Comparison of Four Recommender Systems on Goodreads Dataset

Jay Shi, js133
William Su, ts47

for COMP 340 Statistical Models for Data Science

by Professor Devika Subramanian

1. Introduction

Goodreads is an online platform where its users can "shelf" books that they are currently reading, markdown books that they had read, and rate these books from a scale of 1 to 5. In our term project, we explored and built models on Jupyter Notebook running on AWS EC2 instances that will predict a user's possible rating of a book that they have not read before. The idea is that using these models, we can then recommend books we predict specific users will most highly rate based on the different interactions they've had with the books. The four recommender systems we have built are a user-user similarity model, a weighted user-user similarity model, a matrix factorization model, and a neural network based matrix factorization model. We will be comparing and analyzing the performances of the models.

2. Data

Data Cleaning

We obtained a *Goodreads* dataset (link in footnote) of 726932 users and 258,585 books from University of California San Diego. Each row is a record of an interaction between a user and a book. The average number of interactions per user is 76.36, which means that the dataset has 55508527 rows in total. The dataset has 7 columns: *user_id, book_id, review_id, isRead, rating, date_added,* and *date_updated.* The dataset is encoded in JSON. We used a Python script found on GitHub (link in footnote) to convert the JSON file to a CSV file. From there, we loaded the CSV file using Pandas.

We only kept four columns of the dataset: *user_id, book_id, isRead,* and *rating*. *isRead* is

Data source: https://sites.google.com/eng.ucsd.edu/ucsdbookgraph/home
JSON to CSV converter: https://github.com/Yelp/dataset-examples/blob/master/json_to_csv_converter.py

a boolean variable that indicates if a user has finished reading a book. If *isRead* is 0, it may be the case that the user has never read a book and it may also be the case that the user is currently reading the book but has not finished yet. Actual ratings range from 1 to 5. A rating of 0 indicates that the user has not rated the book. We validated that the unique values of the rating column are indeed 0 to 5.

One of our early troubles is that values of *user_id* are strings of numbers and characters instead of consecutive numbers. Even though values of *book_id* are all numbers, they are not consecutive. Our solution was to map every unique *user_id* to an integer starting from 0 and apply the mapping to the *user_id* column. We did the same for *book_id* column.

There was another serious problem that we did not discover until late: there are duplicate user-book pairs in the dataset. After running half of our models we noticed that there were entries larger than 5 in the rating matrix. The reason is that when there are duplicated row-column index pairs in the table, Scipy adds the values of entries up when converting to COO sparse matrix. After discovering this problem, we decided to drop all the users that rated a book multiple time.

Extracting Matrices

Now the data is clean, we can then extract a 679113 by 14473 rating matrix based on th e users' rating column and another 679113 by 14473 isRead matrix based on the isRead column. To create a shelved matrix, we simply added another column in the raw data frame where every value is a 1; we then used this to create a 679113 by 14473 shelved matrix, where if a user i has s helved book j, the entry at *i,j* will be 1 and 0 otherwise.

Unfortunately, loading the rating matrix to dense form exhausts all of our memory and we can barely do anything else. Hence, we decided to shrink the dataset. We shrunk the number of users from 679113 to 20000 randomly. By the same proportion, we shrunk the number of books to 426. The eventual shelved matrix, isRead matrix, and rating matrix we worked with in this project are thus 20000 by 426.

