Recommending Books using Goodreads Data

Adam Coviensky, Stephanie Doctor, Marika Lohmus, Mark Salama

Business Goal

Our company is called Goodreads, a book recommendation website owned by Amazon. The goal of our website is to provide our users a social platform where they can record ratings and reviews of the books they have read and interact with other book-loving users, as well as to recommend them new books that they would ultimately buy from our parent company.

In the previous project, we tested out different personalized recommender engines that would provide our users with the most accurate results. These engines included item-item and user-user neighborhood approaches, a matrix factorization approach, and an SVD approach. While we were able to achieve pretty accurate results, we noted that these approaches lacked in serendipity and novelty - they recommended fairly popular books to everyone.

In order to provide our customers more value, we want to give our users the ability to discover books that they might not have expected. This includes novel books from the genres they have already read, and also introducing books from genres that they may not have explored yet.

Taking inspiration from TasteWeights[1] and Professor Vintch's blog post[2], we have created an interactive recommender system. Rather than supplying the user with a black-box list of recommendations, we believe a recommendation will be better received if the user has an input. Through a simple web app (what-should-i-read-next.herokuapp.com), the user can choose to upweight or downweight twelve genres using sliding scales. This allows a user to receive recommendations based on their current mood/interest.

Furthermore, we believe that recommender systems often suffer from serendipity. We have thus analyzed what genres each user prefers and built a separate recommender system to recommend each user a genre they are less familiar with. We then recommend them popular books from this genre that they have not already read.

---

[1] Bostandjiev, S., O'Donovan, J., & Höllerer, T. "TasteWeights: a visual interactive hybrid recommender system" (2012). *RecSys '12*.
[2] Vintch, B. "A Generative Model for Music Track Playlists" (2017). *Medium.*

The Dataset

We have continued to work with the goodbooks-10k dataset[3]. The dataset contains 5,976,479 ratings (between 1 and 5) for 10,000 most-rated books from Goodreads. The ratings are spread among 53,424 users, giving the data a density of 1.12%.

Users rated between 19-200 books each (median 111). Each book was rated between 8-22806 times (median 248). For more details on this dataset, please refer to our first report.

Recommender Engine Approaches

Book Recommendation

Baseline Model

As a baseline for accuracy of book recommendations on our full dataset, we chose the best model from part 1 of this project, which was SVD implemented using the package SurPRISE[4]. We ran the model with 30 factors and 50 epochs.

Factorization Machine

Factorization machines are typically an excellent way to perform regression and classification tasks under extreme sparsity. Firstly, they use a flattened matrix representation. This allows us to take outside information on each of the books in this dataset to see if using more information can help improve our rating accuracy.

The factorization machine learns parameters for each feature. We have used a second-order factorization machine which learns a global bias w_0, a coefficient for each feature, w_i, and finally, a latent vector for each feature, v_i.

A new predicted rating value is made using the following formula:

$$\hat{y} = w_0 + \sum_{i=1}^{n} w_i x_i + \sum_{i=1}^{n} \sum_{j=i+1}^{n} \left(\overline{v_i} \cdot \overline{v_j}\right) x_i x_j$$

The advantages of second-order factorization machines are that they allow us to model feature interactions efficiently in sparse settings by using the dot product of the latent vectors. A new rating is made by taking a global bias, added to the dot product of the feature coefficients with a

[3] Zając, Zygmunt. “goodbooks-10k”. Obtained from https://github.com/zygmuntz/goodbooks-10k
[4] Hug, N. “SurPRISE”. https://pypi.python.org/pypi/scikit-surprise

