Capstone 2 Project:

Building a Board Game Recommendation Engine

TLDR:

I designed a board game recommender system that asks a user for a list of games that they already like, and then recommends games based on their game preferences. The system was trained using ratings data from the BoardGameGeek website, and I tested three model variations. After tuning and selecting the best recommender model, I deployed it as a working web app.

Project Github: https://github.com/johnmburt/springboard/tree/master/capstone_2

Introduction

The board game market has experienced enormous growth in the last decade and is forecast to reach a global value of more than $12 billion by 2023. There are currently thousands of games on the market, and many different genres of games providing extremely diverse player experiences. From thematic zombie survival games to lightweight party games to light and heavy strategy games to wargames and more, there is most likely a game out there to fit the tastes of just about any person.

In fact, one of the problems that impedes the growth of the boardgaming hobby is that there is an embarrassment of choices: it can be hard for a given person to find the right game for them. Good board game stores hire experts who can recommend the games that customers might like, but many consumers don't live near a board game store, or prefer to shop online. This consumer pattern suggests that a digital solution in the form of a web-based board game recommender system could be a useful tool for consumers and online businesses.

For this project I developed a board game recommendation engine that can be deployed on a website for consumers shopping for board games. This recommender could be used by an online board game store such as Amazon or Coolstuffinc.com. The system asks a user to list some games that they like, as well as optional filter parameters such as preferred genre or game difficulty (weight). Then the recommender uses those preferences to search for and select a list of game titles to suggest.

The project had two phases: model testing and selection, and implementation and deployment of a web app that demonstrates the functionality of the recommender. For the model development phase, I created and tested two distinct models, then selected the best one to use for the web app.

Data preparation

For the project, I made use of the extensive data available from the board game hobbyist website boardgamegeek.com (BGG), which tracks millions of users and thousands of games and makes this data available via their web API. The principal dataset used is board game ratings logged by users (over 15 million ratings by about 70,000 users for 12,600 games). The BGG site also provides a rich set of metadata and comments that can be used to cluster games and users to improve recommendations.

Notebooks:

- Utility functions for all notebooks.
- Scraping game titles and IDs.
- Scraping game metadata.
- Scraping user ratings.
- Filtering users and generating a utility matrix.
- Filling in missing ratings using Alternating Least Squares matrix factorization.
- Generating Model 1 web app dataset
- Generating Model 2 web app dataset

Downloading and preparing the game data was a multi step process. Downloading was accomplished via the BGG API and web scraping packages requests and BeautifulSoup.

1) Download a list of all board games on BGG with >= 100 ratings.
2) Download metadata for all games in the board game list.
3) Download user ratings for all games in the board game list.
4) From ratings, create a game (item) x user rating utility matrix, with NaN values where users did not rate games. Users with fewer than 50 ratings were excluded, as were users who had overly uniform ratings.

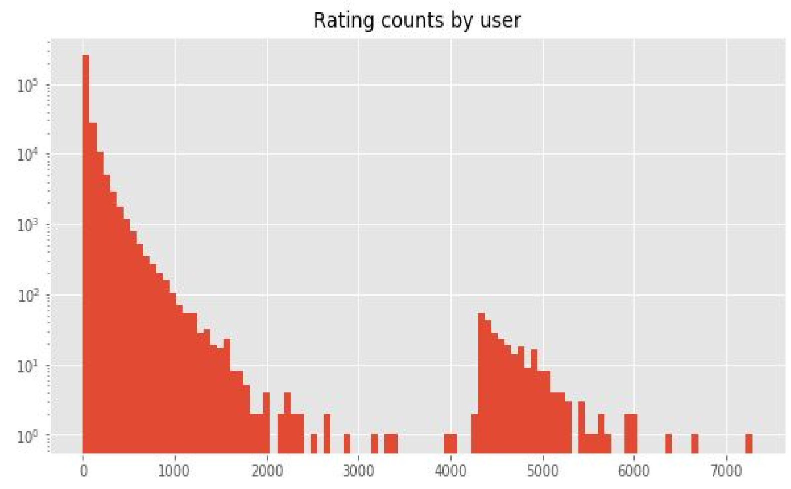
5) Apply Alternating Least Squares matrix factorization to create a set of latent game and user factors.These factors were used to fill in the missing ratings in the utility matrix.
6) For Recommender Model 1, additionally applied PCA to reduce user ratings for each game to a smaller set of game features to use as coordinates in a nearest neighbor game search. These coordinates were combined with game metadata to form a dataframe that could be used by the model for testing and in the web app.
7) For Recommender Model 2, I used the latent game and user factors from the matrix factorization, in addition to the game metadata.

Exploratory Data Analysis

Notebook: Exploratory Data Analysis

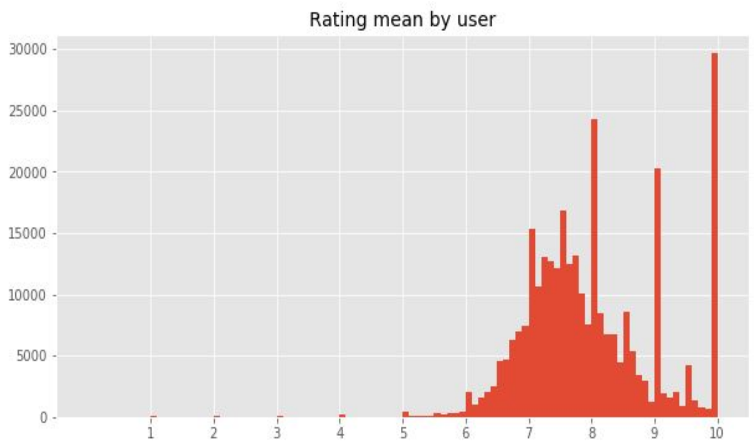
For my EDA, I examined the downloaded ratings, each of which is associated with a user, and a board game. I wanted to verify that the distributions of rating counts and the mean ratings for users and games were not anomalous. In fact, I found serious problems with roughly 25% of the samples.