Train-Test Split

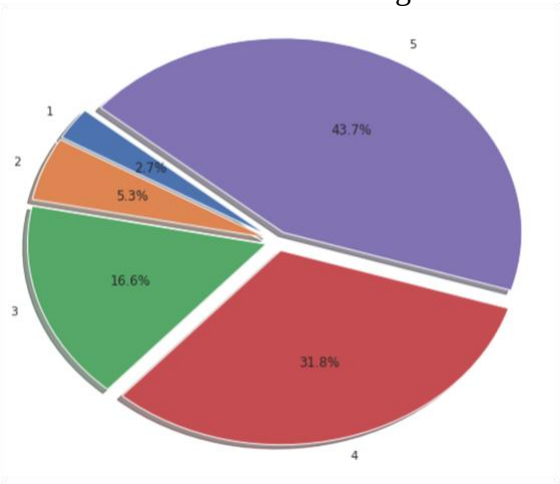
We employed a different method for splitting training data and testing data from what we did in lab. Instead of randomly removing all the ratings of 10% of the books, we filtered out the indexes of nonzero entries in the rating matrix. We then randomly selected 10% of those entries as our testing data. To obtain our training data, we zeroed out the same 10% entries in the original rating matrix.

The reasoning behind this method is that we want all of our models to have the same set of training data and testing data. If we zero out 10% of the columns, the training set is basically implying that no user has ever rated the books corresponding to the columns. Matrix factorization model can still learn the training data just fine, but for user-user similarity model, when predicting for the rating that user i would give to book j, since no other user in the training data has rated book j, we cannot compute the predicted rating. In order that the performances all

our models can be compared against each other using the same training set and testing set, we decided to employ the train-test split method described.

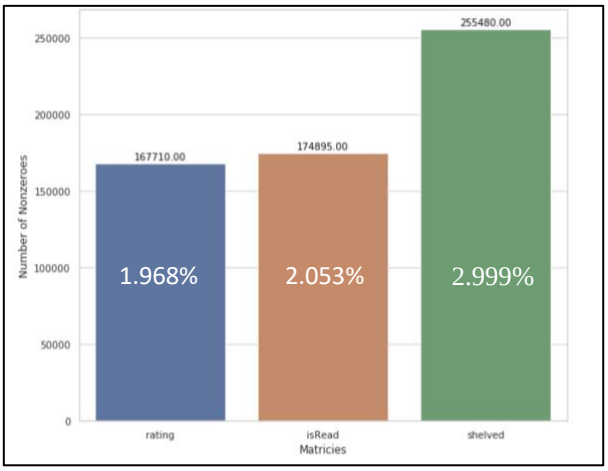
3. Exploratory Data Analysis

Distribution of Ratings



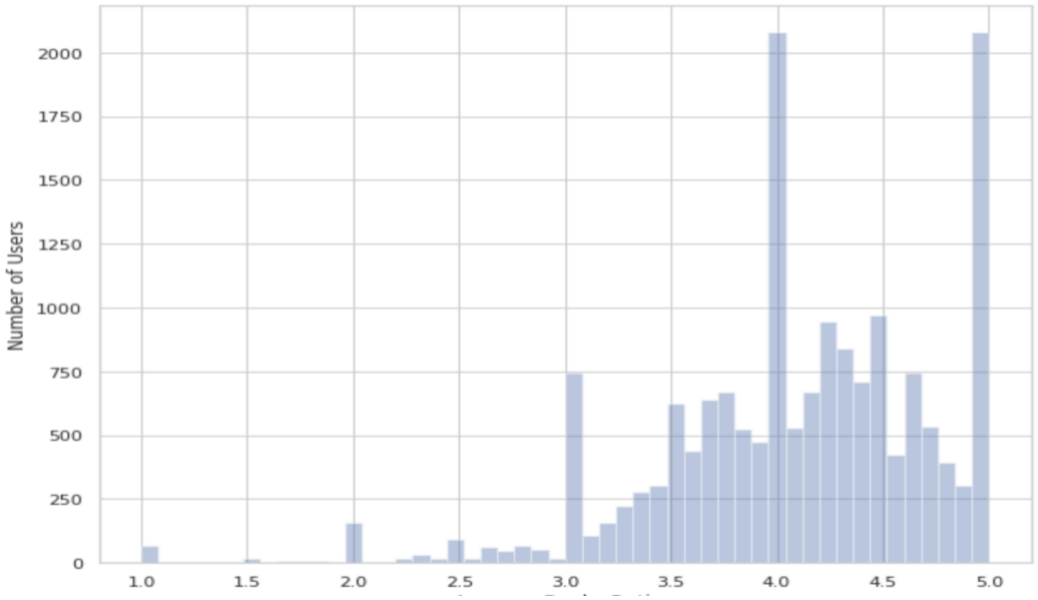
The above pie chart on the right shows the distribution of ratings in the rating matrix. Majority of ratings are 4 and 5 stars ratings and users rarely give 1 or 2 stars ratings.

