Machine Learning Engineer Nanodegree

Capstone Project

Lars Erik Bolstad
December 3, 2018

Book Recommendation Engine

Project Overview

In this project I have implemented a recommendation engine for books based on a Kaggle dataset containing user ratings for 10,000 "popular" books from Goodreads. The dataset is available here: https://www.kaggle.com/zygmunt/goodbooks-10k/home, however I have used an updated version of the dataset with duplicates removed and many more ratings (around 6 million) retrieved from this source: https://github.com/zygmuntz/goodbooks-10k.

Recommendation engines, or *Recommender systems*, are widely deployed to offer users recommended content or products. Such systems broadly fall into two categories depending on the model and algorithms used: *Collaborative* and *Content-based* filtering [1]. Collaborative filtering models produce recommendations based on a user's past behaviour and preferences, as well as those of other users exhibiting similar preferences. Content-based models are based on characteristics of the product or content in question to recommend items with similar properties. Many *hybrid* recommender systems combine these two approaches in various ways.

The solution implemented in this project uses collaborative filtering to generate book recommendations. Given the ratings data in the dataset we generate a *utility matrix* (a matrix containing the actual or *imputed* rating for a given book by a particular user) and apply a *Singular Value Decomposition (SVD)* [2] algorithm to this matrix. The factor matrices produced by SVD are then multiplied to produce a *Prediction Matrix,* which we use to find similar users and produce recommendations. These steps are described in more detail below.

The solution consists of a number of Python scripts for generating the matrices, performing SVD with different parameters and calculating metric scores, and a web application that lets users interact with the Book recommender.

Problem Statement

Regardless of the model used a recommendation engine needs information about a given user's preferences in order to provide recommendations perceived as relevant and useful. Without such information we have what is known as the *cold-start problem*.

For users who have a Goodreads account we use Goodreads' APIs to retrieve their reading history and their ratings. Users who don't have a Goodreads account are presented with random books from the dataset and are asked to pick those they like.

Recommendations are based on finding similar users and recommending books that they have given a high rating. The dataset contains 10,000 books and approximately 6 million ratings by around 54,000 users. The 54,000 by 10,000 *utility matrix* generated from this data is at the outset almost empty: Around 95% of the rating values are missing, since users will generally only have read and rated a small subset of the books.

This is a typical challenge for a ratings-based recommender system, and we need to *impute* (fill in) the missing values before we can apply matrix factorization in the form of SVD. The choice of imputation strategy directly impacts the quality of the recommender system, as measured by the evaluation metric we will use:

Evaluation metrics

When implementing a recommender system we need some way of evaluating its performance. The goal here is to recommend books to users and one option could be to ask the users to rate the quality of the recommendations they get. Such an approach would however not be very useful. We need an evaluation metric that can be objectively measured and explained by the algorithms used.

Root Mean Squared Error, or **RMSE**, is commonly used as an evaluation metric in recommender systems and gives us an objective measure of the quality of the predictions matrix produced by multiplying the SVD factor matrices. RMSE is defined as the square root of the *mean-squared-error,* which we calculate by summarizing the squared difference between the *predicted* and *original* ratings, divided by the number of original ratings:

$$\mathrm{RMSE} = \sqrt{\frac{1}{n}\sum_{j=1}^{n}(y_j - \hat{y}_j)^2}$$

In this project I try different imputation strategies and apply SVD with a number of different *latent factors*. The RMSE score is calculated for each combination and the one with the lowest score is then used in the web application to produce recommendations.

In addition to measuring the error in the predicted values, a *scoring function* can be used to measure the quality of the recommendations. Common scoring functions are *Precision and Recall at k*. Given a set of k recommendations, *precision at k* is the percentage of those recommendations that are *relevant* to the user, where *relevant* is defined (in our case) as a book that the user has actually given a high rating. *Recall at k* is defined as the proportion (in our case) of highly rated books that are in the set of k recommendations.

Precision and Recall at k will be calculated for existing users using both the prediction matrix generated (the one with the lowest RMSE score) and the benchmark matrix, described later in this document.

For this calculation we will define *relevant* as an actual rating that is higher than the user's average rating.

The formulas for Precision and Recall at k are:

Precision@k = `(# of recommended items @k that are relevant) / (# of recommended items @k)`

Recall@k = `(# of recommended items @k that are relevant) / (total # of relevant items)`

Analysis

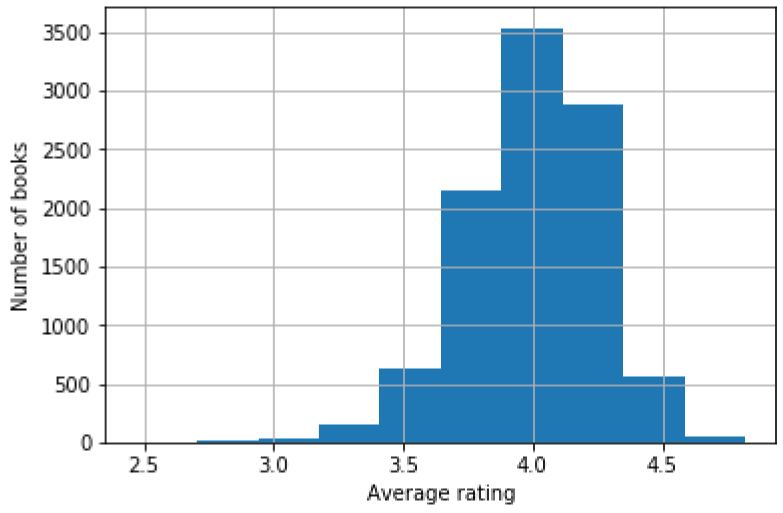
Data exploration

A summary of the data exploration is provided here. For more detail please see the *Data exploration.ipynb* notebook in the GitHub repository.