User rating counts: Looking at rating counts for all users, I would expect to see a rapidly decreasing count frequency starting at one rating, with no discontinuities. I actually found that there are two different distributions, one starting at 1 as expected, and another starting at rating counts of about 4500. This suggests a problem with the data.



User rating means:

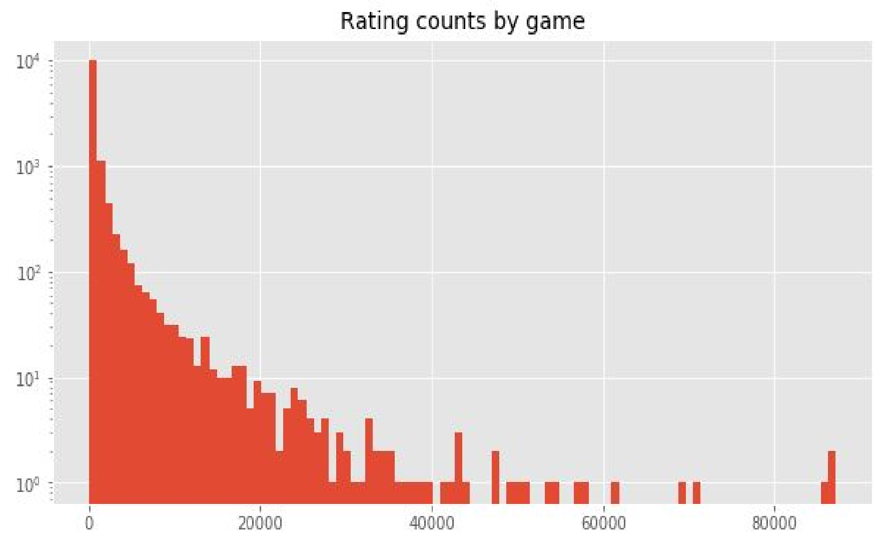
Looking at mean ratings for all users, I would expect a semi-normal distribution



centered on the average overall rating, and this is what I found. There are higher counts of mean ratings at whole number values because people tend to rate using whole numbers. 10-value ratings are common, since people often only rate games they really like. This looks OK to me.

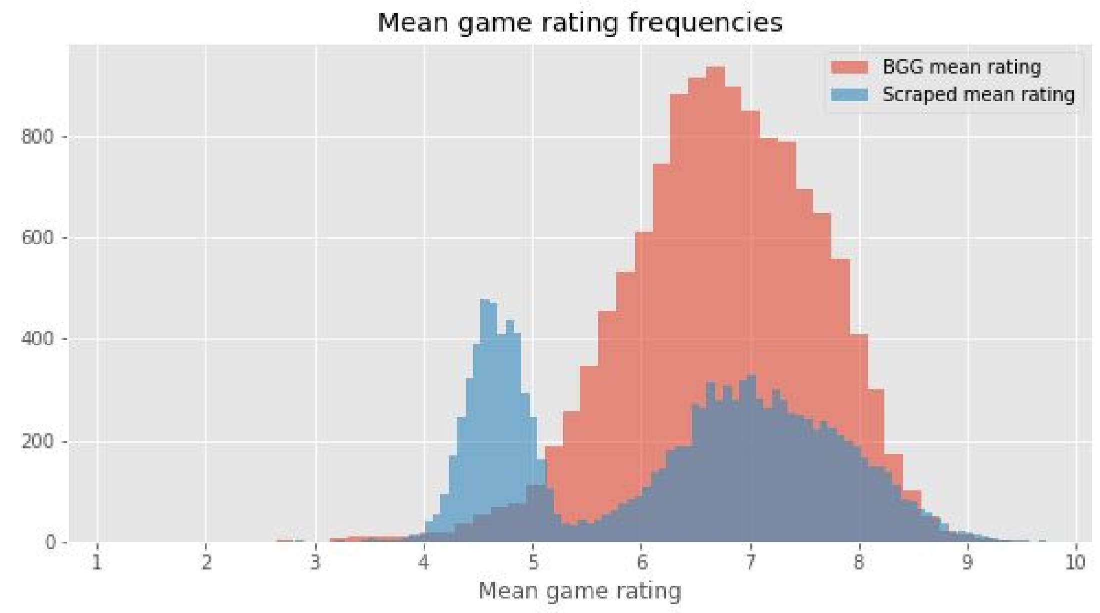
Game rating counts:

Looking at rating counts for games, I see nothing particularly out of place, except a few extremely high counts. These may be ratings for very popular games, such as Catan, which have been on the site for many years.



Game rating means:

The distribution of mean ratings for games reveals two very distinct peaks. This is very different from the distribution of the rating mean provided by Boardgamegeek as metadata. It appears that some of the games have ratings that match the Boardgamegeek average, while for other games the ratings are systematically off.



Testing similarity of distributions

Are either of the two mean game rating distributions (the upper vs. lower peaks) from scraped data similar to the BGG ratings, which I assume are accurate? I used a Kolmogorov-Smirnov test on two samples. The null hypothesis is that the samples are drawn from the same distribution. For each distribution, I randomly sampled 500 points. Significance criterion was 0.05. I split the scraped mean game ratings using a threshold of 5.5: games with mean ratings below 5.5 were assigned to the “lower” set, while those above were assigned to “upper”.

