning S. The Netflix Prize. In KDD Cup and Workshop in conjunction with KDD 2007.

5 Davidson J, Nandy P, Liebald B, Van Vleet T, Liu J. The YouTube Video Recommendation System. Proceedings of the fourth ACM conference on Recommender systems 2010:293-296.

6 Spotify. Company Info [online]. URL: https://newsroom.spotify.com/company-info/. Accessed 20 May 2019.

7 Aggarwal CC. Recommender Systems: The Textbook. Springer International Publishing; 2016.

8 Zając Z. The goodbooks-10k dataset [online]. URL: https://github.com/zygmuntz/goodbooks-10k. Accessed 20 May 2019.

9 Zhou T, Kuscsik Z, Liu JG, Medo M, Wakeling JR, Zhang YC. Solving the apparent diversity-accuracy dilemma of recommender systems. Proceedings of the National Academy of Sciences of the United States of America 107 2010:4511-4515.

10 Sarwar B, Karypis G, Konstan J, Reidl J. Item-Based Collaborative Filtering Recommendation Algorithms. WWW '01 Proceedings of the 10th international conference on World Wide Web 2001:285-295.

11 Pedregosa F, Varoquaux G, Gramfort A, Michel V, Thirion B, Grisel O, Blondel M, Prettenhofer P, Weiss R, Dubourg V, Vanderplas J, Passos A, Cournapeau D, Brucher M, Perrot M, Duchesnay E. Scikit-learn: Machine Learning in Python. The Journal of Machine Learning Research 2011;12:2825-2830.

12 Python Software Foundation. The Python Tutorial [online]. URL: https://docs.python.org/3/tutorial/index.html. Accessed 20 May 2019.

13 TIOBE Software BV. TIOBE Index for May 2019 [online]. URL: https://www.tiobe.com/tiobe-index/. Accessed 20 May 2019.

14 Carbonnelle P. PYPL PopularitY of Programming Language [online]. URL: http://pypl.github.io/PYPL.html. Accessed 20 May 2019.

15 RedMonk. The RedMonk Programming Language Rankings: January 2019 [online]. URL: https://redmonk.com/sogrady/2019/03/20/language-rankings-1-19/. Accessed 20 May 2019.

16 Python Software Foundation. About Python [online]. URL: https://www.python.org/about/. Accessed 20 May 2019.

17 Project Jupyter. The Jupyter Notebook [online]. URL: https://jupyter-notebook.readthedocs.io/en/latest/notebook.html. Accessed 20 May 2019.

18 The Pandas Project. Package overview [online]. URL: http://pandas.pydata.org/pandas-docs/stable/getting_started/overview.html. Accessed 20 May 2019.

19 NumPy developers. NumPy [online]. URL: https://www.numpy.org/. Accessed 20 May 2019.

20 The SciPy community. Introduction [online]. URL: https://docs.scipy.org/doc/scipy/reference/tutorial/general.html. Accessed 20 May 2019.

21 Project Scikit-learn. Scikit-learn: Machine Learning in Python [online]. URL: https://scikit-learn.org/stable/index.html. Accessed 20 May 2019.