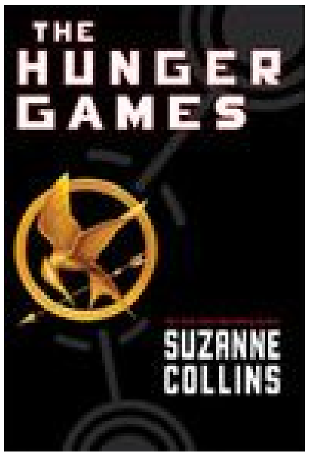
The dataset consist of the following csv files:
books.csv, ratings.csv, tags.csv, book_tags.csv, to_read.csv

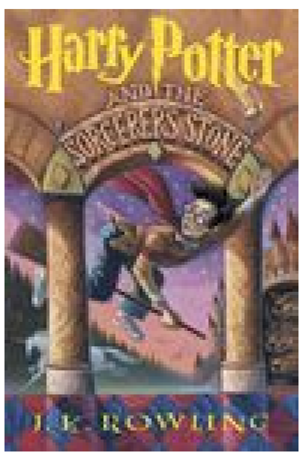
books.csv
This file contains data on 10,000 “popular” books from the Goodreads database, with the following columns:
'book_id', 'goodreads_book_id', 'best_book_id', 'work_id', 'books_count', 'isbn', 'isbn13', 'authors', 'original_publication_year', 'original_title', 'title', 'language_code', 'average_rating', 'ratings_count', 'work_ratings_count', 'work_text_reviews_count', 'ratings_1', 'ratings_2', 'ratings_3', 'ratings_4', 'ratings_5', 'image_url', 'small_image_url'

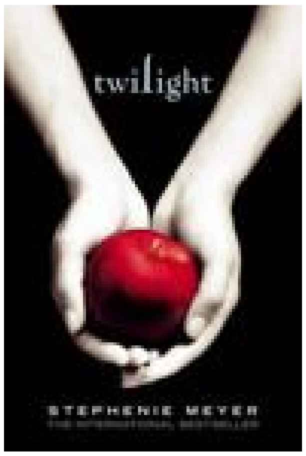
A few interesting characteristics of this dataset can be noted:
The overall average rating is 4.00, proving that we indeed have a selection of “popular” books. The distribution of average ratings is plotted below:

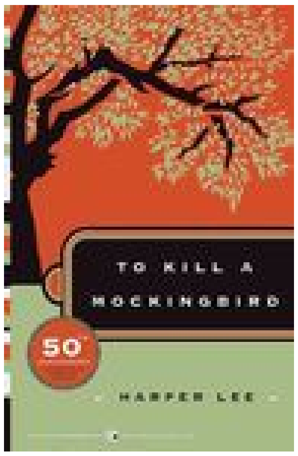


Another interesting characteristic to look at is the distribution of *ratings_count* values, i.e. the number of user ratings for each book. A few books stand out with a very high number of ratings (more than 3 million):

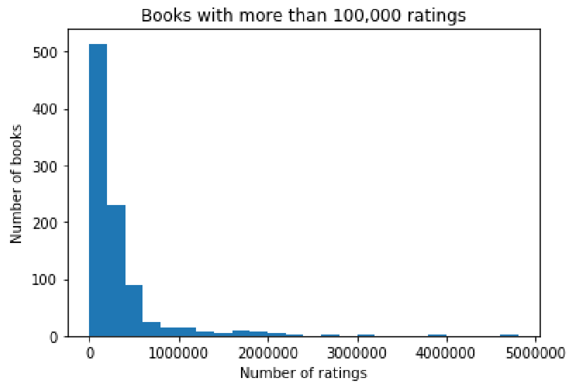


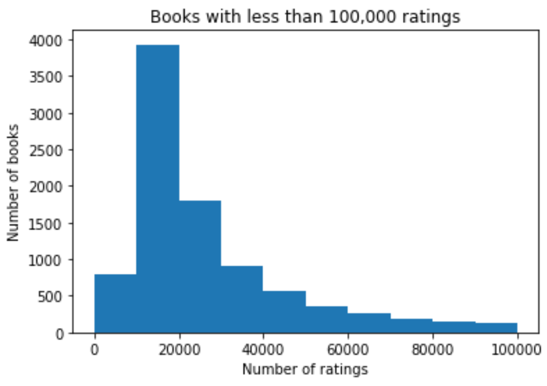






Among the 10,000 books there are 54 books with more than 1 million ratings, whereas the great majority of books have fewer than 100,000 ratings. Below are two plots of the distribution of values. One for those with more than 100,000 ratings and one for those with less than 100,000 ratings.

