Prototyping an ALS Recommender System

Final Report

Date: 2020-05-11
Name: Silas Mann
NetId: manns03
NYU email address: manns03@nyu.edu

Overview

The UCSD Book Graph provides public user-book interactions data scraped from goodreads.com (1). The CSV file provided (“goodreads_interactions.csv”) contains 228,648,342 interactions between 876,145 users and 2,360,650 books. The goal of this project is to leverage these data to prototype a book recommender system to that generates highly individualized book recommendations.

The main problem faced when creating such a recommender system is how to handle the sparsity of the data: most users only read a tiny fraction of the total books. Without a strategy for extracting latent factors from the interactions, it would only be possible build a model that provides recommendations for the most popular books without catering to the individual preferences of the users. This project also explores t-SNE as a method for visualizing the distribution of the books in the learned item space.

Data Processing

The first step of my data processing pipeline was to read the csv into Spark and save the dataset in parquet format for more efficient processing. I was only interested in ratings provided for users who actually read the books, so before saving I filtered out any interactions where `is_read=0`.

```
spark.read.csv(path/to/goodreads_interactions.csv', header=True, schema='user_id INT, book_id INT, is_read INT,
rating INT, is_reviewed INT').filter("is_read=1").write.parquet('goodreads_interactions.parquet')
```

After this step, 112,131,203 interactions (~50%) remained in the dataset. To efficiently develop and test the recommender system, I also downloaded a subset of the interactions for just one genre (poetry). The subset included 2,734,350 interactions, representing only 1.2% of the total interactions.

`sub_split.py:`

To eliminate users with few interactions, I began by filtering to with at least 10 interactions. I created downsampled subsets of the interactions at 1%, 5%, and 25% to allow for testing in smaller batches.

For both the full datasets and the subsets, I randomly split users into training, validation, and test sets at a ratio of 60:20:20. In lieu of a cold start approach, ALS requires interactions in the training data for each of the users from the validation and test sets in the training set. To achieve this, I joined half of the interactions for validation and test users to the training set. Finally, I removed interactions for books with no training data from the validation and test sets. I saved all results in parquet format for efficient processing.

Model and experiments

The approach used in this project was to build an Alternating Least Squares (ALS) model using Apache Spark (via the Python API *PySpark*). ALS is a parallelizable matrix factorization algorithm that decomposes user-item interactions into the product of two lower-dimensional matrices: one representing latent factors for users, and the other for items. Scores are calculated as a dot product of vectors from these matrices.

I selected two hyperparameters to tune: `regParam` (the regularization parameter), and `rank` (the target number of latent factors).

For my metric, I chose mean average precision (MAP). I reasoned that for this application:

- RMSE evaluates the prediction score (the standard deviation of the residuals) for the user-book interaction against the actual rating. For a recommender system, we more interested in rank than precisely predicting the score, thus RMSE has limited value for evaluation.
- Precision at k does not account for the order of the recommendations, but to suggest books to users it is important for the most relevant books to show at the top of the list.
- NDCG at k would have been my ideal choice, because ideally it can account for order of the predictions in addition to the relevance of the truth values. However, in the scala source code for RankingMetrics, I discovered a comment that this implementation only supports binary relevance.
- MAP evaluates the number of books recommended for a user that are in the truth set for that user, but adds a penalty based on the order in which they are recommended. Ideally, I wanted to model to recommend higher rated books before others, and MAP is a better metric for this than RMSE or precision at k because it prioritizes rank over accuracy or precision. I chose the truth set for each user to include only books with a rating of 3 or more.

`als_rec.py:`

During testing on the smaller poetry subset, I tried several strategies (also commented in the code).