Results:

	Mean	**Stdev**
BGG mean game ratings	6.721	0.837
Scraped upper distribution game ratings	7.208	0.765
Scraped lower distribution game ratings	4.682	0.307

When samples from the upper and lower scraped mean ratings distributions are tested against the BGG ratings distribution, they are both found to be significantly different.

Same distribution test for original mean rating samples.

	D-score	P-value
BGG vs. Upper	0.2300	P < 0.00001
BGG vs. Lower	0.9380	P < 0.00001

The upper scraped distribution is very similar in shape and overlap to the BGG rating distribution, but the mean appears to be shifted slightly higher. To account for the mean shift, I de-meaned the samples and ran the test again. This time, the upper scraped ratings were not significantly different from the BGG ratings, but the lower scraped ratings were still different.

Same distribution test for de-meaned mean rating samples.		
	D-score	**P-value**
BGG vs. Upper	0.0380	0.86368
BGG vs. Lower	0.2560	P < 0.00001

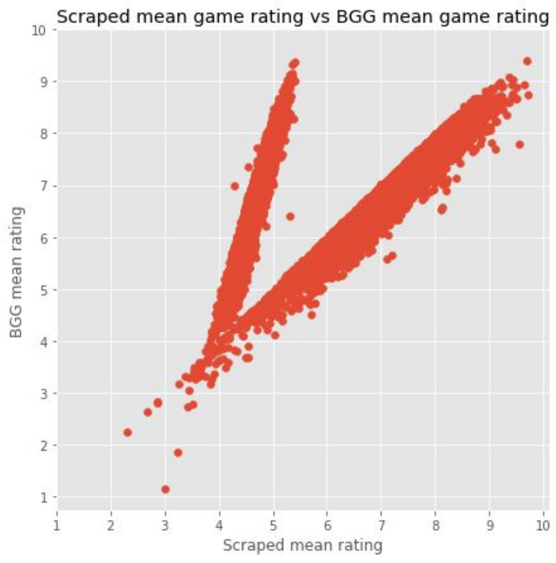
Finally, I standardized the three distributions and tested them again. After standardizing the distributions, all were found to be not significantly different from each other.

Same distribution test for standardized mean rating samples.		
	D-score	**P-value**
BGG vs. Upper	0.0660	0.22635
BGG vs. Lower	0.0360	0.90269
Upper vs. Lower	0.0660	0.22635

My conclusion from the statistical analysis:

The upper scraped ratings have very similar distribution to the BGG ratings, with a slight shift in rating level. This suggests that the upper scraped ratings are valid ratings.

Unlike the upper set of scraped ratings, the lower have quite different variance (and of course different means). However, after all sample sets are standardized, they are not significantly different, meaning that whatever process is affecting the lower scraped distribution of ratings, it is a linear one because it can be corrected by standardization. Next, I'll look at the BGG vs scraped mean ratings in a scatter plot.

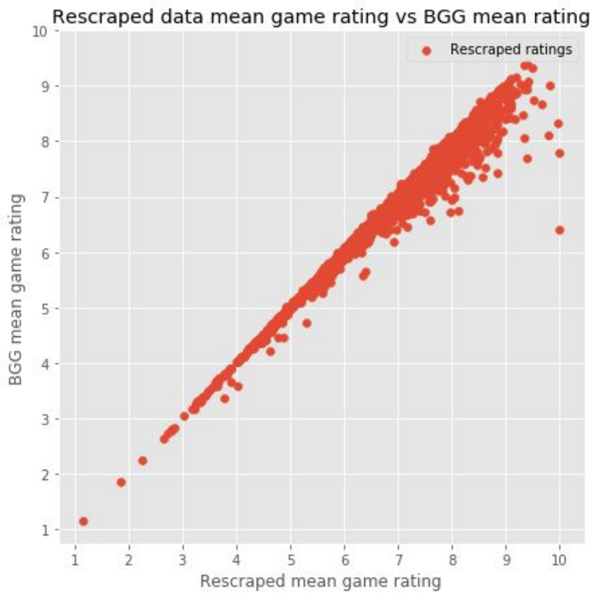


Scraped vs Boardgamegeek game mean ratings:

When game rating means from scraped vs BGG are scatter plotted, I would expect points to lie along the diagonal x=y line. Some games (roughly 75%) do approximately this, while others lie along a distinct line off the diagonal. Those games are clear anomalies.

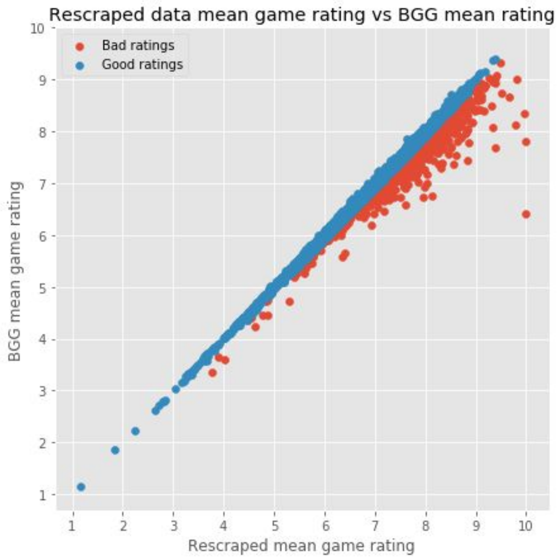
Solution:

One of the problematic aspects of the game rating web scraping procedure is that it takes a long time (about 24hrs) and any pause or crash and restart might have caused inaccurate data to be saved. I did have several crashes/restarts in the first scraping run, so maybe that was the cause. I re-ran a complete rating download with no crashes. I also fixed a bug I discovered in the download script. The result looks a lot better, with the bulk of games showing similar mean ratings as their BGG mean rating.