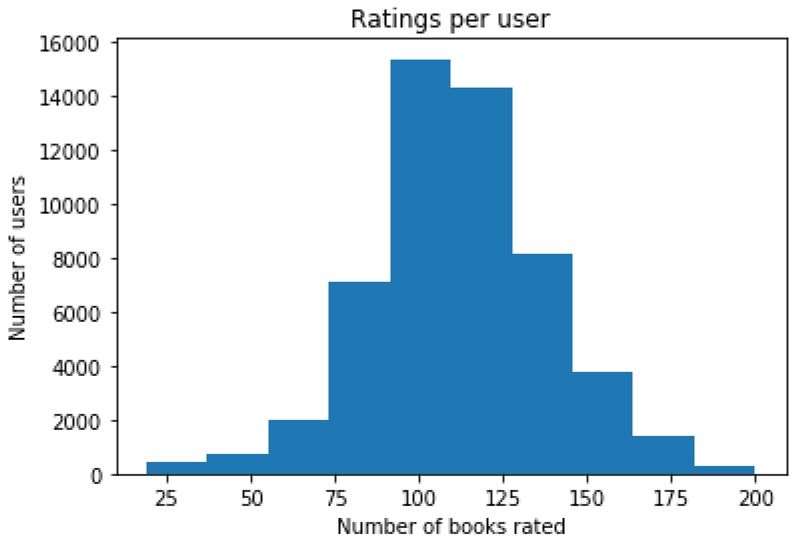




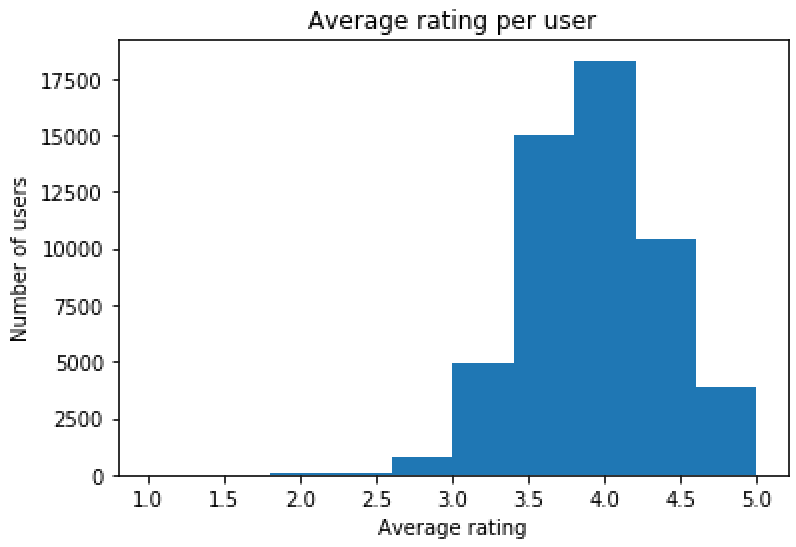
These numbers tell us a lot about the relative popularity of the books, but our dataset only contains a limited selection of these ratings.

ratings.csv

This file contains 5,976,479 user ratings on the format (user_id, book_id, rating), where the ratings are on a scale from 1 to 5. There are 53,424 unique user_ids, and the ratings cover 10,000 unique book_ids, meaning every single book in our dataset as at least one rating.
The plot below shows how the number of books rated per user looks to be normally distributed around an average value of 112.



We will later calculate a *user bias* when imputing missing values in the utility matrix, which will be based on how each user's average rating deviates from the overall average. The plot below shows the distribution in average rating per user.



While the majority of average ratings fall within a narrow band of values as expected, it is interesting to note how many users have an average rating near the top of the scale. Similarly there are some outliers in the other direction, with a small number of users who either read a lot of books they don't like, or simply apply the scale differently from most others.

tags.csv, book_tags.csv
tags.csv contains 34,252 user-defined tags that Goodreads users have applied to books. book_tags.csv contains 999,912 mapping of tags to books along with a 'count' for each indicating how many times the given tag is applied to the given book.

My initial plan was to try to make use of this data for content-filtering, i.e. to group books together based on how they are tagged by users. However, I found that this would require a lot of work *cleaning" the tags by removing all the ones that don't say anything about the book, e.g. format (where the user bought the book, when it was read, etc), duplicates and misspellings, non-English tags, and so on. In the end I dropped the whole idea and am therefore not making use of these two files in the project.

to_read.csv
This csv file contains 912,705 combinations of user_id and book_id and represents data from Goodreads' "To-read" shelf, which users can use to keep lists of books they intend to read.

I could have used this data as an additional data source when creating the utility matrix, however since these books are not rated by the users I chose not to use this data at all.

Methodology and algorithms

The end result of this project is a web application that offers a user book recommendations based on his or her reading history, or stated preferences when it comes to books if no reading history can be retrieved. In order to be able to provide recommendations we first need to generate what we will call a *Predictions Matrix*.

The project and the solutions developed therefore consist of two parts:

1. Creating the Predictions Matrix
2. Producing recommendations

Creating the Predictions Matrix

As the name implies, a Predictions Matrix contains a *predicted* rating for each combination of user and book. Once we have such a matrix, producing recommendations is a matter of identifying similar users based on their ratings and extracting books based on these users' ratings.

The first step is to create a *utility matrix,* which is the *m*x*n* matrix containing the ratings given by *m* users for *n* books. Given our dataset with around 6 million ratings, 10,000 books and 54,000 users, the resulting 54,000x10,000 matrix is extremely **sparse**, meaning most of the values are simply missing. In fact, we only have around 1% of the values that we need in the utility matrix!

As mentioned above we will use *Singular value decomposition,* a matrix factorization method, to generate the Predictions matrix, however SVD is not able to deal with missing values. We therefore need to fill in, or **impute**, the missing values before the SVD algorithm can be applied. There are many ways we can chose to do this, and the chosen strategy will directly impact the quality of the predictions as measured by our chosen metric, RMSE. The imputation step is in other words perhaps the most important step in the whole solution.