- Option 1: Transform the validation set, select the top 500 predictions for each user
- Option 2.1: Recommend 500 books for all users in the validation set
- Option 2.2.1: Same as 2.1, but remove interactions from the training set
- Option 2.2.2: Same as 2.1, but keep only interactions in the validation set

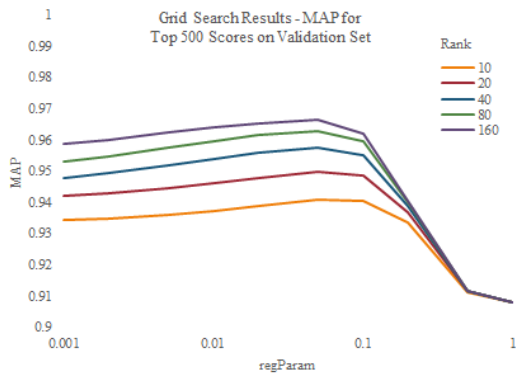
Initially I tested Option 1, which provided very reasonable-looking results, so I immediately started a grid search to identify the best hyperparameters on the HPC, working up from 1% to 100% of the data.

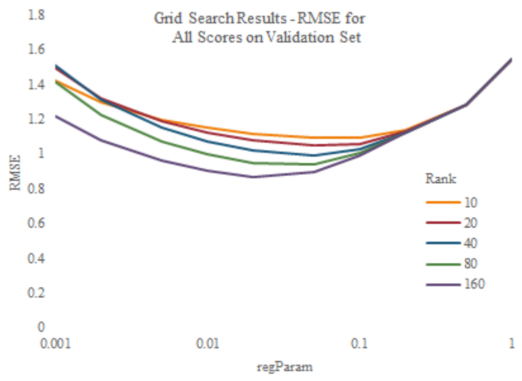
While that job was running, I continued to test locally. I realized that by transforming the validation set, I was likely discarding a good deal of the top 500 predictions that the model would have made for each user if it were not restricted to evaluating interactions in the validation set. This led me to option 2.1: recommend 500 books for each user. I found that this resulted in extremely low scores, which I initially thought was incorrect. I reasoned that I should exclude predictions from the training set, as these were polluting my results, leading me to Option 2.2.1: remove training interactions. My results were still low, and I realized that it was because there were a vast number of books being recommended relative to the small number of books with ratings greater than 3 for any given user. This led me to Option 2.2.2: eliminate all interactions outside the validation set. This artificially increased the score but was reasonable given that the list was still limited to the true top 500 books predicted for each user and the order was maintained.

After testing each of the above options, I decided that because I was comparing relative metrics, the best method was Option 2.1: keep it simple, run metrics on all of the top 500 results. I began running a grid search with this approach, but due to limited resources available on the HPC cluster, I have only preliminary results from this analysis (Addendum 1). The results of hyperparameter tuning that I present here are based on the transformed validation set. While this approach does explicitly test the true top 500 recommendations, I believe it was still effective for tuning hyperparameters to improve the model.

For my grid search, I tested the following parameters:

```
ranks=[10,20,40,80,160]
regParams=[.001,.002,.005,.01,.02,.05,.1,.2,.5,1]
```





For all ranks tested, the best results were found with `regParam=.05`.

Metrics generally improved with increasing rank, but only marginally above `rank=50`, even when scaling exponentially. I also considered my choice in extension, noting that in the sklearn documentation for t-SNE, “It is highly recommended… to reduce the number of dimensions to a reasonable amount (e.g. 50) if the number of features is very high”. Taking into account these considerations, I chose `rank=50` for my final combination of hyperparameters.

Using my initial strategy of choosing 500 top predictions from the validation set, these are the final scores on the hold-out test set from the 100% split with `rank=50` and `regParam=.05` (see Appendix 2 for additional metrics):

- **MAP at k=500: 0.960**

Extension

For my extension, I chose to explore the books’ latent factors with visualization of t-SNE (5). While exploring CUDA applications in Python, I discovered `cuda-tsne` (7), an implementation of t-SNE that leverages a technique for accelerating t-SNE from a follow-up paper (6) published by the author of the original t-SNE technique.