An additional problem:

The re-scraped data was not perfect, however. For a small number of games, the scraped rating mean still deviated considerably from the BGG rating mean (1118 games differed by >= 0.1). One was off by about 3.25. While most of the games had reasonably small error, I wanted to understand why a few were still off.

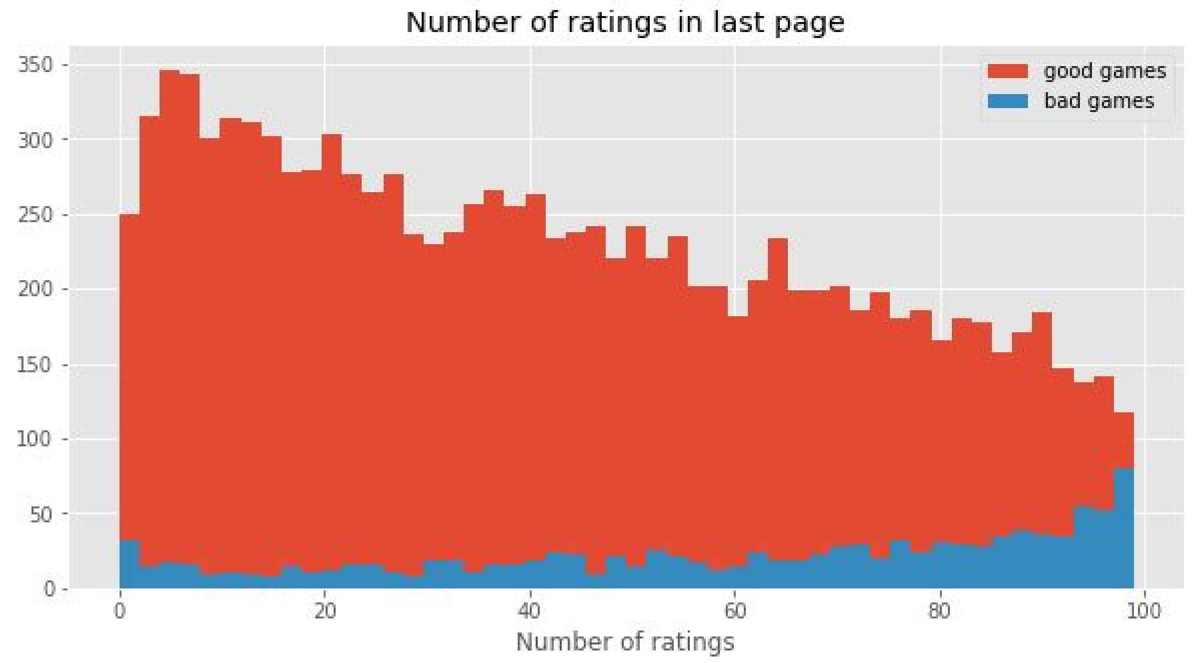


Testing for a possible cause:

Most of the games with high scraped vs BGG mean rating errors had smaller numbers of ratings (my original inclusion threshold for games was >= 100 ratings). It was possible that the ratings scraping code didn't properly download the last block of ratings, which varied in count between 1 and 100. If few or none of those last ratings were acquired properly, then I might see differences in the scraped vs BGG rating deviations based on how many ratings were in the last block.

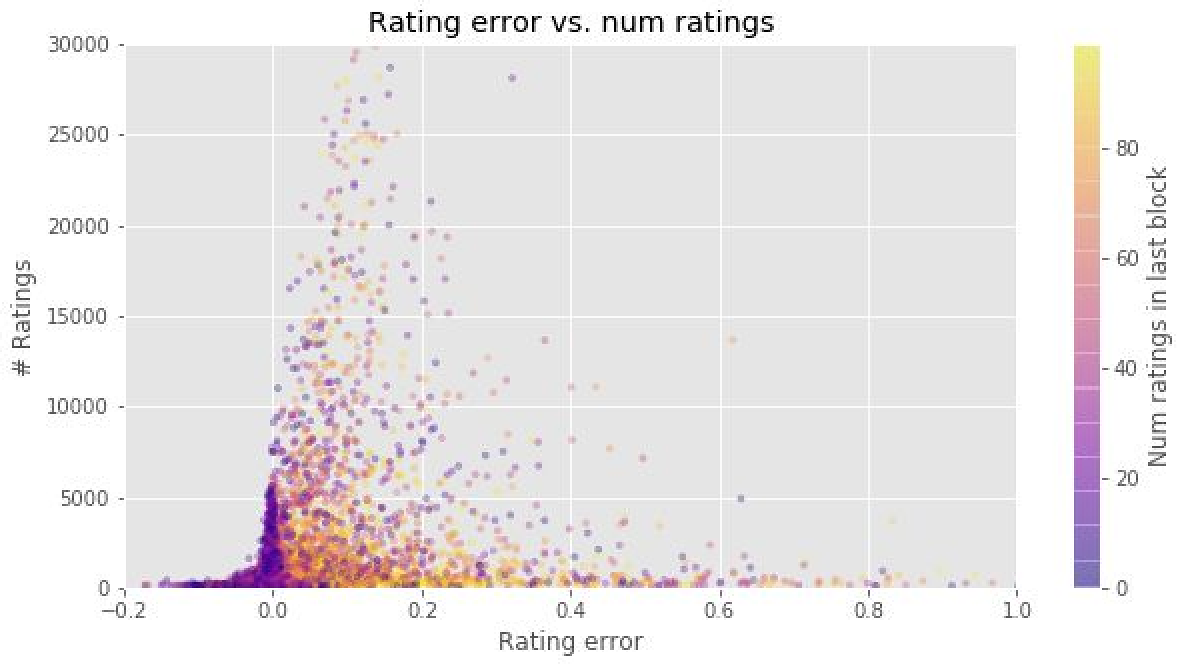
Looking at the number of ratings in the last ratings download page:

It appears that my hypothesis is supported: games with acceptable mean rating error mostly had either many pages of ratings, or lower counts of ratings in their last page. Conversely, the problem games had more ratings in their last page.



