Quarantined Quants: Big Data Basic Recommender System and Extensions

Amanda Kuznecov (anr431), Sophia Tsilerides (smt570), Ilana Weinstein (igw212)
DS-GA 1004 - New York University

1. OVERVIEW

In the age of information, recommendation systems have become an integral part of making sense of large collections of data. People now have access to an abundance of resources on online platforms compared to the constricted selection in physical stores. Many tech companies including Amazon, Netflix, Facebook, and Linkedin use recommendation systems to contextualize their customers and provide a personalized preference prediction for their millions of products to enhance the traditional search experience.

In this paper, we use Spark's MLlib Alternate Least Squares (ALS) matrix factorization method to make book recommendations for users in the Goodreads dataset collected by Wan and McAuley. The dataset has a total number of 228,648,342 records with 876,146 users that have given 104,551,549 greater than 0-star ratings to 2,360,651 books. Altogether, the users have read 112,131,203 books! There are no duplicate `'user_id'` and `'book_id'` pairs in the dataset. The most read book is Book 943 with reviews from 314,685 distinct users and User 320562 has given the most reviews: 38,884 greater than 0-star reviews to be exact. The distribution of the ratings is shown in Table 1.

Table 1: Ratings Distribution

Rating	Count
0	124096793
4	37497451
5	35506166
3	23307457
2	6189946
1	2050529

2. DATA PROCESSING

As part of our data pre-processing, we removed any users from the data set that had an `'is_read'` value of zero. Theoretically, this could mean the user showed interest in the book by placing it in their cart or adding it to their favorites, but never performed the action of purchasing it, so they would not be able to provide a real rating. To confirm, we queried the `goodreads_interactions` file with the SQL clause `spark.sql('SELECT * FROM data WHERE is_read == 0')` and then filtered for distinct values in `'rating'` with the operation `.select('rating').distinct().show()`. The results showed that all records with a value of zero for `'is_read'` also had zero values for `'rating'` which confirmed our theory and proved we could not train on these instances. The records were removed by filtering with the SQL clause `.where('is_read > 0')` on the original `goodreads_interactions` file, and the resulting DataFrame was parqueted for efficient data splitting. Before performing modeling, we also removed any other books with a zero rating that this exercise omitted.

To split the dataset into training, validation and testing sets, first all users with less than ten interactions were removed from the original dataset by aggregating the data on `'user_id'`, performing a count, and filtering. Next, the distinct User IDs left were randomly split 60% for train, 20% for validation and 20% for testing using the `.randomSplit` function. The three sets of users were joined with their interactions using a left join operation on the original data set: `users.join(data, on=['user_id'], how='left')`. Lastly, half of the validation, and test user's interactions were concatenated to the train dataset. This process is motivated by the idea that users with absent history can not generate predictions. This was done using the `pyspark.sql.window` package to create subgroups of the DataFrames called "windows" partitioned on `'user_id'`.

3. MODEL AND EXPERIMENTS

Spark's MLlib library has the matrix factorization algorithm Alternate Least Squares (ALS) that is capable of predicting the future preference of a set of items for a user. Matrix factorization decomposes a dataset into some smaller representations to find latent factors. That is, $\hat{r}_{u,i} = q_i^T p_u$ where $\hat{r}_{u,i}$ is the predicted ratings for user u and item i, and q_i^T and p_u are latent factors for item and user, respectively. However, $q_i^T p_u$ is non-convex, which ALS addresses by alternating fixing q and p until convergence [1]. ALS runs its iterative computation in parallel across multiple partitions for efficiency. We train the ALS model on five percent of total users in the Goodreads dataset.

Hyperparameter tuning was performed on model parameters *rank* and *regParam* to optimize model performance. The *rank* parameter refers to the anticipated number of latent factors, contributing directly to the dimensionality of the model and, consequently, complexity. A lower value for *rank* produces broader model trends,

while a higher value will lower model generalizability. The parameter *regParam* controls regularization and helps to reduce overfitting by penalizing the model inputs. Tuning was conducted on the validation set in order to extract hyperparameter values that yielded the best possible model performance. With good performance on the validation set, the final model was configured with these same hyperparameters to make book recommendations for users in the hold-out test set. The default value for *maxIter* is ten, but we lowered it to five for faster evaluation.

Models were tuned using Root Mean Square Error (RMSE) because it was a quick and standard way of measuring error as the distance between predicted and observed values. In general, lower RMSE values indicate better model quality, as there are fewer incorrect predictions. Results for hyperparameter tuning on 25% of the dataset are in Table 2. Based on the results, *rank* = 2, and *regParam* = 0.1 produced the best model.