Number of Nonzero Entries and Sparsity

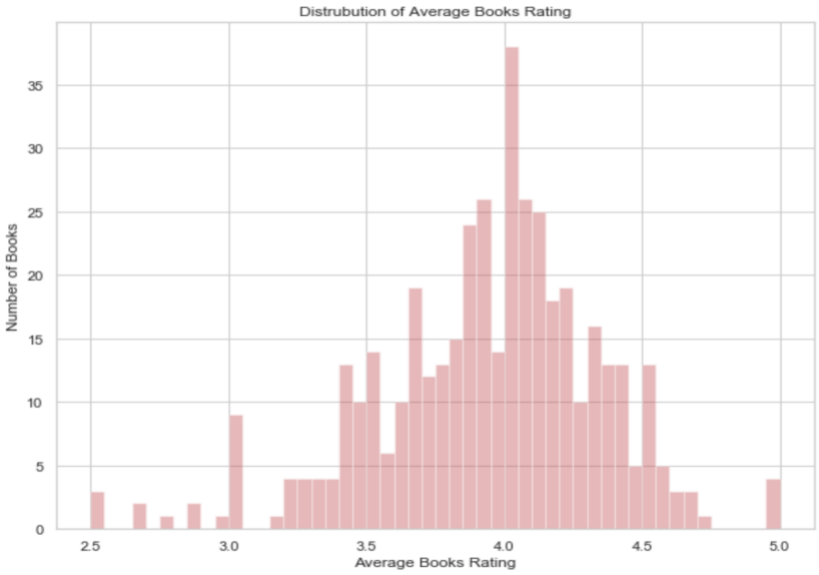


The above bar plot shows the number of nonzero entries in each of the three matrices. We can see that if a user has read a book, he or she is very likely to rate it. The percentage on each bar shows the sparsity of each matrix.

Distribution of Average Ratings Given by Users



The above histogram shows the distribution of average ratings given by users. It appears that the users like books very much – a lot of them give 5 stars reviews to every book. Most of the average ratings are between 3 and 5.



The distribution of average books rating tells a similar story. Most books get average ratings between 3.5 and 4.5. It appears that there are not many books disliked by the users in this dataset. The distribution looks bell shaped, with a mean at around 4.

4. Models

To compare the performance of each models, all the models were trained on the same training set and tested on the same testing set. Mean squared error was the metric for all the models.

Model 1: User-User Similarity Model

User-user similarities are computed by Pearson correlation. To have a benchmark for our models, we first ran an unweighted user-user model to find out the testing MSE of this model. After doing so we found out that the unweighted user-user model has a quite a low testing MSE:

MSE = 0.9756388191656609 (unweighted user-user model)

This is then the MSE that our weighted user-user model attempts to be lower than.

Model 2: Weighted User-User Similarity Model

Inputs:

1. A sparse matrix S where at $S_{i,j} = 0$ if $user_i$ had not shelved $book_j$ and $S_{i,j} = 1$ if $user_i$ had shelved the book.
2. A sparse matrix Q where at $Q_{i,j} = 0$ if $user_i$ had not read $book_j$ and $Q_{i,j} = 1$ if $user_i$ had read the book.
3. A sparse rating matrix R where at $R_{i,j} = 0$ if the $user_i$ had not rated $book_j$ and $R_{i,j} = 1$, 2, 3, 4, or 5 if $user_i$ had rated $book_j$.