Appendix 1. Implementation of the evaluator

```
import numpy as np
import pandas as pd
import scipy.sparse as sp
from sklearn.metrics.pairwise import cosine_similarity
from tqdm import tqdm_notebook as tqdm


class Evaluator():
    def __init__(
        self,
        k=10,
        training_ratings=None,
        testing_ratings=None,
        book_sim=None,
        novelty_scores=None
    ):
        self.k = k
        self.book_sim = book_sim
        self.novelty_scores = novelty_scores
        if training_ratings is not None:
            self.training_ratings = training_ratings
            self.num_users = len(self.training_ratings.user_id.unique())
            self.num_books = len(self.training_ratings.book_id.unique())
        if testing_ratings is not None:
            self.testing_ratings = testing_ratings
            self.testing_idx = {}
            for user_id in tqdm(testing_ratings.user_id.unique()):
                self.testing_idx[user_id] = \
                    testing_ratings[testing_ratings.user_id == user_id]\
                    .book_id.values
        self.result = {}

    def _average_precision(self, pred, truth):
        in_arr = np.in1d(pred, truth)
        score = 0.0
        num_hits = 0.0
        for idx, correct in enumerate(in_arr):
            if correct:
                num_hits += 1
                score += num_hits / (idx + 1)
        return score / min(len(truth), self.k)

    def _novelty_score(self, pred):
        # Recommend the top 10 books in novelty score results in ~11
        # Crop the score to 10.0 for consistency
        return min(self.novelty_scores.loc[pred].novelty_score.mean(), 10.0)

    def _diversity_score(self, pred):
        matrix = self.book_sim.loc[pred, pred].values
        ils = matrix[np.triu_indices(len(pred), k=1)].mean()
        return (1 - ils) * 10

    def _personalization_score(self, preds, user_ids, book_ids):
        if len(user_ids) > 3000:
            np.random.seed(42)
            user_ids = np.random.permutation(user_ids)[:3000]
        df = pd.DataFrame(
            data=np.zeros([len(user_ids), len(book_ids)]),
            index=user_ids,
            columns=book_ids
        )
        for user_id in user_ids:
            df.loc[user_id, preds[user_id]] = 1

        matrix = sp.csr_matrix(df.values)

        # calculate similarity for every user's recommendation list
        similarity = cosine_similarity(X=matrix, dense_output=False)

        # get indicies for upper right triangle w/o diagonal
        upper_right = np.triu_indices(similarity.shape[0], k=1)

        # calculate average similarity
        personalization = np.mean(similarity[upper_right])

        return (1 - personalization) * 10

    def evaluate(self, model):
        print("Calculating recommendations:")
        if len(model.preds) == 0:
            model.fit(self.training_ratings)
        preds = model.all_recommendation()
        user_ids = list(preds.keys())
        book_ids = np.unique(np.concatenate(list(preds.values())))
        ap_sum = 0
        nov_score_sum = 0
        div_score_sum = 0
        print("Calculating metrics:")
        for user_id in tqdm(preds.keys()):
            pred = preds[user_id]
            truth = self.testing_idx[user_id]
            ap_sum += self._average_precision(pred, truth)
            nov_score_sum += self._novelty_score(pred)
            div_score_sum += self._diversity_score(pred)

        self.result[model.name] = {}
        self.result[model.name]['Mean Average Precision'] = "%.2f%%" % (
            ap_sum / self.num_users * 100)
        self.result[model.name]['Coverage'] = "%.2f%%" % (
            len(book_ids) / self.num_books * 100)
        self.result[model.name]['Novelty Score'] = "%.2f" % (
            nov_score_sum / self.num_users)
        self.result[model.name]['Diversity Score'] = "%.2f" % (
            div_score_sum / self.num_users)
        self.result[model.name]['Personalization Score'] = "%.2f" % \
            self._personalization_score(preds, user_ids, book_ids)

    def print_result(self):
        print(pd.DataFrame(self.result).loc[[
               'Mean Average Precision',
               'Coverage',
               'Novelty Score',
               'Diversity Score',
               'Personalization Score']])
```

Appendix 2. Implementation of the random-based recommender system

```
class RandomRecommender():
    name = 'Random-based RS'

    def fit(self, training_ratings):
        user_ids = training_ratings.user_id.unique()
        book_ids = training_ratings.book_id.unique()
        self.preds = {}
        np.random.seed(42)
        for user_id in tqdm(user_ids):
            excluded_books = \
                training_ratings[training_ratings.user_id==user_id]\
                .book_id.unique().tolist()
            recommendable = book_ids[~np.in1d(book_ids, excluded_books)]
            self.preds[user_id] = np.random.permutation(recommendable)[:10]

    def recommendation_for_user(self, user_id):
        if user_id not in self.preds:
            return []
        return self.preds[user_id]

    def all_recommendation(self):
        return self.preds
```

Appendix 3. Implementation of the popularity-based recommender system

```
class PopularityRecommender():
    name = "Popularity-based RS"
    preds = {}

    def fit(self, training_ratings):
        user_ids = training_ratings.user_id.unique().tolist()
        five_ratings = training_ratings[training_ratings.rating==5]
        ranked_books = five_ratings\
            .groupby('book_id')\
            .count()[['rating']]\
            .rename(columns={'rating': 'weight'})
        ranked_books = ranked_books\
            .sort_values(by='weight', ascending=False)
        top_books = ranked_books[:200]
        book_ids = np.array(top_books.index.tolist())
        self.preds = {}
        for user_id in tqdm(user_ids):
            excluded_books = \
                training_ratings[training_ratings.user_id==user_id]\
                .book_id.unique().tolist()
            recommendable = book_ids[~np.in1d(book_ids, excluded_books)]
            self.preds[user_id] = recommendable[:10]

    def recommendation_for_user(self, user_id):
        return self.preds[user_id]

    def all_recommendation(self):
        return self.preds
```