Which features are associated with higher rating error?

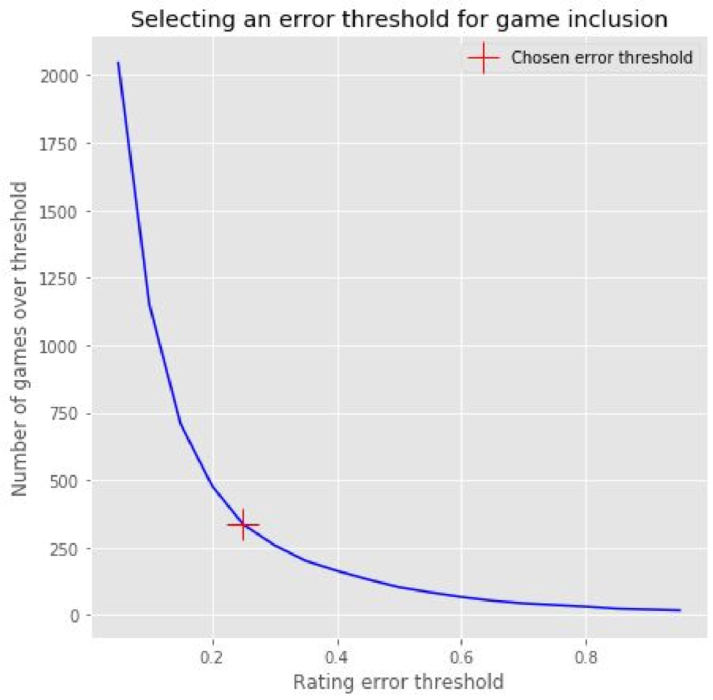
Comparing rating error vs. number of ratings vs. last block number of ratings:
There are two trends here: rating error tends to be higher when there are more ratings in the last block (yellow points), and when games have fewer overall ratings.



Solution: drop games with high mean rating error:

I tried re-scraping the higher error games, and I saw no change in the results. This means there is still apparently a subtle bug in the ratings scraping script, but it seems to affect very few games. For now, rather than spend time tracking down this bug, I will simply drop the small number of games (and their ratings) that have too large of a difference error between their mean BGG rating and mean scraped rating.

The last task was to determine what threshold I should use to exclude higher error games. To do this, I plotted number of games above error threshold vs. error threshold value, and selected the value at the inflection point of the curve, which was 0.25. This threshold resulted in a loss of 336 games from the dataset, with 12264 remaining. The total ratings lost were 544,886, with 15,142,922 remaining.



Recommender models

The goal of my project is to develop a working board game recommender using a dataset of explicit game ratings from the boardgamegeek.com forum site. My idea was to ask a user to name some games that they like, and my recommender algorithm will use those games to predict other games the user might like. The first problem I encountered was that any user asking for a recommendation will be unknown to the system - i.e., they are "new users" and that presents a classic cold start issue: how do you give a recommendation if you don't know much about the user? Well, we do have the user's "liked games" to work with, but any model will need to somehow match that pattern of preferences to the preference patterns in the existing data. I chose two methods, tuned them, and then selected the best model based on performance with my custom evaluation metric.

Model 1: Item Search for Nearest Neighbors (ISNN):

The first model I conceived is a hybrid between a content based and collaborative filtering system. The model estimates recommendations by creating a game (item) coordinate space. Each game is defined by a set of features/coordinates based on the dimensionally reduced ALS filled item x user ratings matrix. The assumption is that games rated similarly by similar users will be closer in the game coordinate space. If a user likes one game, then they might also like nearest neighbor games in the coordinate space and so they can be used as recommendations.

Algorithm method:

- Perform Alternating Least Squares (ALS) matrix factorization on user x game rating matrix, then use latent factors to estimate ratings for all user/game combinations.
- Apply Singular Value Decomposition (SVD) to user dimension, reducing to a small number of features (number of dimensions (n_dims) was a model parameter and varied between 5-1000).
- Given a set of game selections the user liked, for each game:
    - Search for n_neighbors nearest neighbor games in the coordinate space.
    - Add these games to a list of possible recommendations
- Select recommendations based on:
    - Most repeated games
    - Random sample thereafter.

Model 2a: Top ALS Ratings (new user folded in):

The second recommender model I developed is actually what I would call the "baseline" collaborative filtering model: that is, it's based on the most common means of generating recommendations: using ALS based matrix factorization on the target user's ratings to fill in estimated ratings for all items, then selecting the highest rated to use as recommendations. The problem with this method for new users is that you need to "fold-in" the new user: add their ratings to the dataset and re-factorize the matrix. This is very CPU and memory intensive and not a viable production solution unless you can apply some tricks that I can't replicate here. However, it is worth testing the "baseline" procedure against my initial solution.

This version of the "Top ALS rating" recommender takes a user's "liked games", adds the user to the utility matrix (the "liked" games are all rated 10), and then factorizes the ratings matrix using ALS to estimate the user's ratings for all other games. The algorithm then selects the top rated games as recommendations, excluding the input "liked games".