In order to gain insight into the books’ latent factors found by the ALS model, I chose to perform t-SNE in 2 dimensions and create a scatter plot of the results, coloring each point differently based on the book’s known genre. I hypothesized that the points would tend to cluster together based on genre.

The `cuda-tsne` package cannot be run on Windows Subsystem for Linux, so I installed Ubuntu along with the package dependencies, including NVIDIA’s CUDA Toolkit (8). I was unable to install `cuda-tsne` using `conda` due to a CUDA version issue, so I cloned the github project and built it from source, using the flag `-DWITH_ZMQ=TRUE`. The `libzmq-dev` required for visualization via ZMQ is no longer available, but I was able to use `libzmq3-dev` instead. Unfortunately, the resulting animated visualizations did not allow for coloring points based on genre and thus were of limited value for my purposes.

`get_top_genres.py:`

The UCSD Book Graph provides “fuzzy” book genres (“goodreads_book_genres_initial.json”), which are based on the virtual ‘shelves’ on which users have placed each book. To limit to one per book, I chose the most frequently selected category for each book as the genre. For each genre, I created a numeric genre ID, and I then mapped each book to the book ID used in the interactions dataset (“book_id_map.csv”).

`merge_top_genres.py:`

Once I had my top genres, I joined them with the latent factors for the books produced by ALS and saved the results as a CSV file for processing within Python.

`genre_tsne.py:`

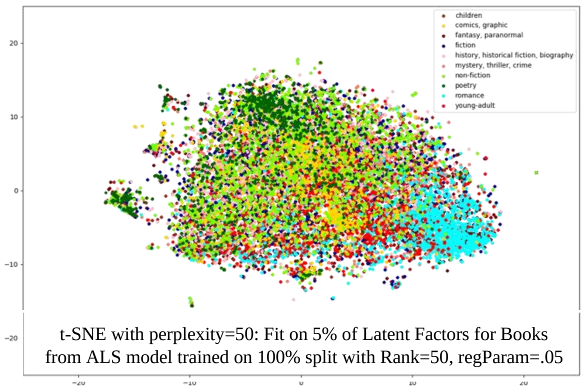
I fit the t-SNE model to the latent factors, transformed to 2 dimensions, and visualized the resulting points. At this phase, I performed many additional iterations of testing. I began with the poetry subset[1] before progressing to the final results, which were generated on the latent factors for books from the best model using the 100% split. Tests included…

- training on the full dataset vs. training on a subset (`downsample_by_pct`);
- plotting the full transformed dataset vs. plotting a subset (`pct=print_pct`);
- downsampling to even numbers by genre prior to fitting the model (`downsample_to_min`);
- downsampling to even numbers by genre before plotting (`downsample=True`); and
- visualizing the merged results (`print_tSNE`) vs visualizing individually (`save_tSNE_split`)

The best clustering resulted from downsampling the dataset to 5% of the books, fitting the model, and then downsampling to even numbers by genre before plotting. I was unable to improve results by using 100% of the data by altering t-SNE parameters; for example I saw no improvement when increasing the number of iterations from 100 to 200, 500, 1000, 2000, 5000, or 10000.

One of the main t-SNE parameters is perplexity, for which I found some documentation (10) but had limited success in tuning. When set to 30 or below, I did not get any results at all. I tested 40, 50, 60, 100, 150, and 200, and I found that `perplexity=50` consistently produced marginally better results in my visualization by visual assessment.