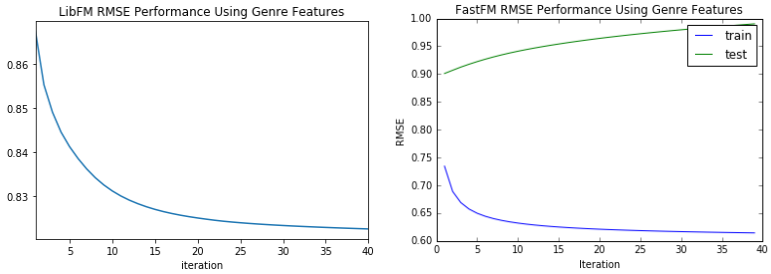
feature vector, added to the pairwise interactions (the dot product of each pair of latent vectors and the interaction of the two corresponding feature values). Since factorization machines take a flattened array as their input, the model creates a coefficient and latent vector for each user, each book, and each additional feature.

Model Package Selection

We explored both the fastFM and original libFM packages for training our factorization machines. LibFM was published by Steffen Rendle, the original author introducing factorization machines. FastFM was later published by Immanuel Bayer specifically for use in python, and it claims to be faster than the LibFM implementation[5] by leveraging the scikit-learn API.

Both packages provide coordinate descent using alternating least squares (ALS), stochastic gradient descent (SGD) and Markov Chain Monte Carlo (MCMC) optimizers. Though MCMC tends to perform the best on both packages, the results would not allow us to easily manipulate the rankings using user input. This is because the MCMC optimizer provides a model in which each iteration is a link in a chain and the results generated by each link are dependent on the links that came before it. Therefore, the resulting model is not just a bias, a feature vector, and an interaction matrix, but rather a more complex combination of various iterations.

We compared implementations of ALS and SGD on both packages, and saw that libFM performed better in terms of testing RMSE. This finding has been also reported by other comparing the two packages[6]. The below graphs show the difference between the LibFM and FastFM performance on the testing set which included the 22 genre features.



The libFM graph above shows clear convergence and an RMSE of approximately 0.82 using the 22 genre features. Since we were obtaining RMSE values of about 1 for sgd in fastFM, we decided to test ALS. Finally, it appears that it diverges in the convergence graph. This is more reason for us to continue our analysis using the libFM package.

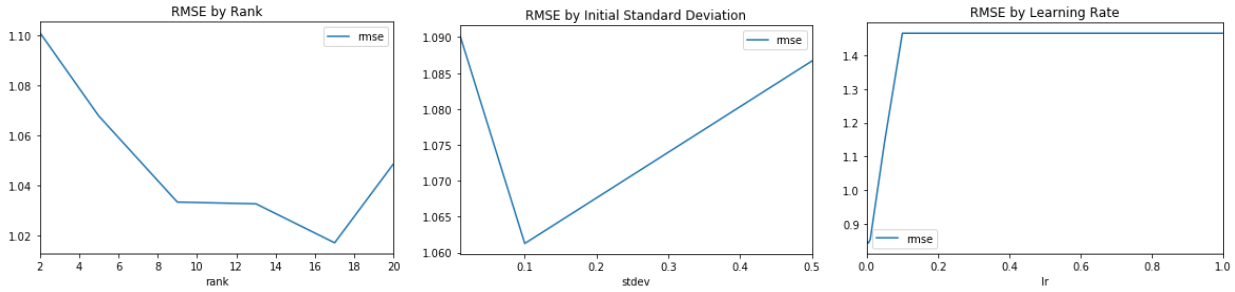
---

[5] Bayer, I. 'fastFM' Accessed from: https://github.com/ibayer/fastFM
[6] Rogozhnikov, A. "Testing Implementations of LibFM" (2016) Obtained from: http://arogozhnikov.github.io/2016/02/15/TestingLibFM.html