Algorithm method:

- Given a set of game selections the user liked, create a rating vector with 10 ratings for every liked game and unfilled cells for all others.
- Append user’s rating vector to the user x game rating matrix.
- Perform Alternating Least Squares (ALS) matrix factorization on user x game rating matrix.
- Use ALS latent factors to estimate ratings for all games for the user.
- Sort the games by rating and select the top rated games (that aren’t liked games) as recommendations.

Note: For this analysis, I'm cheating a little bit: I ALS factorize all of my test users at the same time for each hyperparameter run, because otherwise the tuning would take forever. This may have an effect on the results, but given that the test users are randomly selected and only 10% of the total data, it seems unlikely to be noticeable.

Model 2b: Top ALS Ratings (proxy users):

In the previous version of the Top ALS Ratings recommender, I “folded in” the user’s liked games and (effectively) re-factorized the ratings matrix to get estimated ratings of all games for the user. This method works, but is too computationally expensive. To fix this, instead of folding in the user, I searched for users in the existing factorized rating data with similar game preferences to use as proxies for the new user. The proxy users' ratings are averaged and the top rated games from the averaged ratings are used as recommendations.

To select the proxy users for a given new user who has provided a list of "liked" games, I select the top rated games for each existing user and score each based on how many of their top games are in the new user's "liked" game list. The existing users with the highest overlap in top games become the new user's proxy users (number of proxy users is a model parameter).

Algorithm method:

- Perform Alternating Least Squares (ALS) matrix factorization on the utility matrix (the user x game rating matrix). Use latent factors to fill in all user/game cells with estimated ratings.
- For each user in the utility matrix, select the top *n_top_rated rated* games.
- For each user in the utility matrix, count the number of top rated games that are the same as the new user's liked games. This count is used as a similarity score between each user in the rating matrix and the new user.
- Select the *n_proxies users* with the highest similarity score and average their ratings. These "proxy" ratings are used as an estimate of the new user's ratings if they had been folded-in to the matrix and the matrix refactored.
- Sort the games by proxy rating and select the top rated games (that aren't liked games) as recommendations.

Note: The utility matrix ALS factorization procedure and top game conversion to boolean array are calculated only once, and can be done off-line. The main on-line computation is the boolean AND against the new user's vector, which is very fast.

Evaluation metric

Models were assessed using a custom evaluation metric that amounts to a simplified Mean Average Precision (MAP) formula. MAP scores recommendations based on their rank compared with an ordered target item set. But, the input to my models, the new user "liked" game list, is not ranked. Therefore, my metric does not account for order and is simply precision:

$$P = \frac{number\ of\ recommendations\ in\ the\ target\ set}{number\ of\ recommendations}$$

, and then precision is averaged over all N test samples.

$$AP = \frac{\sum_{1}^{N} P}{N}$$

Model hyperparameter tuning

Model hyperparameters were tuned using the hyperopt tuning package.

Notebooks:

- Model 1 hyperparameter tuning and cross-validation
- Model 2a (new user folded in) hyperparameter tuning and cross-validation
- Model 2b (proxy users) hyperparameter tuning and cross-validation

Performance testing and model selection

After hyperparameter tuning, models performed in K-folds cross-validation (5 folds) as follows:

Model	**Average Precision**
Model 1: Item Search for Nearest Neighbors	0.001
Model 2a: Top ALS Ratings (new user folded in)	0.159
Model 2b: Top ALS Ratings (proxy users)	0.135

Both versions of model 2 performed much better than model 1. The 2a “fold-in” version had the highest AP score, but this version will not be practical to implement as a real-time recommendation system. The 2b version model that avoids re-factorization performed nearly as well as 2a, and was designed for low computation overhead, so this model was chosen for web app.

The recommender web app

Web app project folders:

- Recommender web app using Model 1
- Recommender web app using Model 2b

The goal of this project was to produce a working prototype web app that demonstrates my chosen recommender model in a real-world scenario. I chose to implement the web app using a Bokeh web server, deployed to Heroku.

The Web app UI consists of a tabbed dashboard with two page tabs, both of which allow the user to input board game titles into a “liked games” list, then click a “recommend” button to receive recommendations provided by the model. The recommendations are shown as a list with a thumbnail image of the recommended game box, a link to get more information from the Boardgamegeek page for that game, and an Amazon buy link that searches for the title in Amazon’s toys and games section.

A note about the use of Bokeh as a web server for the app: the Bokeh server is not an ideal vehicle for this type of application, however it was convenient to implement, and offered a feature that is really helpful: a list selection widget that fills in your board game title selection as you type. That was necessary to make it quick and easy to enter board game titles, while ensuring only titles in the recommender dataset can be entered.