Method:

The goal of this weighted user-user recommender is to produce a lower MSE than an unweighted user-user recommender. To clarify between the two, note that the unweighted user-user model is simply the most basic form of a user-user model – a single rating matrix is inputted into a similarity algorithm and a new prediction matrix is produced. However, the weighted user-user recommender is a model that we thought of implementing, inspired by the paper "Item Recommendation on Monotonic Behavior Chains" by MengTing Wan and Julian McAuley (link in footnote).

The idea behind the weighted user-user model is to linearly combine the three matrices that we can obtain from the raw data – a shelved matrix, a isRead matrix, and a rating matrix. We believe that if we only input the rating matrix into the user-user model, we are missing out on information that the raw data is providing.

Consider the following example where there are only 3 users and 1 book:

$user_1$ rated the book with a score of 3.
$user_2$ read the book but didn't rate the book.
$user_3$ shelved the book but didn't finish reading the book.

If we proceed with the unweighted user-user model, both *$user_2$* and *$user_3$* will have the same similarity score when compared to *$user_1$*. However, we believe that because *$user_2$* actually finished reading the book, *$user_2$* has a more meaningful interaction with the book than *$user_3$* and therefore *$user_2$* should be more similar to *$user_1$*. The reasoning behind weighting the three matrices and adding them together thus is to capture the idea that even if users didn't rate a certain book, they may have a more meaningful interaction with the book than other users and this should be accounted as an input into the user-similarity algorithm.

Implementation 1:

In our first implementation, we linearly combined the three matrices with the equation below:

$$M = (w_1 * S) + (w_2 * Q) + R$$

where $w_1 < w_2 < 5$ and M is the final matrix inputted into the unweighted user- user model. w_1 should be less than w_2 because a read book should be more meaningful than a shelved book. However, this model turns out to be **very inaccurate** in predicting the rating scores. We believe it's because in this equation we are actually double counting some of the data. For instance, check out the following matrices:

$$M = 0.3\begin{bmatrix}0 & 1\\ 1 & 1\end{bmatrix} + 0.5\begin{bmatrix}0 & 0\\ 1 & 1\end{bmatrix} + \begin{bmatrix}0 & 0\\ 0 & 4\end{bmatrix} = \begin{bmatrix}0 & 0.3\\ 0.8 & 4.8\end{bmatrix}$$

Here, $M_{1,2} = 0.3$ and $M_{2,1} = 0.8$ both capture what we want. $M_{1,2}$ shows that users who have shelved the book should have some weight in the final matrix but this weight should be the smallest. In $M_{2,1}$, we show that users who read the book should have a slightly higher weight than that of a user who only shelved the book. The problem occurs at $M_{2,2} = 4.8$. Here, because of the nature of our equation we added 0.8 to the original rating of 4 too. This is not the aspect that we are trying to capture. We believe this is the problem that caused our first implementation to have MSE's that are 1.5+, which is way more than the 0.9756 benchmark we are trying to beat.

Implementation 2:

To solve the issue that we add in implementation 1, we came up with a new way to linearly combine these three matrices which is simply:

For all entries (i, j) where $R_{i,j} = 0$:

$$M_{i,j} = (w_1 * S_{i,j}) + (w_2 * Q_{i,j})$$

else:

$$M_{i,j} = R_{i,j}$$

This way the example matrices that we had in implementation 1 will become:

$$0.3\begin{bmatrix}0 & 1\\ 1 & x\end{bmatrix} + 0.5\begin{bmatrix}0 & 0\\ 1 & x\end{bmatrix} + \begin{bmatrix}0 & 0\\ 0 & x\end{bmatrix} = \begin{bmatrix}0 & 0.3\\ 0.8 & x\end{bmatrix}$$