Appendix 4. Implementation of the content-based recommender system

```
top_sim_books = {}
book_ids = book_sim.index
for book_id in tqdm(book_ids):
    top_sim_books[book_id] = book_sim.\
        loc[book_id].sort_values(ascending=False)[1:51]

list_of_5_ratings = training_ratings[training_ratings.rating==5]\
    .groupby('user_id')['book_id'].apply(list)

class ContentBasedRecommender():
    name = "Content-based RS"
    preds = {}

    def fit(self, training_ratings):
        user_ids = training_ratings.user_id.unique().tolist()
        self.preds = {}
        for user_id in tqdm(user_ids):
            excluded_books = \
                training_ratings[training_ratings.user_id==user_id]\
                .book_id.unique().tolist()
            most_similar_books = pd.Series([])
            for book_id in list_of_5_ratings[user_id]:
                most_similar_books = most_similar_books\
                    .append(top_sim_books[book_id])

            most_similar_books = \
                np.array(\
                    most_similar_books.groupby(most_similar_books.index)\
                    .sum().sort_values(ascending=False).index\
                )
            recommendable = \
                most_similar_books[~np.in1d(\
                                        most_similar_books,
                                        excluded_books\
                                    )]

            self.preds[user_id] = recommendable[:10]

    def recommendation_for_user(self, user_id):
        if user_id not in self.preds:
            return []
        return self.preds[user_id]

    def all_recommendation(self):
        return self.preds
```

Appendix 5. Implementation of the item-based collaborative filtering recommender system

```
top_sim_books = {}
book_ids = cf_sim.index
for book_id in tqdm(book_ids):
    top_sim_books[book_id] = cf_sim\
        .loc[book_id].sort_values(ascending=False)[1:51]

list_of_5_ratings = training_ratings[training_ratings.rating==5]\
    .groupby('user_id')['book_id'].apply(list)

class ItemCFRecommender:
    name = "Item-based CF RS"
    preds = {}

    def fit(self, training_ratings):
        user_ids = training_ratings.user_id.unique().tolist()
        self.preds = {}
        for user_id in tqdm(user_ids):
            excluded_books = \
                training_ratings[training_ratings.user_id==user_id]\
                .book_id.unique().tolist()
            most_similar_books = pd.Series([])
            for book_id in list_of_5_ratings[user_id]:
                most_similar_books = most_similar_books\
                    .append(top_sim_books[book_id])

            most_similar_books = \
                np.array(\
                    most_similar_books.groupby(most_similar_books.index)\
                         .sum().sort_values(ascending=False).index
                )
            recommendable = most_similar_books[~np.in1d(\
                most_similar_books, excluded_books)]

            self.preds[user_id] = recommendable[:10].tolist()
    def recommendation_for_user(self, user_id):
        if user_id not in self.preds:
            return []
        return self.preds[user_id]

    def all_recommendation(self):
        return self.preds
```

Appendix 6. Implementation of the hybrid recommender system

```
cf_top_sim_books = {}
book_ids = cf_sim.index
for book_id in tqdm(book_ids):
    cf_top_sim_books[book_id] = cf_sim\
        .loc[book_id].sort_values(ascending=False)[1:51]

cb_top_sim_books = {}
book_ids = book_sim.index
for book_id in tqdm(book_ids):
    cb_top_sim_books[book_id] = book_sim\
        .loc[book_id].sort_values(ascending=False)[1:51]

list_of_5_ratings = training_ratings[training_ratings.rating==5]\
    .groupby('user_id')['book_id'].apply(list)

class HybridRecommender:
    name = "Hybrid CF RS"
    preds = {}

    def __init__(self, rate=1):
        self.rate = rate
        self.name = "Hybrid CF RS (rate=" + str(rate) + ")"

    def fit(self, training_ratings):
        user_ids = training_ratings.user_id.unique().tolist()
        self.preds = {}
        for user_id in tqdm(user_ids):
            excluded_books = \
                training_ratings[training_ratings.user_id == user_id]\
                    .book_id.unique().tolist()
            most_similar_books = pd.Series([])
            for book_id in list_of_5_ratings[user_id]:
                most_similar_books = most_similar_books.append(
                    cb_top_sim_books[book_id])
                most_similar_books = most_similar_books.append(
                    cf_top_sim_books[book_id] * self.rate)

            most_similar_books = np.array(most_similar_books.groupby(
                most_similar_books.index).sum()\
                    .sort_values(ascending=False).index)
            recommendable = most_similar_books[~np.in1d(
                most_similar_books, excluded_books)]

            self.preds[user_id] = recommendable[:10].tolist()

    def recommendation_for_user(self, user_id):
        if user_id not in self.preds:
            return []
        return self.preds[user_id]

    def all_recommendation(self):
        return self.preds
```