Table 2: RMSE Hyperparameter Tuning Results

Rank	Regularization Parameter				
	0.001	**0.01**	**0.04**	**0.1**	**1**
2	1.02	0.86	0.78	0.77	1.36
10	1.05	0.96	0.85	0.79	1.35
25	1.06	1.00	0.82	0.80	1.35

4. EVALUATION

RMSE is not the best metric to evaluate recommender systems because it disproportionately penalizes bad predictions and does not take into account user behavior. Mean Average Precision (MAP) treats recommendations like a ranking task, where the order of the result set matters. Books at the front of the list are more relevant, compared with books near the end, which are less relevant. It is thus a useful metric for recommender systems where typically, the top k items are presented to the user as recommendations. MAP is calculated based on the following formula:

$$MAP = \frac{1}{M}\sum_{i=0}^{M-1}\frac{1}{D_i}\sum_{j=0}^{Q-1}\frac{rel_{D_i(R_i(j))}}{j+1}$$

where $rel_D(r)$ refers to the relevance score of a recommended book, assigning one to a recommended book that is considered a relevant book, and zero when it is not. For each user, these values are summed and then divided by the total number of relevant books, yielding the average precision for a given user. The average precision for each user is then summed and divided by the total number of users [2].

MAP takes as input a list of predicted books and a list of "ground truth" books presented in descending order of their rating. For the list of predicted books, additional pre-processing was required on the predicted ratings DataFrame. Each recommended book/rating pair for a given user was aggregated into a single list using `collect_list` and a window to rank the books in descending order of rating, creating a DataFrame consisting of `'user_id'` in one column, and an ordered array of recommended books in the second column [3]. A similar task was performed on the "ground truth" list, which was taken from the validation or test data, and then grouped into a single list of books for each user presented in descending order of rating. Since the "ground truth" list consists of books the user likes, we judiciously chose a rating less than three to be considered irrelevant.

Upon implementing MAP as an evaluation metric, we discovered that there were two different ways to do so, both producing much different MAP values. The first approach uses `model.recommendForAllUsers(500)` to force the model to make 500 predictions for each user. The issue with this method is that using a small dataset will produce shallow MAP values because the model will not be able to make 500 good recommendations based on limited information. Thus, it is beneficial to use the full dataset, which allows the model to learn more about user-book interactions and make better predictions. We verified the number of predicted books per user to be 500 by evaluating the size of the array in predictions using the operation `select(size('books')).show()`. The second approach involved evaluating MAP using the operation `model.transform(val_data)`, in which the model selects an arbitrary number of predictions to make, which are considered good. Then, by applying `rank <= 500` to each user, we removed any predictions for a given user that were not in its top 500. Using this method on a small dataset will produce much less than 500 predictions, but MAP values are higher because there are fewer, and more accurate predictions. In fact, MAP values using this method were always close to one regardless of hyperparameter values. For comparison, we have included both MAP evaluation methods in the evaluations.py file. Since we were unable to run large portions of the data, the second MAP method is more appropriate and more feasible for our evaluations, as shown in Table 3.

Based on the results, it is evident that with a larger subset of the data, we can achieve much better MAP values. This is because the model can learn more interactions between users and books, and therefore it can make more informed decisions about which books to recommend to

users. We can see that the validation and test sets both performed the best with 25% of the data.

Table 3: Evaluation Results

Data Subset		RMSE	MAP (Method 1)	MAP (Method 2)
Validation	1%	1.29	0.00	0.954
	5%	1.26	9.870e-09	0.988
	25%	1.18	*	0.995
Test	1%	1.45	0.00	0.958
	5%	1.28	*	0.987
	25%	1.24	*	0.995

Note: * indicates the model run was killed by YARN for exceeding memory limits

5. EXTENSIONS

5.1 LightFM

To compare the performance of Spark's parallel ALS model, we used LightFM, which is a single-machine implementation of matrix factorization algorithms. LightFM models were created by altering the hyperparameter *no_components*, which is somewhat equivalent to the *rank* hyperparameter used in ALS. It appeared that *no_components* did not change the resulting evaluation metrics by much, however, we noticed that a larger value for this parameter caused run times to be longer. Therefore, we selected 30 as a reasonable parameter value. We used the 'warp' loss function as it optimizes the rank of positive instances by continuously sampling negative instances until it identifies a rank violation.

LightFM requires that input to the model be represents a sparse user-item interaction matrix with all positive instances represented by their ratings, and all negative instances to be represented by zero [4]. We considered all ratings less than three to be negative ratings. First, we created a dictionary with each key representing each column of our input data. Then, we transformed the dictionary into a sparse matrix by first converting to Pandas, and then using the operation `coo_matrix` [5].