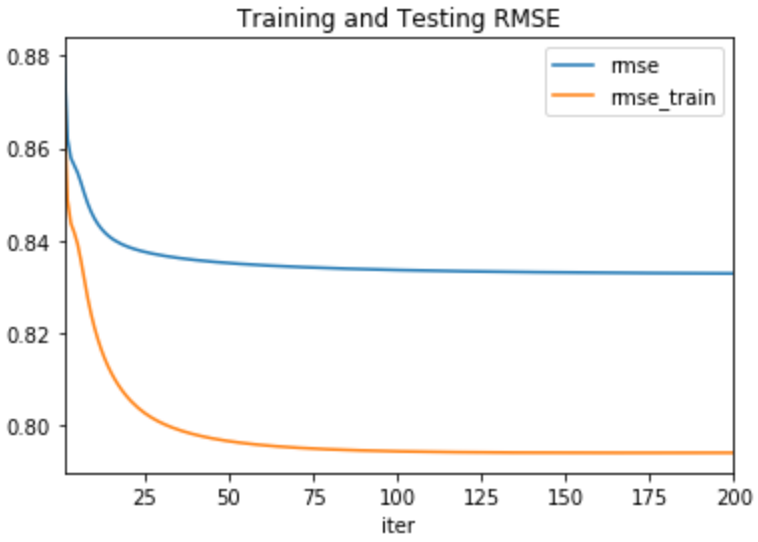
Parameter Tuning

To learn the model parameters, we chose to use libFM's stochastic gradient descent (SGDA), which assigns a learning rate for each regularization parameter and finds the optimal values by maximizing the parameters over a validation set.

We set up a grid search to search over rank (2, 5, 9, 13, 17 and 20), the initial standard deviation (0.01, 0.1, and 1), and the learning rate (0.001, 0.005, 0.01, 0.05 and 0.1). The rank is the length of the interaction latent vectors v_i. The below figures show the results for tuning each of the three parameters:



The best RMSE results were obtained by running the model with a rank of 17, an initial standard deviation of 0.1 and a learning rate of 0.005. Running SGDA using these hyperparameters gave us a model with an RMSE of 0.832892.



The above figure shows how the testing RMSE (blue) and training RMSE (orange) perform over 200 iterations. There is a steep decline within the first 25 iterations, and the testing RMSE eventually converges. As the model keeps running, the training RMSE will continue to decline

and eventually the testing RMSE will diverge as the model starts to overfit on the training data. We chose to use the model generated at around the 100th iteration.

Feature Engineering

Genres

The Goodreads API does not currently have a method of getting the genres for a given book. However, the github repository from which we got the data provides the tag counts for each book. A user can label a book with as many tags as they want, and the tags are not predefined, leading to very varied tags, both in the labels themselves and in the number of tags per book.

First, we had to determine which tags corresponded to genres. We found a list of 40 genres on goodreads which we used as our reference list, and, using the Python package fuzzywuzzy and a minimum ratio score of 0.96, performed a fuzzy matching to account for spelling mistakes or different capitalization. We then summed the tag counts from all of the corresponding tags for each book to get the final tag counts for each genre for each book.

To simplify the model and user experience, we bucketed the genres, narrowing from 40 to 22, 12 of which we let the user adjust in our app. We took the L1 norm of the genres for each book to give us a percentage of how many times users assigned the book that genre, eliminated book-tag counts less than 10%, and then normalized again to get our final model features.

Summary Text Analysis

Summary Collection

Each book was submitted to the Goodreads API to collect a summary. If no summary was available, it was next submitted to the Python package isbnlib. All summaries were cleaned using the Python package beautifulsoup. Of 10000 books, 9106 had a valid summary from one of these libraries.