where $x = 4$, so

$$M = \begin{bmatrix}0 & 0.3\\ 0.8 & 4\end{bmatrix}$$

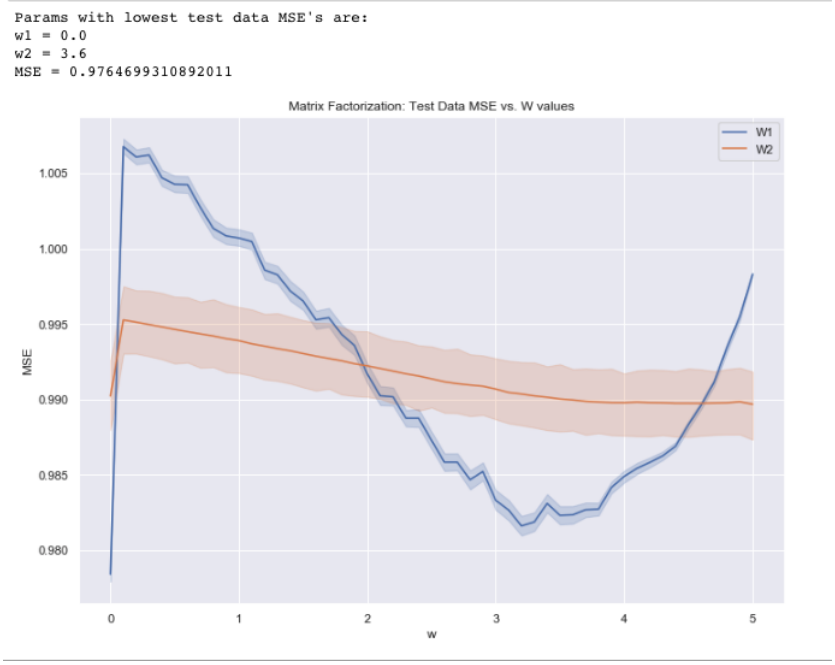
We then input this matrix M into the unweighted user-user model and use the Pearson similarity coefficient to produce our prediction matrix.

To implement this model, we have two unknown parameters to sweep: w_1 and w_2.

Our initial prediction is that w_1 and w_2 should be both smaller than 1 as they shouldn't be given weights more than those that have actually rated the books. However, the initial models with these parameters didn't give us quite the best models. We then realized that the two weights can probably be slightly higher, especially w_2. We had a notion that if the book is read by the user but not rated, perhaps the user is quite neutral about the book and so we tried out $w_2 = 3$ and we saw a huge improvement in performance.

To accommodate for the huge memory that these matrices take up, we opened up two Amazon Web Service EC2 instances with 32GB of RAM that can run these models without the Jupyter Notebook kernels shutting down due to MemoryError. Since we now have the computing power, we decided to just sweep through the entire spectrum from $w_1 = 0$ and $w_2 = 0$ to $w_1 = 5$ and $w_2 = 5$ in steps of 0.1 each.

On each of the two AWS EC2 instances, we ran 4 Jupyter Notebooks in parallel to sweep through the parameters. See below for a plot of the errors:



(a plot of MSE vs. W values)

Model 3: Matrix Factorization

Inputs:

A sparse rating matrix R where at $R_{i,j} = 0$ if the user$_i$ had not rated the book and $R_{i,j}$ = 1,2,3,4,or 5 if the user had rated the book.

Method:

We factorize the rating matrix into two smaller matrices U and V.

UV is our prediction matrix. However, because these two matrices are initially initialized at random, we use the alternating least squares technique to minimize the error at every iteration of our decomposition of matrix R.

Implementation:

In this model, there are 3 parameters unknown to us:

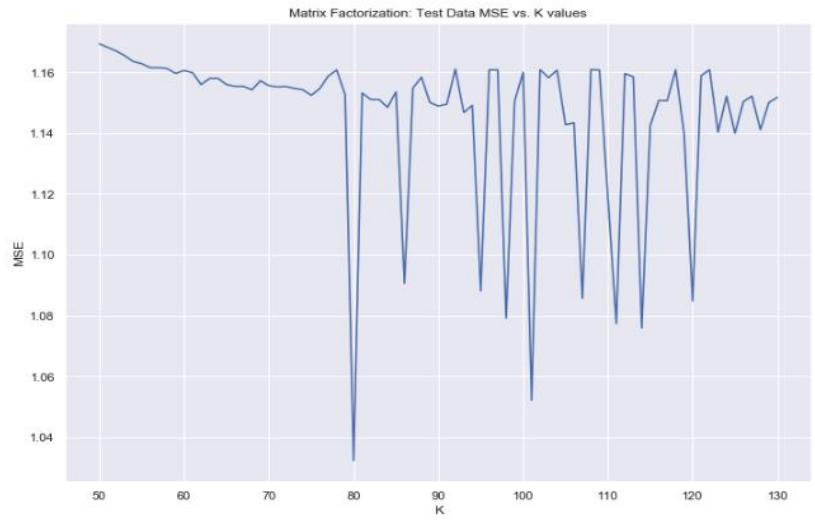
1. *k*, the latent factor of matrices U and V
2. λ, a regularization parameter to prevent overfitting
3. *niter*, the number of iterations that we should run to decompose R into UV.

To find out the best parameters to use for our model, we first experimented on *niter*. We found that *niter* = 20 seems to be a safe number to use as *niter* < *20* was often not enough for the UV^T to produce the lowest mean squared error (MSE), while *niter* > *20* often produced MSE's that are higher than before. Therefore, the parameters left to sweep are *k* and λ.

The rating matrix that we are running on consists of 20000 users and 409 books, which produces quite a big sparse matrix. Because of this, we decided that we should start with a larger *k* to sweep and chose *k = 50* as our starting point and swept up till *k = 130*. For our λ, we realized that the smaller numbers from 1 – 3 usually result in the lowest MSE's. Therefore, we decided that a sweep from $\lambda = 1$ to $\lambda = 10$ will be sufficient for our model.

On each of the two AWS EC2 instances, we ran 15 Jupyter Notebooks in parallel to sweep through the parameters. Below is a plot of parameters and their respective MSE's:

```
Params with lowest test data MSE's are:
K = 80.0
reg = 5.0
MSE = 1.032222617050581
```



Model 4: Neural Network based Matrix Factorization

Architecture

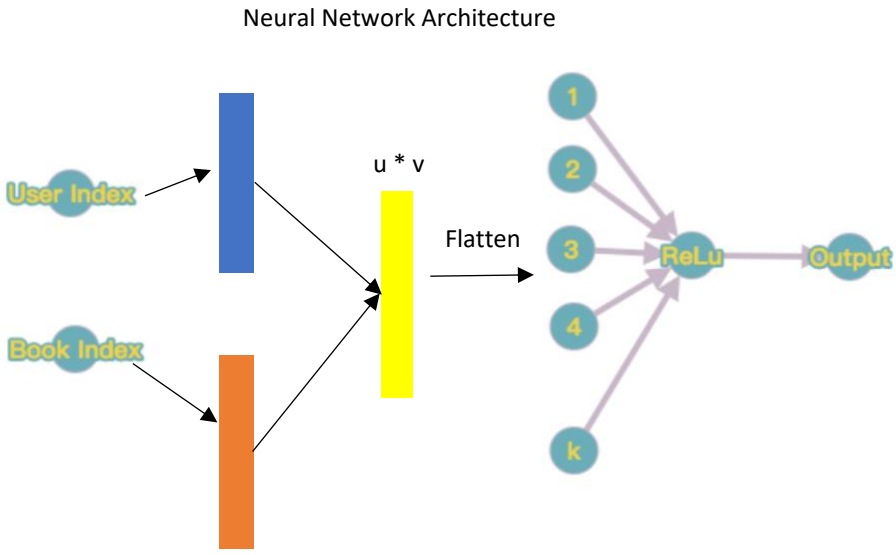
Inputs:
1) user index, an integer;
2) book index, an integer.

Layer 1 Embedding: The user index and the book index are embedded to vectors *u, v in R^k,* where *k* is the dimension of the latent space that we need to specify for the model.