Model fit times for ALS and LightFM are compared in Table 4. In general, LightFM is much quicker to fit than the ALS model. This is because LightFM uses Stochastic Gradient Descent to arrive at a solution which requires more iterations, but is very fast to converge, while ALS is slow due to cubic time complexity from having to update one matrix at a time while fixing the other. Additionally, ALS model fitting time varies as the dataset size increases. The 1% subset likely takes the longest because it does not have much information, and therefore the model is trying harder to interpret or find more hidden interactions with what data it is given. However, as more data is added, it seems that fitting times get quicker, which is likely due to the explicit inclusion of more user-item interactions rather than the model having to make its own interpretations. Notably, the fitting time for LightFM models does not change with dataset size because the method is scalable and produces efficient solutions no matter dataset size [6].

Table 4: Model Fitting Times

Model	Data Subsets		
	1%	5%	25%
ALS	4.03min	0.99min	2.34min
LightFM	10.9ms	10.9ms	10.8ms

LightFM has a built-in precision@k evaluation metric which we used to compare against values produced by the ALS model. Precision@k has a very similar definition to MAP, except it does not take into account the order of predictions, so each correct prediction is not divided by its order in a ranked list. The equation for precision@k is as follows [2]:

$$Precision\ at\ k\ = \frac{1}{M}\sum_{i=0}^{M-1}\frac{1}{k}\sum_{j=0}^{min(Q_i,k)-1} rel_{D_i}(R_i(j))$$

As discussed in the previous section, since we were unable to run our model on more than 25% of the data, making 500 item predictions for each user is not appropriate based on these smaller data sizes. Therefore, for this comparison, we decided to evaluate precision@k where k is equal to the percentage of the dataset being used; ie. for 1%, we would choose k to be 1% of the 500 item predictions, so it would be 5. A comparison between ALS and LightFM for precision@k is presented in Table 5. Unfortunately, because some models were able to run, it was hard to compare the values, however, research shows LightFM generally performs better.

Table 4: Evaluation Comparison

Model	Data Subsets		
	1%	5%	25%
ALS	0.00	0.00	*
LightFM	0.00	0.00127	0.00102

Note: * indicates the model run was killed by YARN for exceeding memory limits

5.2 Visualization

To understand model performance, we chose to visualize the high-dimensional latent factors with the T-distributed Stochastic Neighbor Embedding (t-SNE) machine learning algorithm. T-SNE Is able to transform high-dimensional data to be viewed on a low-dimension space. Theoretically, the user and item latent factors of a high performing model will have visible clusters. If it does not, it will be an indicator that the model can predict accurately.

We chose to collect latent factors of the ALS model starting on 1% of the data. Although a rank of two was the best for ALS model after tuning, we chose to visualize with rank five in order to get the most out of the t-SNE algorithm. Additionally, we also chose to incorporate book's genre as the label, hypothesizing that similar users and items can be grouped based on genre.

After fitting the model, ALS's `model.userFactors` was used to obtain the user latent factor data frame. The data frame contains a feature vector column for each user which was expanded to have a column for each factor. Once the 500 books were predicted for each user, we found the top predicted book for each user and left joined the `'book_id'` onto the feature matrix. Then, the genre of each book was mapped from the Goodreads' genres file and mapped by the correct `'book_id_csv'` from `book_id_map.csv`. This data was then moved locally for t-SNE.

After mapping the top genre of the top predicted book to the user latent factor matrix, we were able to run t-SNE with perplexity of 84 and 1000 iterations. According to a study done by Nikolay Oskolkov, the square root of the length of the data is the optimal perplexity parameter and 1000 iterations is common practice [7]. With these paramters we obtained Figure 1. There are some promising clusters between 'fiction', 'mystery' and 'romance' but the clusters are not defined enough to conclude that the model will perform well.

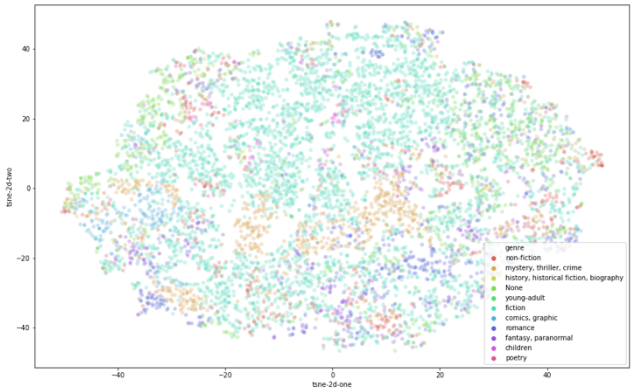

Figure 1: TSNE User Latent Factors (5) – Top Predicted Book Genre

Similar to the process for user latent factor matrix, the method `model.itemFactors` was used to obtain the item latent factor data frame and it was mapped to book genre. Initially, we wanted to follow Oskolkov's optimal hyper-parameter rule, but it was not possible on our machines as the matrix contains 277,551 items with a perplexity of 526. Instead, we performed t-SNE on all items with a perplexity of 100 and based off of those results chose three genres that showed potential clustering. The result of the entire item matrix can be seen in Appendix A: Figure 1, where the genres chosen were non-fiction, fiction and romance. The three genres had a total of 151,605 items and t-SNE used a 389 perplexity. The results in Figure 2 show that, based on available data, it is just as difficult to group items as it is users. Our interpretations from these visuals align with the poor model performance.