Once we have an imputed utility matrix we can apply SVD with a selected number of *latent factors*. Latent factors represent the implicit similarity between users and books they have rated. We will combine different imputation strategies with different numbers of latent factors, and for each combination generate a Predictions Matrix by multiplying the latent factor matrices.

We split the original *ratings.csv* into a **training set** and a **test set**, then apply imputation and SVD to the training set only. Once we have generated the Predictions Matrix we can calculate the RMSE score for the training set and test set separately. Finally we will pick the combination of imputation strategy and number of latent factors with the lowest RMSE score to use in the recommendation engine.

Benchmarking

The RMSE score calculated for each combination of imputation strategy and number of latent factors will tell us which performs best. In addition to this relative comparison we would like to have a **benchmark** score that can we can compare against.

I will use a utility matrix imputed with the average rating of each book and calculate benchmark scores that the other models will be compared with. A recommendation system based on this matrix would recommend exactly the same books to any user, and specifically the ones with the highest average ratings. As such it is not useful for that purpose, of course.

Producing recommendations

The Prediction Matrix contains a rating for each combination of user and book. If we have been able to achieve a low RMSE score the predictions should be close to the original ratings where present, and hopefully be close to how a given user would rate a given book in the cases where we didn’t have a rating.

The first step in producing recommendations for a user is to get a list of books that he or she has read, along with their ratings. SInce Goodreads offers a nice API we use this to retrieve the reading history for those users who have a Goodreads account. Those that don’t are asked to select a minimum number of books that they like from our dataset. In this case we will not have any user ratings.

The second step is to find the most similar users by calculating the dot product of the rating vectors for each users and picking the *n* users with highest value.

Once we have identified the most similar users we can extract the books that they have rated highly (above a given threshold) based on our Predictions Matrix, sort this list of books according to rating, subtract the list of books that the user has already read, and present the end result as a list of recommended books.

Methodology

This section describes in detail how the solution has been implemented, along with certain tradeoffs that had to be made.

Data preprocessing

No preprocessing of the data was necessary since I picked a cleaned version of the original Kaggle dataset. Specifically, duplicates had already been removed and the dataset does not contain any missing values that has any impact on the algorithms used.

Implementation

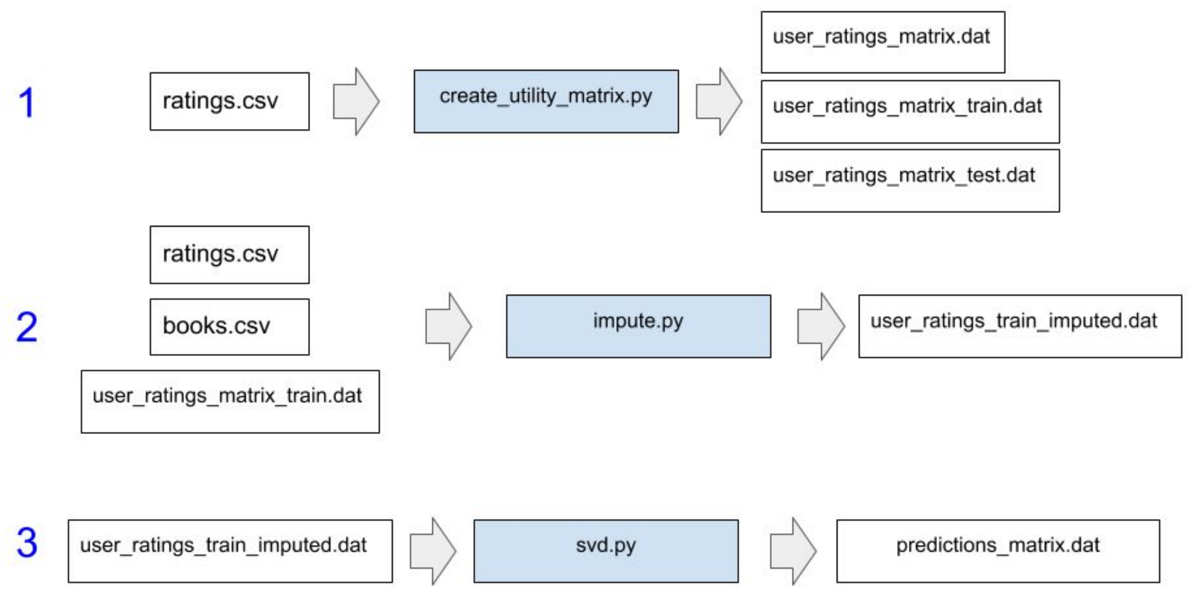
The implementation consists of two parts:

1. Python scripts used to generate the Predictions Matrix
2. A web application for interacting with the recommendation engine

Both the Python scripts and the web application can be found in the GitHub repository along with instructions on how to set them up and run them.

1. Generating the Predictions Matrix

The project was carried out using a fairly standard laptop with limited memory, and generating and processing 54,000 x 10,000 matrices tested the limits of this little laptop. This is the main reason why the steps to produce and score the Predictions Matrix are split into several Python scripts. Each matrix generated is written to file and read from file by the next script in the "pipeline":


1
ratings.csv
create_utility_matrix.py
user_ratings_matrix.dat
user_ratings_matrix_train.dat
user_ratings_matrix_test.dat
2
ratings.csv
books.csv
user_ratings_matrix_train.dat
impute.py
user_ratings_train_imputed.dat
3
user_ratings_train_imputed.dat
svd.py
predictions_matrix.dat

The **first** step is to create the utility_matrix, as well as a training matrix and a test matrix. This is done in **create_utility_matrix.py:**

Create_utility_matrix reads the ratings_csv file into a Pandas DataFrame and first generates the user ratings utility matrix, which is later used to find similar users. It then splits the ratings dataframe into a training set (75%) and a test set (25%) and generates a utility matrix for each of them, which is later used to calculate the RMSE scores. Note that each of these files end up being about 4.2 GB in size!