Layer 2 Merge: Vectors *u, v* are element-wise multiplied and the resulting vector is flattened into *k* nodes.

Layer 3: A dense layer of one neuron. We did not want the models to predict negative ratings, so we added ReLu to cut off the negatives.

Output: An integer, the activation of one node connecting to the single neuron in Layer 3.



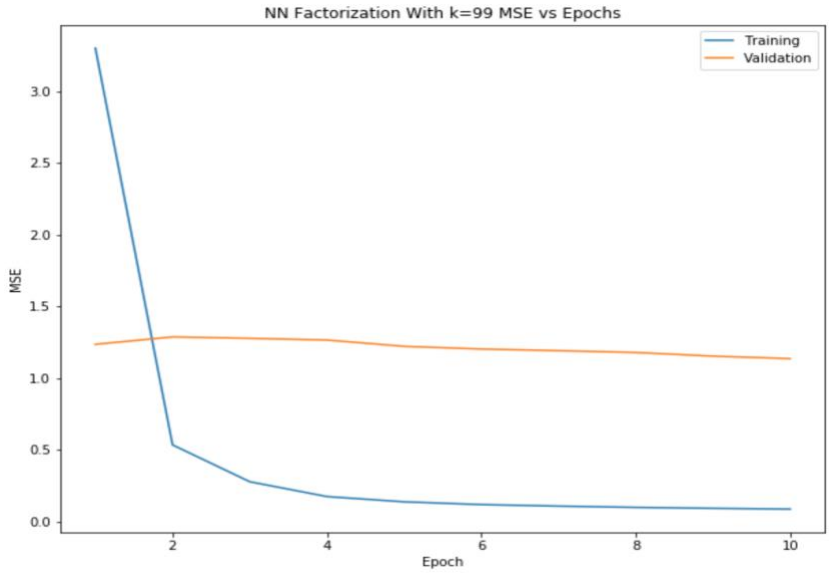
The architecture is shown in the figure above. We learned about this from an article online (link in footnote). This architecture is really mimicking matrix transformation. Vector *u* is just the vector representing a user's preferences and Vector *v* is just the vector representing attributes of a book.

Machine Learning for Recommender systems: https://medium.com/recombee-blog/machine-learning-for-recommender-systems-part-1-algorithms-evaluation-and-cold-start-6f696683d0ed

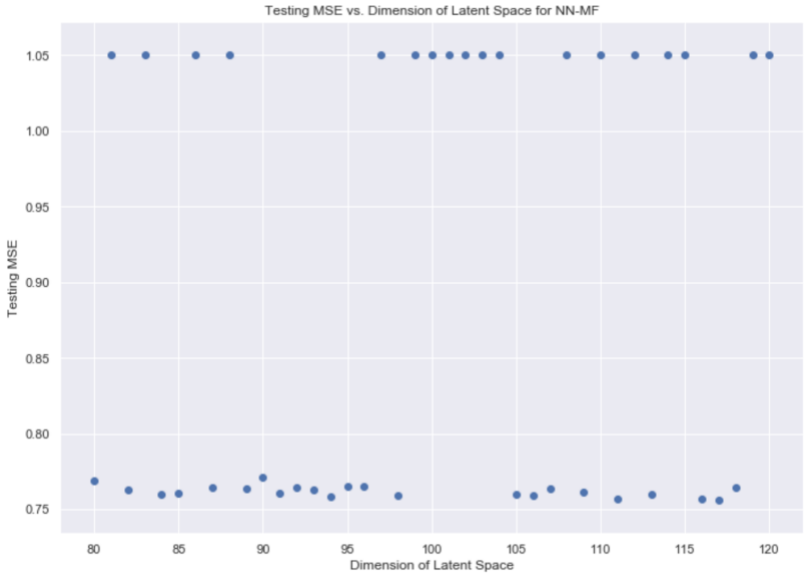
Training Neural Network

For every nonzero entry i, j in the training rating matrix, we converted it into (i, j) as inputs to the neural network and the rating itself is the desired output. We constructed the network using Keras with validation split of 20% and 10 epochs. The loss function we selected is mean squared error and we recorded the losses during the training process.