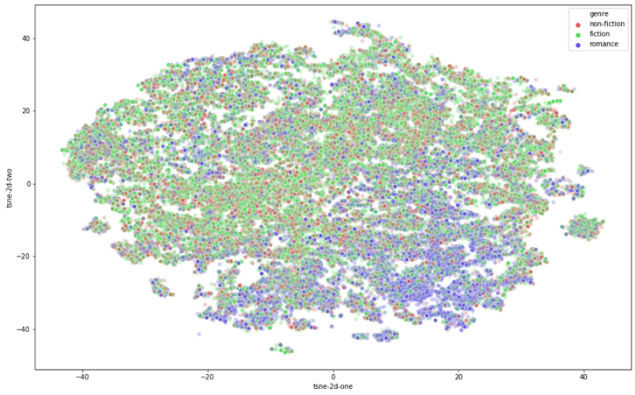

Figure 2: TSNE Item Latent Factors (5) – 3 Book Genres

The top book recommendation as a label on user latent factor matrix of five and ten latent factors was also explored. The noise with ten latent factors led us to continue with only five latent factors and the excessive 333 book recommendation labels led to incorporating genres. All commands to collect the data can be found in `tsne_data.txt` and all t-SNE models and graphs can be found in `TSNE.ipynb`.

CONTRIBUTIONS

Project work was distributed after thoroughly discussing ideas and performing exploratory analysis on the dataset in weekly meetings. Sophia performed data splitting and wrote the modeling scripts. Amanda wrote the ALS evaluation metric scripts and performed modeling and evaluation on LightFM. Ilana implemented t-SNE and interpreted its results.

REFERENCES

[1] Shapiro, Heather B. "Spark Collaborative Filtering (ALS) Deep Dive." Microsoft Azure Notebooks - Online Jupyter Notebooks, notebooks.azure.com/heatherbshapiro/projects/recommenders/html/notebooks/02_model/als_deep_dive.ipynb.

[2] "Evaluation Metrics - RDD-Based API." Evaluation Metrics - RDD-Based API - Spark 2.4.5 Documentation, spark.apache.org/docs/latest/mllib-evaluation-metrics.html.

[3] Chen, Vinta. "Spark ML Cookbook (Python)." I Failed the Turing Test, 21 Oct. 2019, vinta.ws/code/spark-ml-cookbook-pyspark.html.

[4] Safdari, Nasir. "If You Can't Measure It, You Can't Improve It !!!" Medium, Towards Data Science, 2 Dec. 2018, towardsdatascience.com/if-you-cant-measure-it-you-can-t-improve-it-5c059014faad.

[5] Kula, Maciej. "Welcome to LightFM's Documentation!¶." Welcome to LightFM's Documentation! - LightFM 1.15 Documentation, making.lyst.com/lightfm/docs/home.html.

[6] Yu, Hsiang-Fu, et al. "Parallel Matrix Factorization for Recommender Systems." University of Texas, 2013, www.cs.utexas.edu/~inderjit/public_papers/kais-pmf.pdf.

[7] Oskolkov, Nikolay. "How to Tune Hyperparameters of TSNE." Medium, Towards Data Science, 19 July 2019, towardsdatascience.com/how-to-tune-hyperparameters-of-tsne-7c0596a18868

APPENDIX A

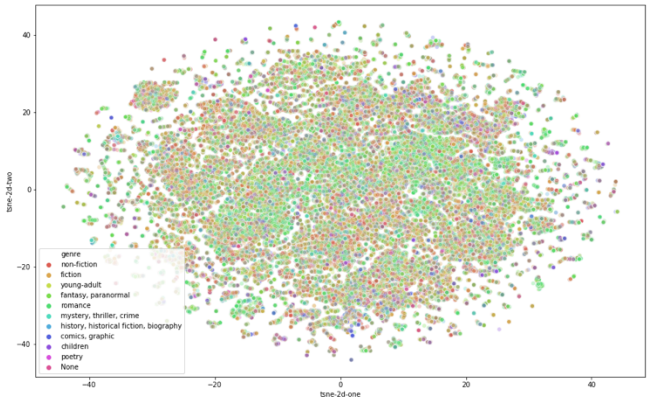

Figure 1: TSNE Item Latent Factors (5) - All Book Genres