The **second** step is to impute the missing values in the training utility matrix, user_ratings_matrix_train. **impute.py** implements three different imputation strategies:

1. Average book rating
In each column, the missing values are replaced by the average rating for that book.

2. Average book rating plus the user bias
For each user we calculate the *user bias* as the difference between the user's average rating and the "global" average rating. In theory this gives us a better imputation value since users clearly tend to use the ratings scale differently.

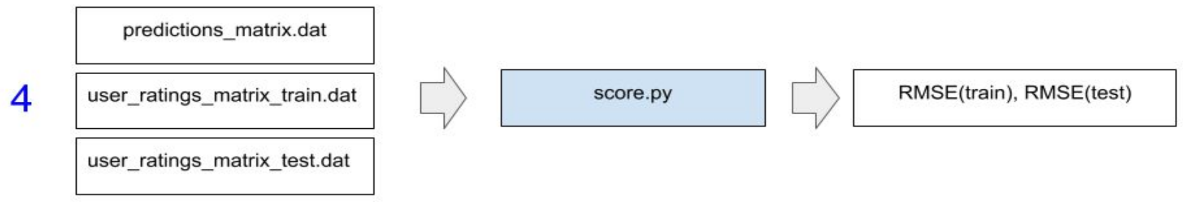
3. "Simon Funk's formula"
Simon Funk introduced a novel SVD algorithm in his entry for the Netflix Grand Prize back in 2006, as described here: https://sifter.org/~simon/journal/20061211.html. In this article he claims that imputing missing values based on the variance and the number of observed ratings is an

even better approach than the simple formula described under 3. Above. As the third imputation strategy I am using the calculated variance and his formula to impute missing values.

In the **third** step, **svd.py** applies SVD on the imputed utility matrix generated in step two. It was in this step that I ran into memory problems on my laptop when trying to use either the numpy.linalg.svd() or the scipy.linalg.svd() methods. To the rescue came *sparsesvd* [3], which my laptop was just about capable of applying to the matrix. SVD factorizes the utility matrix into three matrices, which are then used to calculate the Predictions Matrix.

Scoring and Results

For each imputation strategy we repeat steps 2 and 3 above to generate prediction matrices by applying SVD with different numbers of latent factors. **score.py** takes the generated prediction matrix and calculates the RMSE scores with respect to both the training matrix and the test matrix generated in step 1:

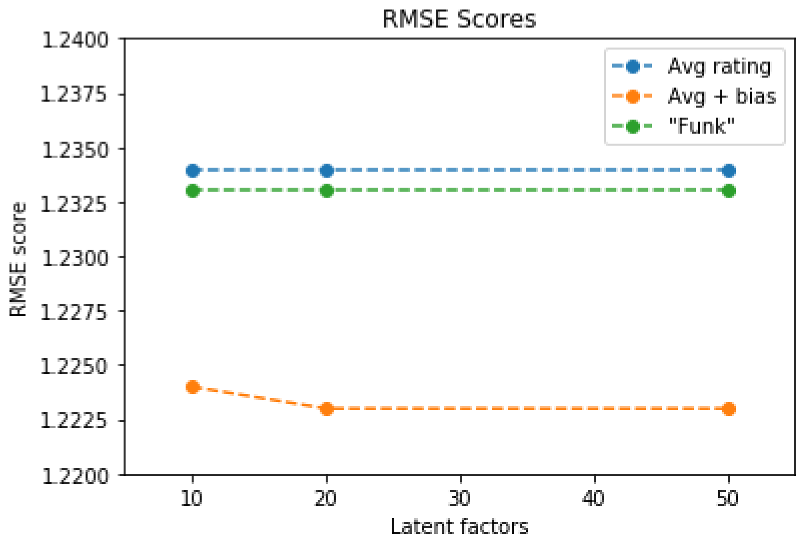


Because of long execution times I had to limit the number of combinations of imputation strategy and latent factors. Denoting the number of latent factors ***k*** I ran the imputation/svd/score pipeline with the three different imputation strategies and ***k*** = 10, 20 and 50.

RMSE scores

Imputation strategy	k = 10 Training	k = 20 Training	k = 50 Training	k = 10 Test	k = 20 Test	k = 50 Test
Avg rating	1.226	1.222	1.218	1.234	1.234	1.234
Avg+bias	1.218	1.213	1.207	1.224	1.223	1.223
“Funk”	1.218	1.216	1.210	1.233	1.233	1.233
Benchmark	**0.963**	**0.963**	**0.963**	**0.963**	**0.963**	**0.963**

These numbers show that we achieve the lowest RMSE on the **test set** using imputation strategy number 3 (book average + user bias) and 50 latent factors. The corresponding prediction matrix is therefore selected and used in the web application, described next.
The plot below shows only the RMSE scores for the test set, with the scores for the random imputations removed.



However, we’re getting nowhere near the RMSE score of the benchmark model! This score is calculated in exactly the same way, using a prediction matrix imputed with the average rating for each book.