Tf-idf

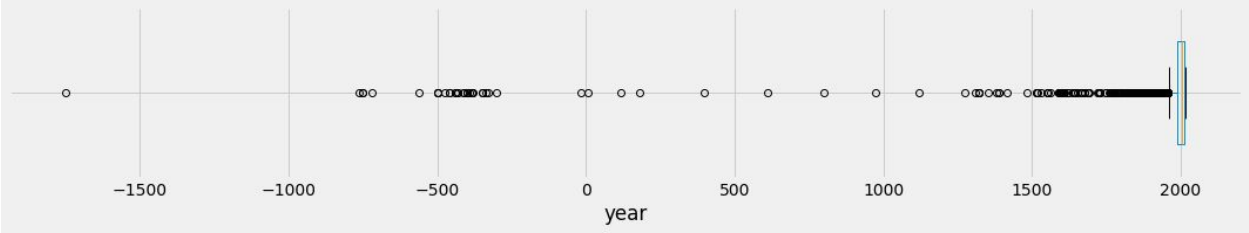
After removing stop words, tf-idf features were computed from unigrams, bigrams, and trigrams with a minimum document frequency of 2 using scikit-learn. These features were then scaled and PCA was performed to extract the first 5 principal components. Books with missing summaries were assigned 0.1 for each of the 5 resulting features. Finally, the values were normalized.

Word Embeddings

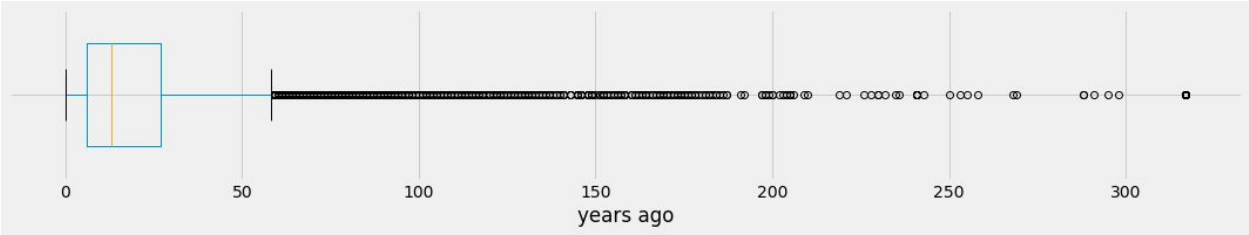
For each valid summary, a document vector was also computed using the 300-dimension pre-trained Google News word embeddings and the Python package gensim. Books with missing summaries were assigned 0.1 for each of the 300 features. Finally, the features were scaled, PCA was performed to extract the first 5 principal components, and the values were normalized.

Other Metadata

In addition to using the user-item-rating dataset, genre information, and NLP features, we also included additional contextual information about the books within our dataset. These columns include the year that the book was published. We can see a boxplot of the distribution of years below:



This shows us there is a book from -1750. Upon further investigation, we have found that this corresponds to The Epic of Gilgamesh, which (according to Wikipedia) was written circa 2100 B.C. We decided to group every book published before 1700 as being published in the year 1700; this accounts for approximately 100 books. We imputed any missing publication year values with the median year, which was 2004. Finally, we calculated how many years ago each book was published (2017 - x). After these computations, the distribution looked as follows:



Finally, the number of years ago was normalized to a feature between 0 and 1 by dividing by 318 (317 years between 1700 and 2017 + 1 so no value is 0). A value closer to 1 indicates that the book is more recent, while a value closer to 0 indicates that it is an older book.

In addition, to publication year, we included the authors of the book as categorical variables. There were 4664 unique authors in the dataset.

Lastly, we included the language the book was written in as a feature. There were multiple English categories, which we bucketed together, as well as grouping missing values as “unknown”, leading to 22 unique language labels (input into the model as separate features).

User Interaction

The web app allows the user to interact with the model. Once we have learned each of the model parameters, we adjust the prediction function using a weight vector z:

$$\hat{y} = w_0 + \sum_{i=1}^{n} z_i w_i x_i + \sum_{i=1}^{n} \sum_{j=i+1}^{n} \left(z_i \overline{v}_i \cdot z_j \overline{v}_j \right) x_i x_j$$

The weight vector is derived from user input. The user sees 12 sliders corresponding to 12 of the 22 genre buckets, which range from -2 to 2 and map to values between 10^{-2} and 10^{2}. These values, which default to $10^{0} = 1$ for both genres that have not been adjusted and genres that do not have a corresponding slider, make up the weight vector *z*.