To further test my input, I ran PCA on the books’ latent factors and found that less than half of them contributed to more than 90% of the variance. Interestingly, however, I found that the best results from t-SNE came from a dataset with high dimensionality, comparing tests using output from `rank=3`, `rank=10`, and `rank=50`.



t-SNE with perplexity=50: Fit on 5% of Latent Factors for Books from ALS model trained on 100% split with Rank=50, regParam=.05

In my final results, there appeared to be some clustering in every group (Appendix 3). However, in reading about the technique (11), I believe some of this may be attributed to random noise. Nonetheless, I was able to clearly identify clusters representing “romance” and “poetry” consistently throughout my testing. The nature of the fuzzy genres may not lend

[1] Interestingly, most but not all books from the poetry subset fell into the “poetry” genre; in fact, all genres were represented in the dataset. Exploring the individual titles, it made sense—books in this dataset include historical texts on poetry, children’s books of poetry, romantic poetry, etc.

itself to clustering (for example “fiction” is a very broad category), but possibly more likely, the latent factors may represent something more nuanced than these simple genre categories.

Contribution of team members

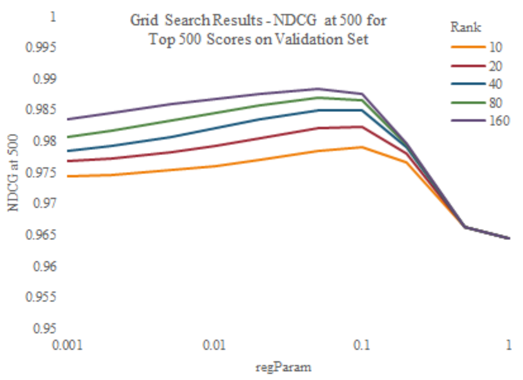
Silas Mann - All work on this project was completed by me.

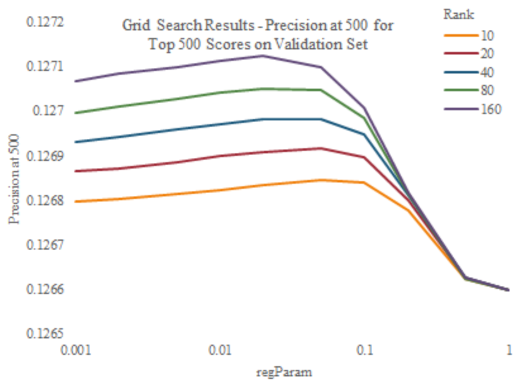
References

1. Goodreads
https://sites.google.com/eng.ucsd.edu/ucsdbookgraph/home
2. Spark ML cookbook
https://vinta.ws/code/spark-ml-cookbook-pyspark.html
3. MRR vs MAP vs NDCG: Rank-Aware Evaluation Metrics And When To Use Them
https://medium.com/swlh/rank-aware-recsys-evaluation-metrics-5191bba16832
4. The confusion over information retrieval metrics in Recommender Systems
http://www.claudiobellei.com/2017/06/18/information-retrieval/
5. Visualizing Data using t-SNE
https://lvdmaaten.github.io/publications/papers/JMLR_2008.pdf
6. Accelerating t-SNE using Tree-Based Algorithms
http://lvdmaaten.github.io/publications/papers/JMLR_2014.pdf
7. TSNE-CUDA
https://github.com/CannyLab/tsne-cuda
8. CUDA Toolkit
https://developer.nvidia.com/cuda-toolkit
9. ZeroMQ
https://github.com/zeromq/libzmq/
10. Visualising high-dimensional datasets using PCA and t-SNE in Python
https://towardsdatascience.com/visualising-high-dimensional-datasets-using-pca-and-t-sne-in-python-8ef87e7915b
11. How to Use t-SNE Effectively
https://distill.pub/2016/misread-tsne/