Precision and Recall @k

While RMSE measures the error in the predictions matrix, Precision and Recall gives us a measure of the relevance of the recommendations. Even though the web application returns 100 recommendations I have set *k* to 50 to calculate these scores, to make sure all users included in the calculation have at least k ratings. The scores are calculated in **precisionrecall.py**, which generates "recommendations" based on each user's predicted ratings, or in the case of the benchmark mode, based on the average rating of each book. The following scores were calculated across the first 10,000 users in the dataset:

Precision and Recall Scores

	Benchmark	**Average+Bias model**
Precision @50	0.011	0.01
Recall @50	0.136	0.111

While none of these scores seem very impressive, at least our model performs better than the benchmark model when evaluating the quality of the recommendations. Given that users have on average rated 112 book out of the 10,000 in the dataset we probably should not expect very high precision and recall overall.

2. The web application

With the Predictions Matrix generated we can use this to produce book recommendations for users. A web application was implemented as part of the project to make the recommendation engine interactive and hopefully fun to use.

Ideally I would have deployed the web application somewhere that anyone could access it, but the size of the matrices represent a challenge. In order to first find similar users and then extract recommended books based on predicted ratings I need access to both the original (sparse) utility matrix generated from ratings.csv, as well as the predictions matrix generated by the Python scripts. Combined these two matrices are 8.4 GB in size and I therefore opted to run the web application locally using the *Flask* framework.

The web application basically consists of two files: rec.py and main.html.
rec.py implements the backend and contains the code for retrieving data from the Goodreads API, finding similar users based on the utility matrix, and picking recommendations based on the predictions matrix.

mail.html implements the front end, including the "client-side" JavaScript code that communicates with the backend using XmlHttpRequest() and leads the user through the required steps that lead to the book recommendations.

Once installed, the web application is available from this url: http://127.0.0.1:5000/rec/

The user is initially asked whether he/she has a Goodreads account. If yes the user is asked to input the user id and the user's reading history is retrieved using the Goodreads API. The user's average rating is calculated and applied to any books that the user has not rated.

If the user does not have a user account the application will present him/her with a random selection of books, 10 at a time, and ask the user to select the ones he/she likes. A minimum of 25 books have to be selected. In this case we don't have ratings, instead each book is given a rating of 1.

Both the previous paths lead to the next step, which is to find *n* similar users based on a vector multiplication of the user's rating list with each row in the utility matrix. Before proceeding to the final step, generating the list of recommended books, the application presents a slider that lets the user tune the recommendations towards "mainstream" books, or towards more "obscure" books.

The predictions matrix is then used to generate a list of books that have a rating above a threshold value (set to 4.0 by default). This list is then sorted by rating, and the books read (or selected) by the user is removed. Based on the slider setting the algorithm will also remove books with a high ratings_count, unless the user explicitly requests "mainstream" books.

Finally, the first 100 books in the list are presented as recommendations.

A number of screenshots of the web application are provided in the appendix.

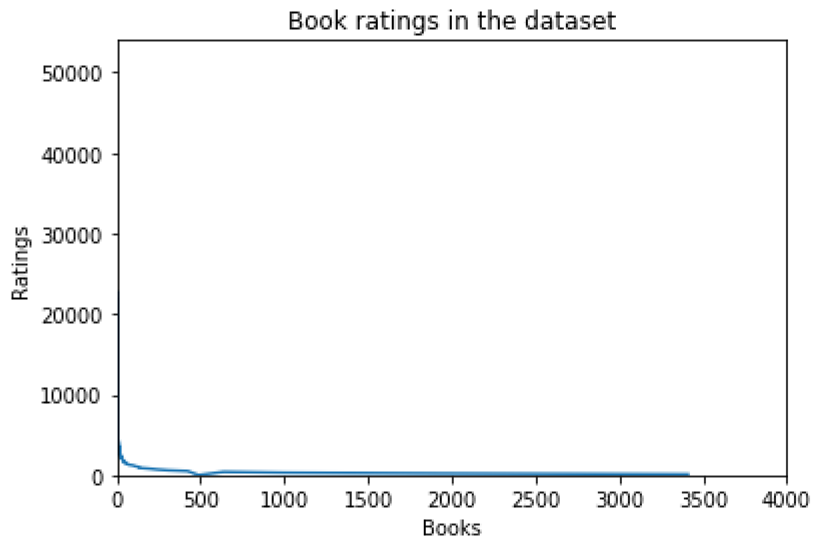
Conclusion

I chose this project and dataset both because I find recommender systems and the various methods that can be used to implement them interesting, but also because I love reading and happen to be a long-time Goodreads user. I'm always interested in book recommendations and building a book recommendation engine has been great fun!

Having read up on SVD and various approaches that can be taken to generate the predictions matrix I realize I have only scratched the surface in this project. Given more time (and computing resources) I would have liked to try out other methods to improve my RMSE score. For instance, in the project proposal I stated that I would evaluate content-based filtering as well. My plan was to cluster books based on a text analysis of the user-defined tags, but I found them to be a bit useless for the purpose. I did consider retrieving a synopsis of each book from the Goodreads API, or alternatively to use the ISBN data in the dataset to fetch additional data for each book from some other API provider, but I didn't find any that suited the purpose.