Once the user chooses the slider values and presses the submit button, predictions are made for each of the 10,000 books. First, *z* is multiplied by *w* element-wise and then by each corresponding latent vector in *V*. We then make a sparse upper triangular matrix *X* consisting of the outer products of $x_i x_j$. This allows us to take the value of each entry in *X*, $x_{i,j}$ and multiply it by the dot product of v_i and v_j quickly.

From the estimated predictions, the top 5 books the user has not already read are returned to the web app. This process, from hitting Submit to receiving new recommendations given new feature weights, takes approximately 13-14 seconds.

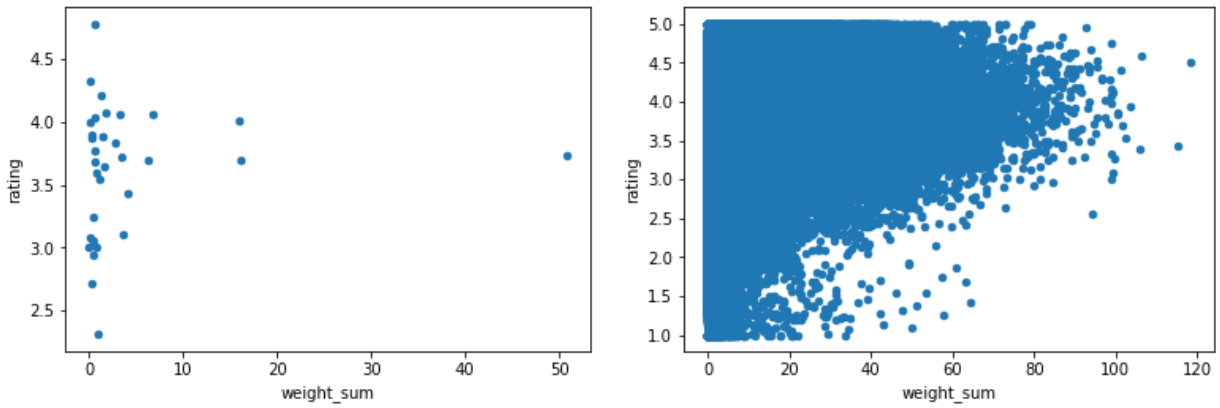
Genre Recommendation

With the aim of providing diverse recommendations, we have also explored users' relationships to book genres. We wanted to see if we could find genres that users do not typically read, but that they might enjoy, such that we could recommend books from these genres. We used the aforementioned genre tags (not bucketed) to get 38 possible categories, and normalized the weights for each book. For example, *To Kill a Mockingbird* was tagged as 77% "Classics" and 12% "Historical Fiction", while *The Hobbit* was tagged as 19% "Classics", 60% "Fantasy", and 16% "Fiction".

By merging these genre weights with the ratings, we obtained genre ratings for each user, as in:

User ID	Genre	Avg Rating	Weight Sum	Book Count	Avg Weight
4	Art	3.87	0.02	2	0.01
4	Biography	4.33	0.03	2	0.02
4	Chick Lit	3.20	0.04	4	0.01
4	Children's	4.79	1.11	17	0.07
4	Classics	4.08	11.99	43	0.28
4	Contemporary	3.58	0.62	25	0.02
4	Crime	2.99	0.13	3	0.04
4	Fantasy	4.35	7.67	31	0.25
4	Fiction	3.88	13.37	47	0.28

Weight sum is the sum of all of the weights for that user-genre. Ratings for genres with low weight sums are based on small samples and might not be representative of how users feel about the genre. We can see below that as users rate more books in a genre, the genre rating has lower variability, at the individual user level (left), and in the aggregate (right).