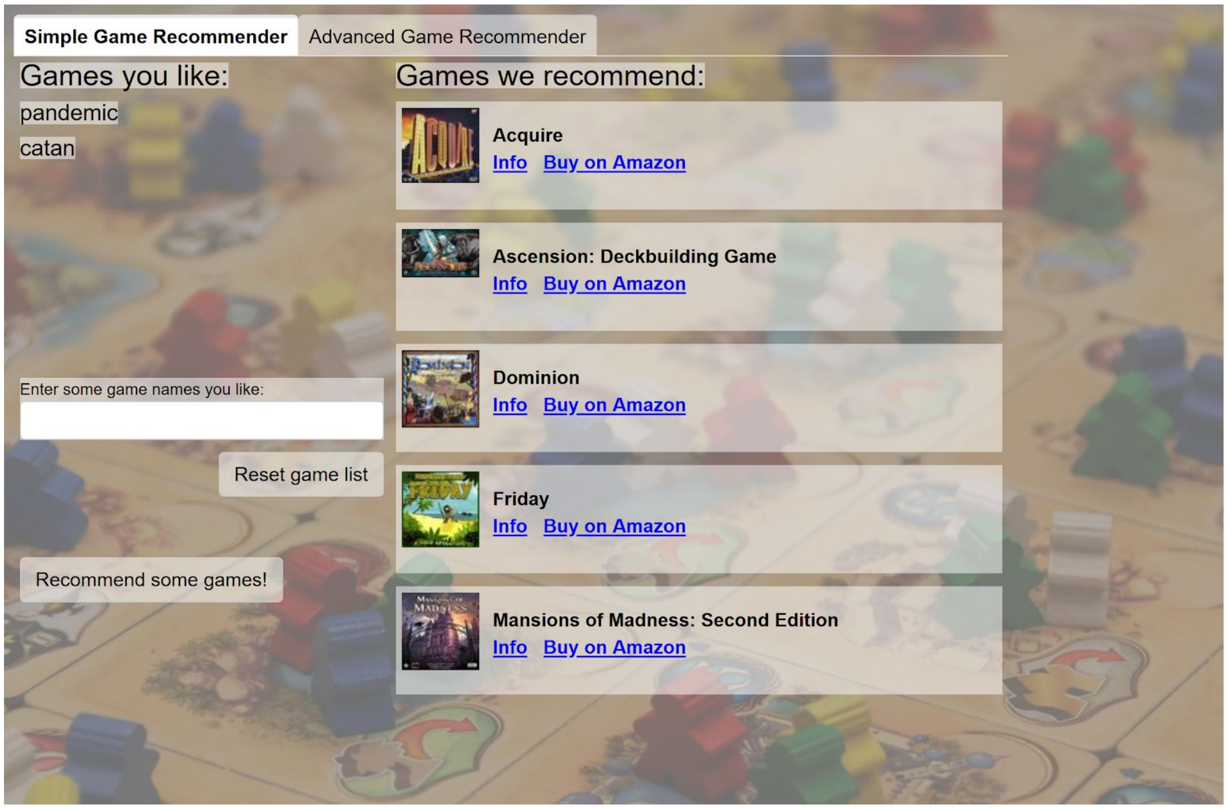
The web app is live right now, and can be accessed by browsing to this address:
jmb-boardgame-recommender.herokuapp.com

About the app: bear in mind that this is a free account, which starts very slowly, so be patient. Also, I could not get the boardgame background image to display in the server (it works OK locally), so there is just a solid color background.

The simple search tab:

The first, and default, tab has minimal complexity: the user can only enter the games they like and click the recommend button. Behind the scenes, the code does some filtering of the recommended games based on metadata provided by the “liked games” list: particularly, the recommendations will reflect the range of game weight (game difficulty) of “liked games”, and mean game ratings must be above a threshold (currently 7.5). The recommender does reasonably well with these restrictions, but it always recommends the same games given the same “liked” list. To provide some variation, I introduced a small random factor into the algorithm so that every time the user clicks “recommend”, a different list of games will be presented.