Once I had the web application up and running, producing recommendations based on my Goodreads reading history, it became clear that I had to add the ability for the user to “skew” the recommendations. I am personally not that interested in having *Harry Potter* or *Hunger Games* topping my list of recommendations, but this was perhaps the inevitable result given that so many users in the dataset have read these books and given them such a high rating.
I think the ability to filter out “mainstream” books from the recommendations greatly improved the perceived quality of this recommendation engine.

As a final note here is a small visualization that sums up the core challenge I faced in this project:



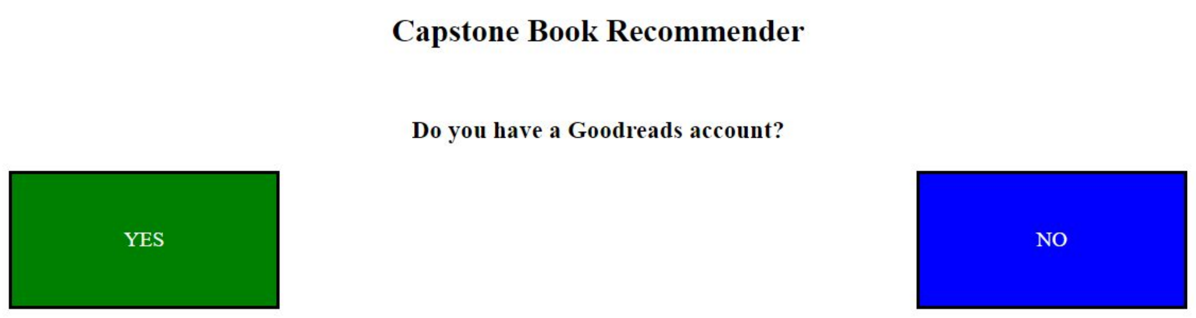
This illustrates the sparsity of the ratings data used to build the recommender engine. The curve shows the number of ratings per book. A small handful of books, quite intentionally barely visible on this scale, have a few thousand ratings. The big majority of books have much less.

[1] https://en.wikipedia.org/wiki/Recommender_system
[2] https://en.wikipedia.org/wiki/Singular_value_decomposition
[3] https://pypi.org/project/sparsesvd/

APPENDIX

Screenshot from the web application

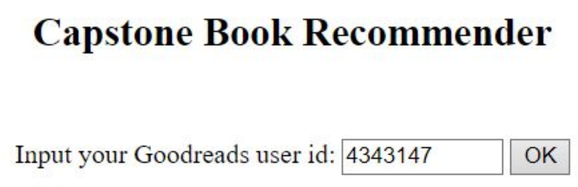
1. The start page



Capstone Book Recommender

Do you have a Goodreads account?

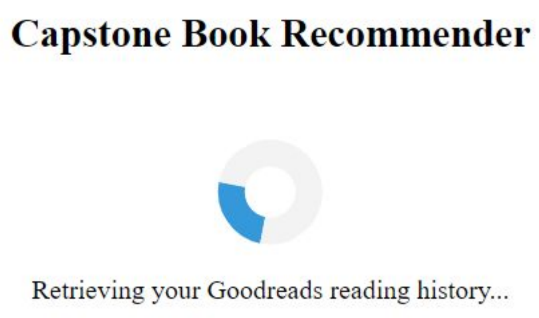
2. User selects “YES”



Capstone Book Recommender

Input your Goodreads user id: 4343147

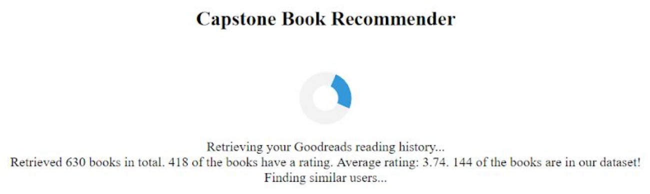
3. User inputs Goodreads user id and clicks “OK”



Capstone Book Recommender

Retrieving your Goodreads reading history...

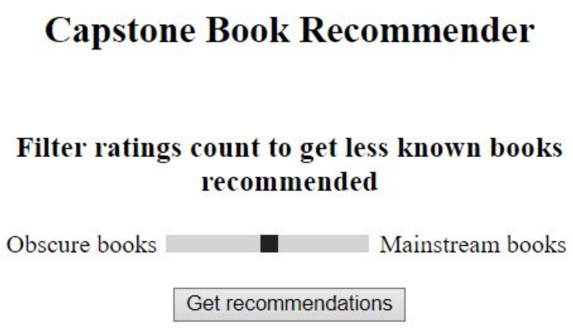
4. Done retrieving the reading history, outputting some numbers.



Capstone Book Recommender

Retrieving your Goodreads reading history...
Retrieved 630 books in total. 418 of the books have a rating. Average rating: 3.74. 144 of the books are in our dataset!
Finding similar users...