For this reason, we have limited the training set to genre ratings for which users have a weight sum above 2. To make genre predictions, we used SVD implemented in the Python package SurPRISE and grid searched to tune hyperparameters (number of factors, number of iterations, regularization parameters, and learning rate).

Using the predicted ratings for each genre, we recommended to each user the two highest-ranking genres that they do not usually read.

To recommend specific books from those genres, we tried several approaches. Our first approach was to find the user's mean genre weighting (i.e. the mean genre weights of the books they have read) and compare it using cosine similarity to the ten most popular books in each recommended genre. We hoped this would return books outside of their preferred genres but still with elements consistent with the books they read. However, the approach returned the same books for all users. Our final approach was to recommend the two highest-rated books in the recommended genre they they had not read and that had at least 100 ratings.

Results

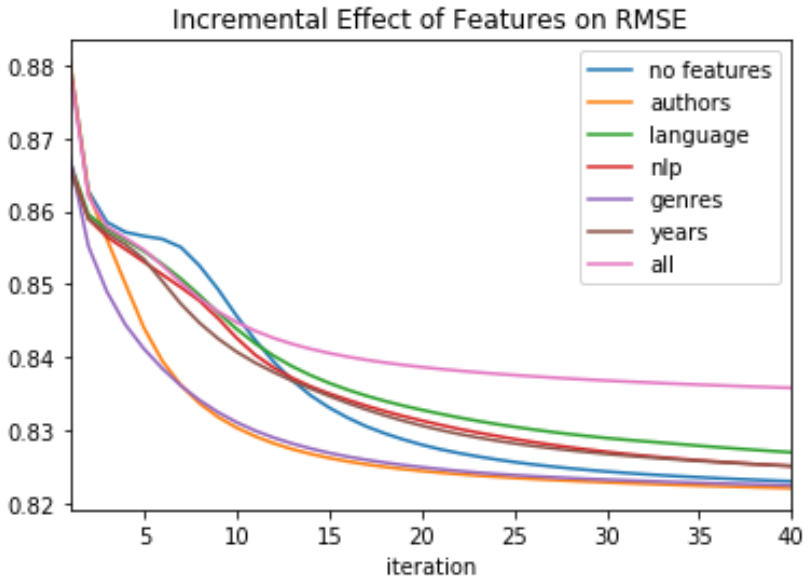
Book Recommendation

Baseline Model

Our baseline SVD model achieved a test RMSE of 0.7442.

Factorization Machine

We ran the model for 40 iterations using no additional features, and then only including one set from the groupings above at a time. Each model was trained using the same hyper parameters of 17 ranks, a initial standard deviation of 0.1 and a learning rate of 0.005.



It is interesting to note that there is little to no correlation between the number of features and how quickly the model converges. For example, including authors adds over 4000 additional features, but the model converges faster than the same model with no additional features. The 22 genres also seem to be a good feature set to include.

Language, the NLP features, and the normalized publication year, however, seem to increase the time that the model converges - this may be due to the fact that the model needs longer to learn how to incorporate those features or takes some time to learn to ignore them.

Including all features together (indicated by the pink line) had the worst performance out of all combinations within the first 40 iterations. This is due to the fact that with each additional feature, the model needs to learn over 60K additional interactions with existing features. Adding all 4,729 creates over 307 million new interactions which decreases the rate of convergence. While the model with no additional features converged in about 45 iterations, the model with all features took over 200 iterations to converge (see the testing vs training RMSE graph above). This means that the model with all features ultimately performs as well as the model with only author features, but it takes it much longer to get to the same accuracy result.

Ultimately, given these considerations, we decided to move forward using just the genres as additional features for the model in our web application This model had an RMSE of about 0.833.

