**Comparison of Three Recommender Systems on Goodreads Dataset**

**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 three recommender systems we have built are a matrix factorization model, a weighted user-user similarity model, and a neural network-based matrix factorization model. We will be comparing and analyzing which model performs the best.

**Data:**

Data Clean

We obtained a *Goodreads* dataset 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 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 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.

Extracting Matrices

Using three columns of the dataset – *user\_id*, *book\_id*, *isRead* – we converted it to a Scipy COO sparse rating matrix. There was a 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.

Now the data is clean, we can then extract a 679113 by 14473 rating matrix based on the 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 useri has shelved bookj, 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 rating matrix we worked with in this project is thus 20000 by 426.

Train-Test Split

We employed a different methodology 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.

Data Source: https://sites.google.com/eng.ucsd.edu/ucsdbookgraph/home

**Recommender Models**

Model 1: Weighted User-User Collaborative Filtering

Inputs:

1. A sparse matrix S where at Si,j = 0 if useri had not shelved bookj and Si,j = 1 if useri had shelved the book.
2. A sparse matrix Q where at Qi,j = 0 if useri had not read bookj and Qi,j = 1 if useri had read the book.
3. A sparse rating matrix R where at Ri,j = 0 if the useri had not rated bookj and Ri,j = 1,2,3,4,or 5 if useri had rated bookj.

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.

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:

*user1* rated the book with a score of 3.

*user2* read the book but didn’t rate the book.

*user3* shelved the book but didn’t finish reading the book.

If we proceed with the unweighted user-user model, both *user2* and *user3*will have the same similarity score when compared to *user1*. However, we believe that because *user2*actually finished reading the book, *user2* has a more meaningful interaction with the book than *user3* and therefore *user2*should be more similar to *user1*. 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 model, the similarity measures we used is the Pearson correlation coefficient. To have a benchmark for our model, we first ran an unweighted user-user model to find out the MSE of this model. After doing so we found out that the unweighted user-user model has a quite a low MSE:

*MSE = 0.9756388191656609 (unweighted user-user model)*

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

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

*M = (w1 \* S) + (w2 \* Q) + R*

where *w1 < w2 < 5* and *M* is the final matrix inputted into the unweighted user- user model. *w1* should be less than *w2* 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:

Here, M1,2 = 0.3 and M2,1 = 0.8 both capture what we want. M1,2 shows that users who have shelved the book should have some weight in the final matrix but this weight should be the smallest. In M2,1, 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 M2,2 = 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:

Mi,j = *(w1 \* Si,j) + (w2 \* Qi,j)*

*else:*

Mi,j =  *R*i,j

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

where *x = 4,* so

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: *w1* and *w2.*

Our initial prediction is that *w1* and *w2*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 *w2.* 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 *w2* **=** 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 *w1* **=** 0 and *w2* **=** 0 to *w1* **=** 5 and *w2* **=** 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.

PLOTTT – waiting to finish running

Model 2: Matrix Factorization

Inputs:

A sparse rating matrix R where at Ri,j = 0 if the useri had not rated the book and Ri,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.

UVT 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:

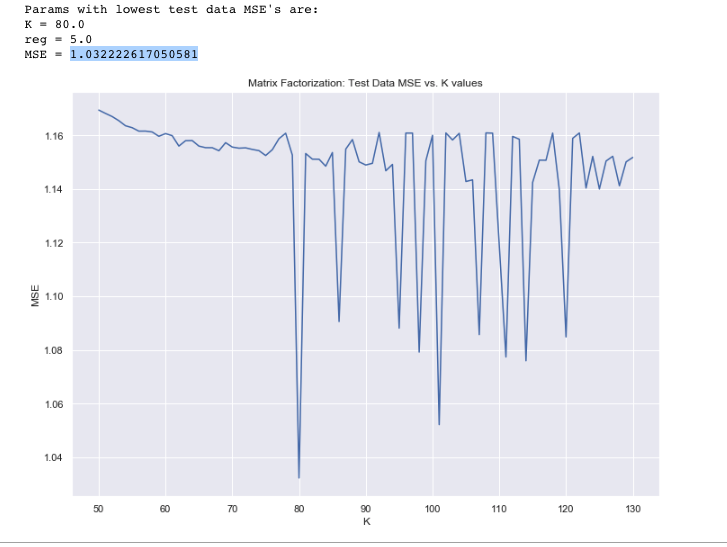
In this model, there are 3 parameters to us are unknown to us:

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 UVT 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 *λ = 1* to *λ = 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:



(a plot of MSE vs. K values)

Model 3: Neural Network based Matrix Factorization

- inputs: user i, book j

- layer 1: 1) embedding of user i into vector u in R^k

2) embedding of book j into vector v in R^k

- merge: multiply u and v element wise

- flatten

- layer 2: dense 128

- layer 3: dense 128

- layer 4: dense 1, which is the output

**Conclusions**

User-User Models

Unfortunately, our weighted user-user model did not outperform the unweighted user-user model (MSE = 0.9756)

regardless of what weights we swept and inputted. This was out of our expectation as we thought the extra interactions we included might provide a better model. A possible reason that the weighted user-user model is that we’re working with a smaller subset of the entire dataset and there are just too few books or users for our model to work.

Matrix Factorization and Neural Network

Our regular matrix factorization has a higher MSE (1.032) than that of both the weighted and unweighted user-user models. However, our neural network actually

Although a regular matrix factorization recommender system has a higher MSE than the user-user model, using a neural network based matrix factorization, we’re able to train a model that has a lower MSE than the user-user model.

Fastest performance: Unweighted User-user

**Future Directions:**

**References:**

https://sites.google.com/eng.ucsd.edu/ucsdbookgraph/home

https://cseweb.ucsd.edu/~m5wan/paper/recsys18\_mwan.pdf

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