Goodreads Book Recommendation System

Aidan Claffey, Kevin Wilson
{adc501,ksw366}@nyu.edu
New York University Center for Data Science
DS-GA 1004: Big Data

May 2020

1 Overview

We developed a recommender system for books using data [1] obtained from the book review website *Goodreads* in 2017 [2] [3]. Our system was built using the alternating least squares (ALS) model in Spark to learn latent factor representations of users and books. We trained our model on a subset of the data and used holdout validation data to tune and evaluate different model hyper-parameters. We considered various metrics for evaluating performance for the top 500 book recommendations for each user including precision at 500, mean average precision, and others, and examined our final model's performance on test data not previously seen. Finally, we implemented an extension to visualize the latent item factors using t-SNE to illustrate how the items are distributed in the learned space.

The dataset [1] included approximately 223 million individual book ratings, across 2.4 million unique books and 876,000 unique users. Given the size of the dataset it was necessary to employ "Dumbo," the university's 44-node high performance computing Hadoop cluster.

2 Basic Recommender System

2.1 Data Processing

2.1.1 Data Preparation

To be sure that our data preprocessing, model training, and evaluation would work on the larger dataset, we tested our ALS program on the smaller Goodreads Poetry Dataset [4]. This allowed for quick prototyping during the early stages of the project.

2.1.2 Data Preprocessing

Before splitting the given interaction data into train, test, and validation sets, we filtered it to ensure that our recommendations engine would provide better recommendations, even at the expense of some users and books with few interactions. We first filter out such interactions where `rating = 0`, since this indicates that the user did not rate the book, even if the user had read it. We made this decision because we do not want to infer how a user feels about a book that they did not rate. Of the 223 million total interactions in the data set, this first filter retained approximately 104 million interactions.

Next, we removed interactions that were for books with fewer than 3 reviews. We filtered out books with low review counts because we assume that books with only 1 or 2 reviews will most likely either only show up in the training or only in the validation/test set, and if that is the case, they will not provide any meaningful information to the recommender. Interestingly, this filter removes around 36% of all books, but only 1.1% of all reviews. Therefore, we are still left with around 103 million reviews.

Lastly, we filter for users with 10 or more book reviews. This filter ensures that all users have enough interactions to provide meaningful recommendations without using any cold-start method. In total, this left us with around 102 million reviews.

2.1.3 Data Splitting and Subsampling

While we hoped to use the full dataset to train and evaluate our recommendations engine, we chose to subsample the full set of unique users with proportions of 1%, 5%, 10%, and 50% using the `sample()` function to more quickly tune and evaluate our ALS Model.

We constructed train, validation, and test splits of the data, first using a 60%/20%/20% train/validation/test split on *users*, and for each validation and test user, respectively, including 50% of that user's interactions in the training set. We implemented this split in PySpark using the `randomSplit([0.6,0.2,0.2])` function on a dataframe of unique users, where the split proportions correspond to ['train', 'validate', 'test']. Then, for each of validation and test, we split each user's interactions in half using the `sampleBy()`, and merged one of the halves into the training set using the `union()` function. The code itself can be found in the project repository in the `dataLoader.py` file.

2.2 Model and Experiments

2.2.1 Model Evaluation Metric

While there are several evaluation metrics for ranking systems included in the Spark documentation [5], including Precision at k, Normalized Discounted Cumulative Gain (NDCG), and Root Mean Squared Error (RMSE), we chose to use Mean Average Precision (MAP) as our metric. We chose MAP because it takes into account the order of the rankings and is easily interpretable. Map is calculated with the following equation:

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

where M is the number of users, N_i is the number of ground truth labels for a user i (in our case, the number of ground truth labels in the validation/test set for user i), Q_i is the number of recommended documents, (in our case 500), $R_i(j)$ is the j^{th} recommendation for user i sorted by relevance, D_i is the set of ground truth labels, and $rel_D(x)$ is a function that returns 1 if item x is in set D and 0 otherwise.

In this evaluation, we removed ground-truth ratings that were below 3, since we figure that if a user rated a book below 3, they wouldn't want it being recommended to them (we kept ratings below 3 in the training, just removed them for the MAP evaluation).

2.2.2 Model and Hyper-parameter Tuning

We trained our alternating least squares model using the `pyspark.ml.recommendation.ALS` module in PySpark. We performed hyper-parameter tuning using the training validation data described above. We selected the below hyper-parameters and values for tuning:

- rank: [25, 50, 75, 100]
- regParam: [0.001, 0.005, 0.008, 0.01, 0.05]

These parameters and values were selected in part based on failed attempts at implementing the model. For example, values of rank higher than 100 were attempted but such attempts were found to require much longer computation time, at very little benefit to precision at 500 or MAP.

Our model training and hyper-parameter tuning were conducted progressively. Once a working model was trained and successfully evaluated on a 1%-of-users subsample of the full dataset, a similar subsample of 5% of users was constructed, and likewise a subsample of 10% of users. As the size of our training data

increased, we realized that models trained using `rank=100` performed unilaterally better than lower-ranked models, and as such lower ranks were abandoned for cross validation. Models for 25% and 50% of the data were built but due to cluster memory and congestion constraints we were not able to evaluate the performance of these models.