For the dimension of the latent space, first we tried $k = 99$, which we had found out to be pretty good when sweeping for the matrix factorization model. With $k = 99$, testing MSE comes down to 0.768, which is a 20% improvement over the best model we had, the user-user similarity model. Below is the plot of mean squared error versus epoch.



We then used an AWS EC2 GPU instance to sweep for the dimension of the latent space from 80 to 120. Below is a scatterplot of testing MSE versus the dimension.



There is a strange phenomenon: testing MSEs are either around 0.76 or around 1.05. The MSEs around 1.05 are all actually different values, but very close. We do not yet understand why this happened. We once hypothesized it might be because our testing set was too small, but we tried to increase it 20% and still got the same result.

In addition, we noticed that the $k = 99$ model this time had a testing MSE of 1.05, which was previously only 0.768. We are very confused by these observations and are still trying to understand the reasons behind them.

5. Conclusions

1. Out of all the models, the Neural Network Based Matrix Factorization Model has significantly better testing MSE than the other models. We believe that this is because its architecture very closely mimics matrix factorization. In addition, to train a matrix factorization model, we need to provide the value of the regularizer, which creates more room for errors, whereas we do not need to for the NN-MF model.
2. Our own attempt at linearly combining the three matrices in the weighted user-user model actually did not outperform the baseline, the unweighted user-user model's testing MSE. We believe this is because the way that we are weighting the three matrices is too naïve. For example, not every user who has read a book but has not rated it will eventually give it the same rating. It requires more sophisticated approaches for this model to ever perform better than the unweighted user-user model.
3. A good method that we employed in this project is that we discovered the parameters for our neural network based matrix factorization by sweeping through these parameters in the regular matrix factorization model. Have we swept these parameters on the neural network model, the time needed to find good parameters will take way too long as each neural network model takes about an hour to train while the matrix factorization model takes several minutes.
4. These performance times also lead to our last point: although neural network based matrix factorization models have the lowest MSE, we much more encourage an implementation of unweighted user-user model to be used, unless a better weighted user-user model is found. On *Goodreads* in production, new data is always created as users interact with the books and these models will constantly have to be re-trained on the new data. It would not be very costly if the model takes hours to train. Therefore, because the user-user models have quite a low MSE among the models we tried and takes the least amount of time to train, we recommend using it.

Future Directions

1. We would like to investigate more ways of incorporating shelved matrix and isRead matirx into rating matrix. One idea is to represent each entry of the rating matrix as a 3-tuple. Then, user-user similarity can be computed by computing some measure of similarity between two lists of 3-tuples, or two matrices.
2. There are more neural-network based models for recommender system that we would like to explore. For example, there is an architecture that concatenates the embedded user vector and book vector instead of multiplying them element-wise. We can employ more unique kinds of neural architecture and test out different k's to see which architecture actually performs the best.
3. We should employ cross-validation when testing the models.
4. We would like to use larger subsets of the original dataset we obtained to test the four models. We want to see if these models can retain their testing MSEs. Also, we want to train and test these models on other datasets, such as MovieLens, so that we have more models we can find in literature that we can compare our models to. This may answer our questions regarding the irregularities of the neural network based matrix factorization model.

References

1. Data source: https://sites.google.com/eng.ucsd.edu/ucsdbookgraph/home
2. JSON to CSV converter: https://github.com/Yelp/dataset-examples/blob/master/json_to_csv_converter.py
3. Wan, Mengting and Julian McAuley. "Item recommendation on monotonic behavior chains." *RecSys* (2018).
4. Neural Network Based Matrix Factorization: https://medium.com/recombee-blog/machine-learning-for-recommender-systems-part-1-algorithms-evaluation-and-cold-start-6f696683d0ed