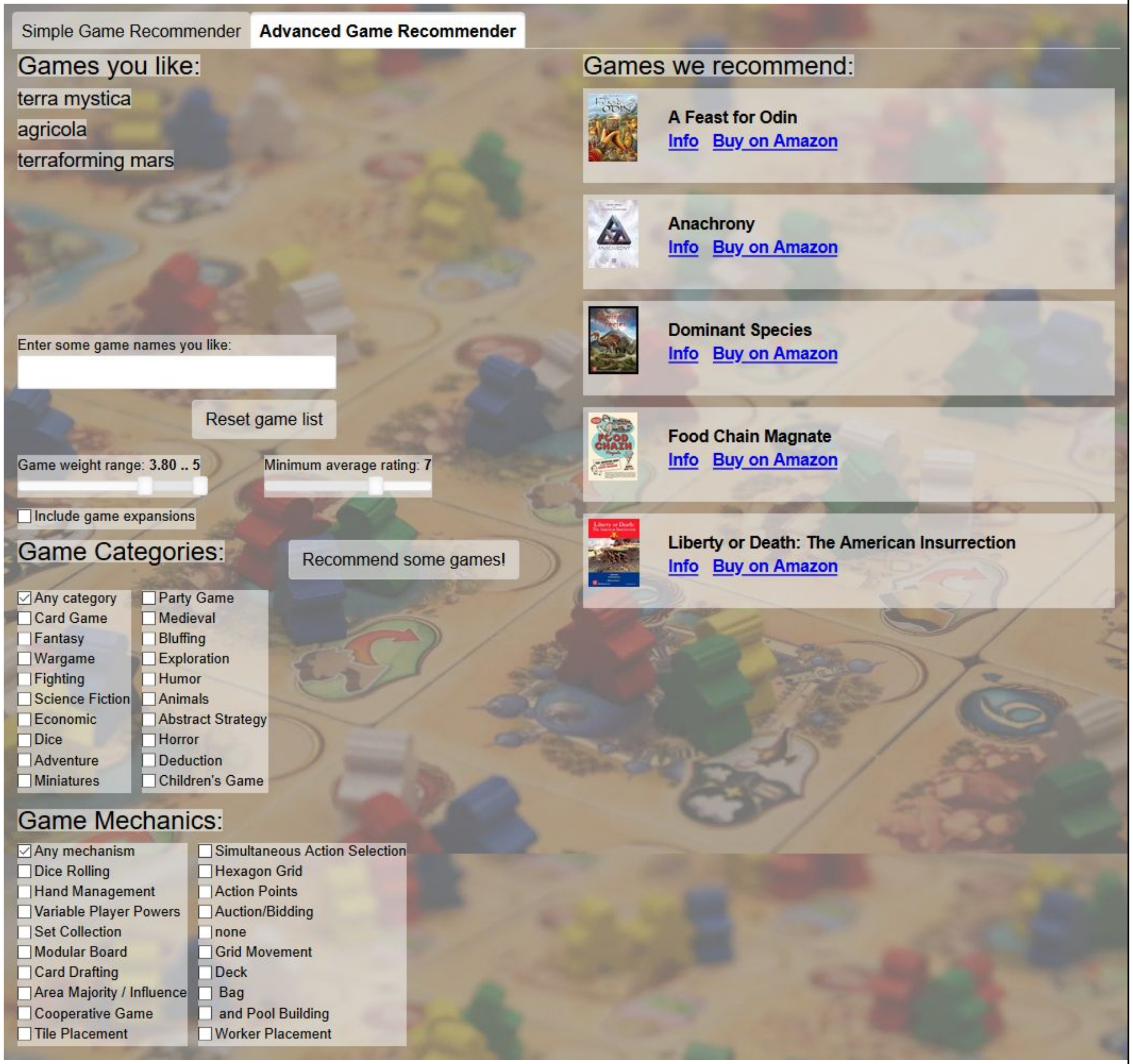
The simple search tab



The advanced search tab:

The second tab includes sliders and check boxes that allow the user much more control over the recommendation search. Sliders allow selection of the range of game weight desired and the minimum rating. You can specify whether you want to include game expansions (excluded in the simple tab). Two check fields allow selection of game type and theme (e.g., “science fiction”, or “economic”), and mechanisms (e.g., “die rolling” or “set collection”). In the recommender code, the user selections are used as filters to exclude all games that don’t match the user selections.

The advanced search tab



Simple Game Recommender | **Advanced Game Recommender**

Games you like:

terra mystica
agricola
terraforming mars

Enter some game names you like:

Reset game list

Game weight range: 3.80 .. 5

Minimum average rating: 7

☐ Include game expansions

Game Categories:

Recommend some games!

☑ Any category
☐ Card Game
☐ Fantasy
☐ Wargame
☐ Fighting
☐ Science Fiction
☐ Economic
☐ Dice
☐ Adventure
☐ Miniatures
☐ Party Game
☐ Medieval
☐ Bluffing
☐ Exploration
☐ Humor
☐ Animals
☐ Abstract Strategy
☐ Horror
☐ Deduction
☐ Children's Game

Game Mechanics:

☑ Any mechanism
☐ Dice Rolling
☐ Hand Management
☐ Variable Player Powers
☐ Set Collection
☐ Modular Board
☐ Card Drafting
☐ Area Majority / Influence
☐ Cooperative Game
☐ Tile Placement
☐ Simultaneous Action Selection
☐ Hexagon Grid
☐ Action Points
☐ Auction/Bidding
☐ none
☐ Grid Movement
☐ Deck
☐ Bag
☐ and Pool Building
☐ Worker Placement

Games we recommend:

A Feast for Odin
Info Buy on Amazon

Anachrony
Info Buy on Amazon

Dominant Species
Info Buy on Amazon

Food Chain Magnate
Info Buy on Amazon

Liberty or Death: The American Insurrection
Info Buy on Amazon

Conclusions and where to go from here

Model 1 was a content based / collaborative filtering hybrid. It performed poorly compared with model 2, which uses a collaborative filtering only method. However, I think that model 1 could be improved by optimizing the nearest neighbor game search algorithm, and by adding game metadata as features in the “game coordinate search”. Numerous attributes exist for each game and can be used as features, such as game weight, number of players, category and game mechanics tags, etc. It seems worthwhile to develop this model further, since it’s fast and it scales well with increasing item count.

Model 2b used a typical collaborative filtering method, except that it substituted for the new user’s ratings “proxy users” from the ALS factorized ratings matrix. This eliminated having to “fold in” and re-factorize with every recommendation request. The method worked nearly as well as the fold-in method used in model 2a, yet with much lower computational overhead. Further refinements of this model allowed it to be deployed on a Heroku server with only 512MB of RAM and a minimal CPU.

All of these models might be improved by doing some data engineering prior to matrix factorization. The games had a very wide spread of ratings counts, from 100 to >80,000. This imbalance could be biasing the factorized rating computation in favor of games with higher ratings counts. If EDA indicates this, I could try balancing rating samples, or weighting during ALS to correct for the imbalance.

Another idea for improving model performance would be to combine recommendations from both models. Since the methods are so different, the combined models might be more accurate at generating recommendations that users will like.

The web app I developed works, but it’s really only a proof of concept demonstration. I intend to develop the app into a full featured website, most likely using flask, and I’ll probably migrate the project to AWS. My goal for these improvements are to 1) provide a useful service at a scale that can serve many users, 2) develop my skills at website design and deployment, and 3) explore using analytics to improve my recommender models (e.g., collecting click data, A/B testing, etc). If all goes well, I may even include ads and derive some revenue, or form an affiliation with boardgamegeek.com.

All notebooks and code:

Project Github: https://github.com/johnmburt/springboard/tree/master/capstone_2

- Utility functions for all notebooks.
- Scraping game titles and IDs.
- Scraping game metadata.
- Scraping user ratings.
- Filtering users and generating a utility matrix.
- Filling in missing ratings using Alternating Least Squares matrix factorization.
- Generating Model 1 web app dataset
- Generating Model 2 web app dataset

- Exploratory Data Analysis

- Model 1 hyperparameter tuning and cross-validation
- Model 2a (new user folded in) hyperparameter tuning and cross-validation
- Model 2b (proxy users) hyperparameter tuning and cross-validation

- Recommender web app using Model 1
- Recommender web app using Model 2b