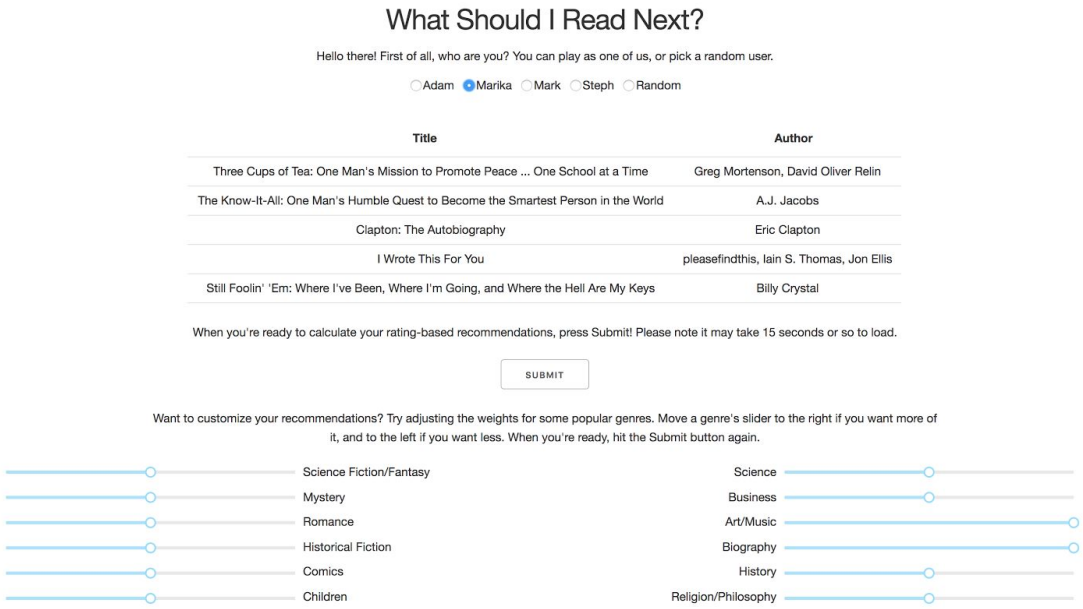
User Input

The last part of our evaluation came from user-input. Since this system is offline, it is virtually impossible to test how well our recommender system was performing with user input. Firstly, we don't have the actual users around to input the information on their genre preferences. Secondly, even if we knew what genres they wanted to receive recommendations from, if they were new genres, we could not test the accuracy.

To solve this problem, we have added ourselves to the ratings matrix. We each rated a different number of books ranging from 17 to over 140. We trained our model with our own unique user ids and ratings appended to the bottom of the ratings matrix. This allowed us to test the app ourselves and decide whether or not we felt we would actually read any of the recommendations from our web app given our input.

In the end, we found that we were all recommended similar books. Before adjusting any sliders, each of us were recommended at least one Calvin and Hobbes book as well as the ESV Study Bible. These were not the greatest recommendations and we would have preferred some variety. By moving the sliders towards more obscure genres, we receive similar recommendations from that genre. However, when we adjust the weight of science fiction, we actually get somewhat different recommendations for each of us. We believe this problem does stem partly from our similar tastes in books. When tested on random users, we often see that they are recommended vastly different books. However, it is still shocking and disappointing to see us all being recommended the ESV Study Bible.

In the screenshot below, we can see a circumstance in which the app does perform well. By moving the sliders up all the way for Art/Music and Biography, Marika gets recommended Eric Clapton's autobiography and Billy Crystal's autobiography. It similarly works for Mark, giving him the Miles Davis and Dr. Seuss autobiographies.

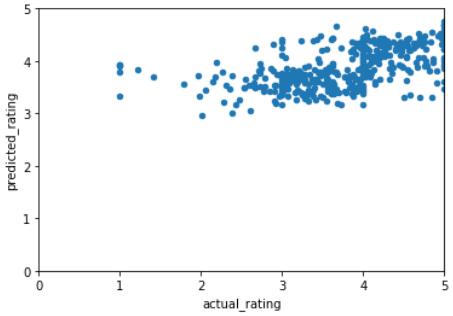


Overall, we are satisfied with the web app and interactivity. We believe that there is still some work to be done on tuning the actual amount by which we want to weigh the genre parameters, as there is a fine line between weighting so heavily as to always get the same recommendations and not enough so that the inputs are not taken into account.