5. Presenting the slider

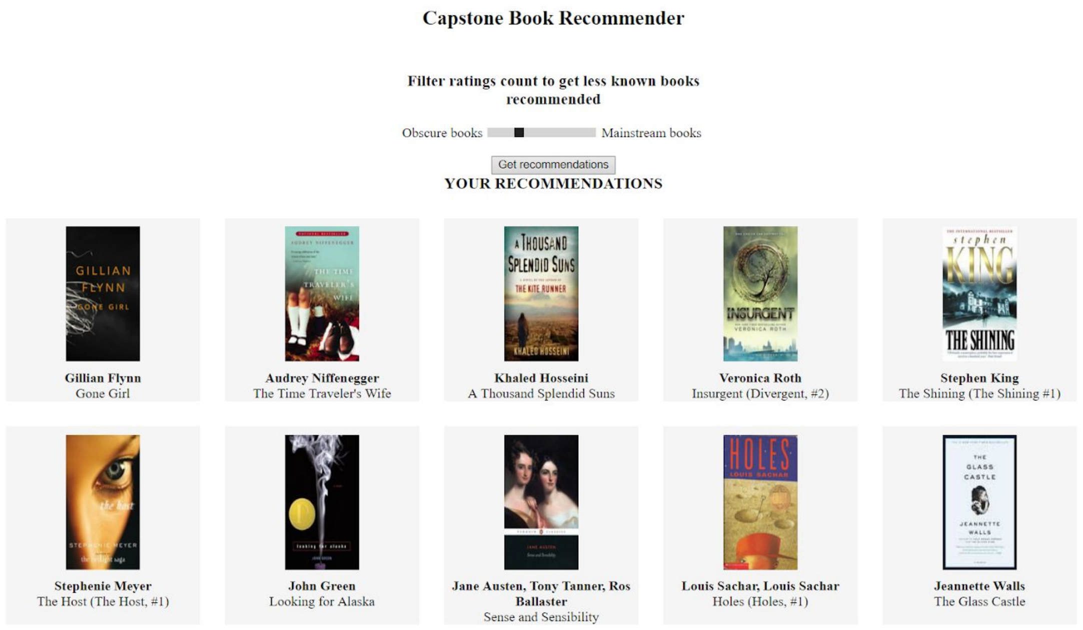


Capstone Book Recommender

Filter ratings count to get less known books recommended

Obscure books | Mainstream books

6. The recommendations!



Capstone Book Recommender

Filter ratings count to get less known books recommended

Obscure books | Mainstream books

YOUR RECOMMENDATIONS

Gillian Flynn
Gone Girl

Audrey Niffenegger
The Time Traveler's Wife

Khaled Hosseini
A Thousand Splendid Suns

Veronica Roth
Insurgent (Divergent, #2)

Stephen King
The Shining (The Shining #1)

Stephenie Meyer
The Host (The Host, #1)

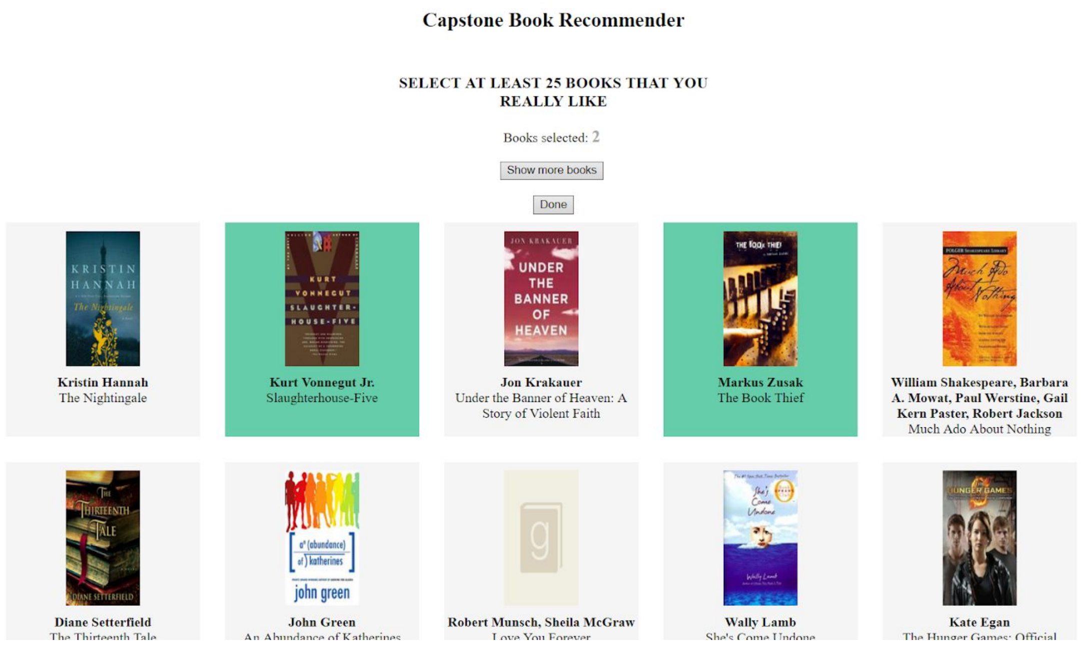
John Green
Looking for Alaska

Jane Austen, Tony Tanner, Ros Ballaster
Sense and Sensibility

Louis Sachar, Louis Sachar
Holes (Holes, #1)

Jeannette Walls
The Glass Castle

7. If the user does not have a Goodreads account this page is shown.



Capstone Book Recommender

SELECT AT LEAST 25 BOOKS THAT YOU REALLY LIKE

Books selected: 2

Kristin Hannah
The Nightingale

Kurt Vonnegut Jr.
Slaughterhouse-Five

Jon Krakauer
Under the Banner of Heaven: A Story of Violent Faith

Markus Zusak
The Book Thief

William Shakespeare, Barbara A. Mowat, Paul Werstine, Gail Kern Paster, Robert Jackson
Much Ado About Nothing

Diane Setterfield
The Thirteenth Tale

John Green
An Abundance of Katherines

Robert Munsch, Sheila McGraw
Love You Forever

Wally Lamb
She's Come Undone

Kate Egan
The Hunger Games: Official