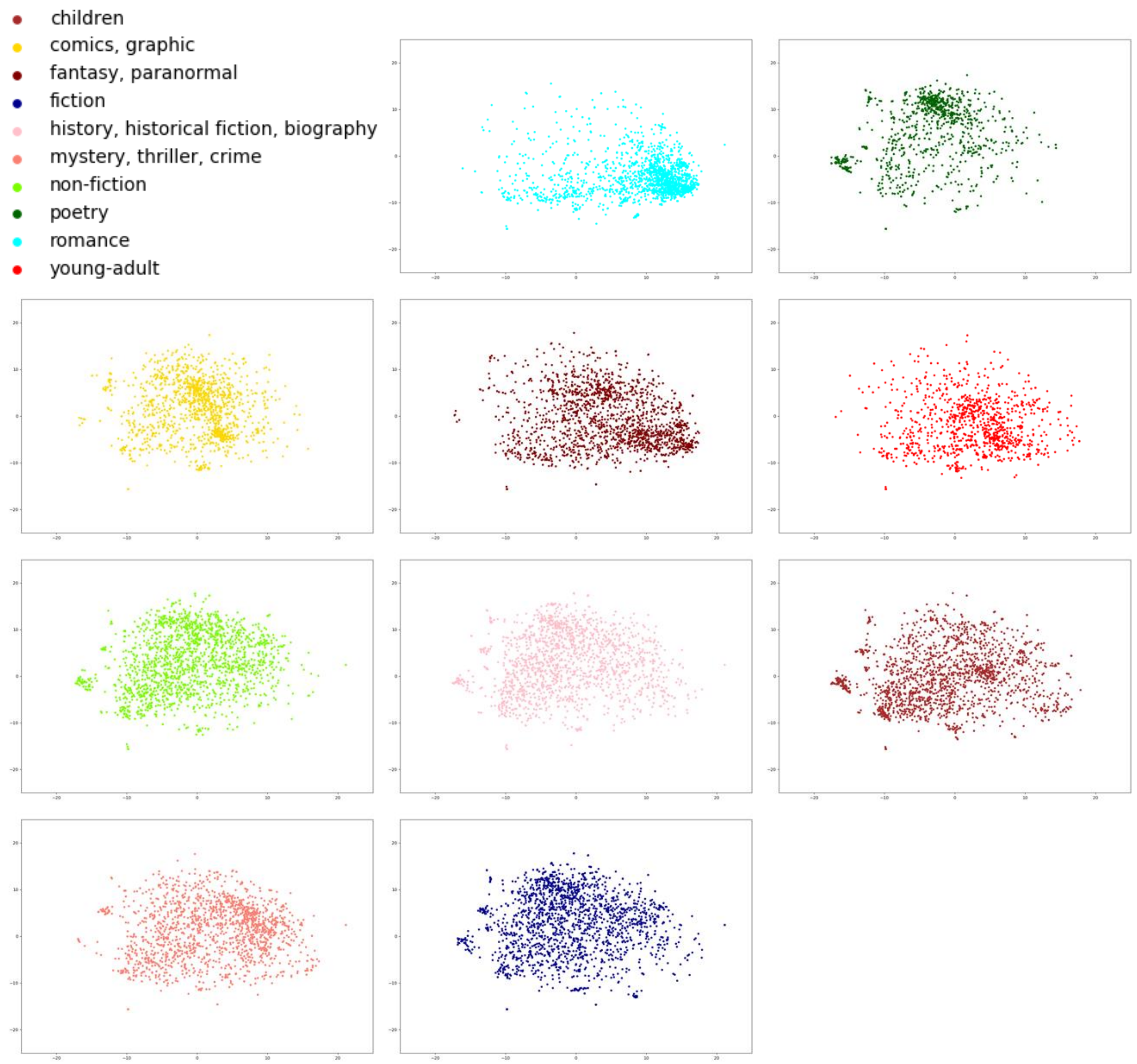
Appendix

1. Additional metrics from grid search on ALS model trained on 100% split





2. Additional out-of-sample test metrics: RMSE for all scores on test set: 0.970; NDCG at k=500: 0.986; Precision at k=500: 0.127
3. t-SNE results printed by genre (perplexity=50; fit on 5% of latent factors for books from ALS model trained on 100% split with Rank=50, regParam=.05)


children
comics, graphic
fantasy, paranormal
fiction
history, historical fiction, biography
mystery, thriller, crime
non-fiction
poetry
romance
young-adult