Genre Recommendation

When we tested the genre recommendation algorithm on high-weight sum data, we had very low errors (RMSE 0.25). When we tested on low-weight sum data, we had much higher errors (RMSE 0.65). This result is consistent with the high variability of ratings for low weight sum data and the fact that model was trained without this low weight sum data.

The figure below compares predicted ratings to actual ratings for low weight sum genres, for a subset of users. The actual ratings are much more varied than predicted ratings.



Anecdotally, the model seems to predict genres that we are interested in. For Mark, the model only had rating data for Fantasy, Classics, Fiction and Nonfiction because all other genres had a weight sum less than 2. With this limited data, the genres it predicts that Mark would rate the highest are Children's, Historical Fiction, History and Religion and those he would rate the lowest are Chick Lit, Business, Self Help and Romance. While these predictions are not perfect (e.g., Mark enjoys Business and Self Help books), there is some value to these rankings. Perhaps more importantly, by ultimately recommending the most popular books in each genre, the user is served decent recommendations.

Conclusions, Caveats, and Future Work

We were surprised and a little disappointed to see that adding more information to our model reduced performance, reinforcing a point from Lees-Miller et al. (2008)[7].

Adding features to our factorization machine lowered the accuracy, as we found when a model using all features had the worst test RMSE (even after it converged to about 0.832 after 200 iterations). We have also witnessed the filter bubble in full effect. Viewing the output recommendations in our web app, we see that all 4 of us are recommended books from the Calvin and Hobbes series. Despite this, we still believe that more data can possibly improve the recommendations given ideal feature engineering. Going forward we would like to run the model with NLP features longer as it did not seem to converge. We also would like to add implicit features, such as the following features x_j:

$$x_j = \frac{1}{\sqrt{|N_u|}},\ if\ j\ \varepsilon\ N_u$$

That is, for each user, append the feature vector by 10k features, one for each book. If the user has rated the book, input a value of $1/\sqrt{|N_u|}$, where $|N_u|$ is the total number of books that user has rated. These are equivalent to running SVD++.

Adding features also lead to increased time to train the model. Time also posed a problem when making predictions. Although 15 seconds is pretty quick to be able to get predictions for 10,000 items and output a top k list, ideally a user would not have to wait this long each time they slide the bars. The best method of speeding up predictions we have come up with, which we would like to implement in the future, is to pre-compute each rating given the dot product and all of interaction terms that do not include the genres. Then, as a user updates the sliders, we just compute the additive value from all the genre related features. We could pre-calculate the rating $y_{initial}$ where g is our set of genre related features:

$$\hat{y}_{initial} = w_0 + \sum_{i \in g} w_i x_i + \sum_{i \in g} \sum_{j \in g} \left(\overline{v}_i \cdot \overline{v}_j \right) x_i x_j$$

Then our final rating is:

[7] Lees-Miller, J., Anderson, F., Hoehn, B., & Greiner, R. "Does Wikipedia Information Help Netflix Predictions?" (2008) *ICMLA '08.*

$$\hat{y} = \hat{y}_{initial} + \sum_{i \in g} z_i w_i x_i + \sum_{i \in g} \sum_{j \in g} \left(z_i \overline{v}_i \cdot z_j \overline{v}_j \right) x_i x_j$$

On a brighter note, we have seen that it is indeed possible to create a recommender system with user input. These models do not always have to be a black box. Through adjusting the weights of the learned model parameters, we have been successful in recommending new books to users from the genres they have chosen. Furthermore, we have created a system allowing for more serendipitous recommendations by deliberately recommending users books from two genres they do not typically read.