Table 1 shows the results of our hyper-parameter tuning on 10% of the training and validation data (a value of 'n/a' indicates we did not train and evaluate the model for that hyper-parameter combination).

		rank			
		25	50	75	100
regParam	0.001	n/a	n/a	n/a	0.008848
	0.005	n/a	n/a	n/a	**0.011143**
	0.008	n/a	n/a	n/a	**0.011084**
	0.01	0.00184	0.004172	0.007593	0.010601
	0.03	n/a	n/a	0.006254	0.007424
	0.05	0.001816	0.004119	0.005880	0.006658
	0.1	0.001174	0.002066	0.002384	n/a

Table 1: Mean Average Precision(MAP) on 10% of users for the given hyper-parameters of the ALS model

Tables 3 and 4 in the appendix show the RMSE and Precision at 500 scores for different hyper-parameters. Additionally, all training and evaluation code is in the modelBuilder.py file in the project repository

2.2.3 Model Evaluation

Using the optimal hyper-paremters of `rank = 100` and `regParam = 0.005`, we trained a model on the training data and evaluated its recommendations for the test set that had previously been held out of all hyper-parameter tuning to prevent data leakage. The results can be found in Table 2.

Metric	*RMSE*	*Precision@500*	*NDCG*	*MAP*
Result	0.299402	0.021250	0.028058	0.011132

Table 2: Test results for 10% of users

We recognize that evaluating a recommendation system is not the easiest, especially when the number of books range in the millions. As a qualitative check, we look at some of the actual results by mapping the `book_id` to the book title from the Goodreads dataset[1]. As an example, we recommended one user 3 different French *Harry Potter* books, when their ground truth book set contained several of the English versions of Harry Potter. Therefore, we know our recommendation engine gave *okay* results, since it recommended the correct series. However, these recommendations would not raise the MAP score since they were technically not in the ground truth. A future implementation of the recommendations engine could use the language of the book as a feature.

2.3 Discussion of Constraints and Limitations

At each stage of the process in building this recommender system, we encountered issues given the size of the dataset (223 million individual book ratings). We were successfully able to manage and resolve many of these issues: we used Parquet at multiple points throughout the process to load, manipulate, and save intermediate information; we using iterative subsampling to correct errors in our training pipeline before running on more computationally intensive data sizes; we tweaked Spark configuration parameters (e.g. raising executor memory to 30g) to enable memory intensive tasks to execute successfully. Unfortunately, however, we were not able to train and evaluate a model using the complete dataset; our complete training, cross validation and evaluation procedure was only completed using data from 10% of users. It is likely that this limitation impacted the model performance, as a less exhaustive history of user-book ratings was used than was available.

3 Extension: Exploration Using Visualization

We implemented a data exploration extension on top of the baseline collaborative filtering model. We extracted the latent item factors from our best learned ALS model, specifically a model trained on 10% of the full dataset (see Discussion of Constraints and Limitations, above), with the following key hyper-parameters: `rank=10, regParam=0.005`.

From the *Goodreads* data site [1] we downloaded a mapping of `book_id` values to "fuzzy book genres", and manipulated that mapping such that each `book_id` mapped to the book genre with the highest "fuzzy" genre value. Books with no or invalid genre mapping were discarded. This listing was joined with the latent item factors (using appropriate un-indexed `book_id` values) extracted from the model as described above. A random sample of 100,000 books was selected to make ultimate visualization more feasible. Using `sklearn.decomposition.PCA` we performed principal component analysis on the item factors to initially reduce the dimensions from 100 to 50. We found doing so made the procedure described below much faster to execute.

We next used `sklearn.manifold.TSNE` to implement t-distributed Stochastic Neighbor Embedding, which converts similarities among data to joint probabilities and attempts to minimize divergence between the computed low-dimensional joint probabilities and the actual high-dimensional data. t-SNE offers multiple hyper-parameters, and we attempted tuning with the following values: `perplexity = [30, 40, 50]`; `learning_rate = [100, 200, 500, 1000]`; and `n_iter = [500, 1000]` (maximum number of iterations for the optimization). We plotted the 2-dimensional embedded space and used colors to indicate each book's genre, hoping to find somewhat clear groupings of books by genre. While the books were not perfectly separated in the embedded space by genre, there are very visible groupings particularly for certain genres. For example, most of the books classified as "romance" are included in a single area of the plot separated from the books of most other genres; it makes sense that this genre is more separated from others, particularly considering that some of the other genres have some logical overlap. For example, "fiction", "mystery,thriller,crime", and "fantasy,paranormal" could all describe the same or similar books while "romance" is more distinct. A similar phenomenon appears to be happening with the "comics, graphic" genre.

Our best plot used the following hyper-parameters: `perplexity=50`, `learning_rate = 500`, `n_iter=1000`. This plot is included in Appendix A.3.

We also attempted to use *publication year* to visualize the latent item factors in a lower-dimensional space. We extracted publication year data from the book metadata file and discretized by decade, discarding books with invalid years and limiting to books published after 1920. We again used PCA to perform an initial dimensionality reduction to 50 dimensions, and the same t-distributed Stochastic Neighbor Embedding to visualize this reduced data in 2 dimensions. We once again hoped that books published in the same decade would appear somewhat near each other in the embedded space. While the data are significantly skewed toward later years, there are many clusters within the plot that appear to clump books with close publication years together. This appears, for example, with a large visual grouping of books published during the 1990s. The plot is included in Appendix A.3.

Refer to `tSNE_setup.py` and `Extension_tSNE_Exploration.ipynb` for more details on the implementation of this extension.

4 Contributions of Team members

Both team members worked in parallel (like Spark) on data preprocessing and the initial local implementation of the ALS model. We did this since we knew that there were several approaches to take, so we wanted to see the approaches we would take and then pick which we liked better. We took the same approach when initially moving to the Dumbo cluster to train our models.

After that, Kevin (ksw366) worked on the exploration extension while Aidan (adc501) worked on hyper-parameter tuning on larger subsamples of the data.

Appendix A

A.1 Hyper-parameter tuning for different validation metrics

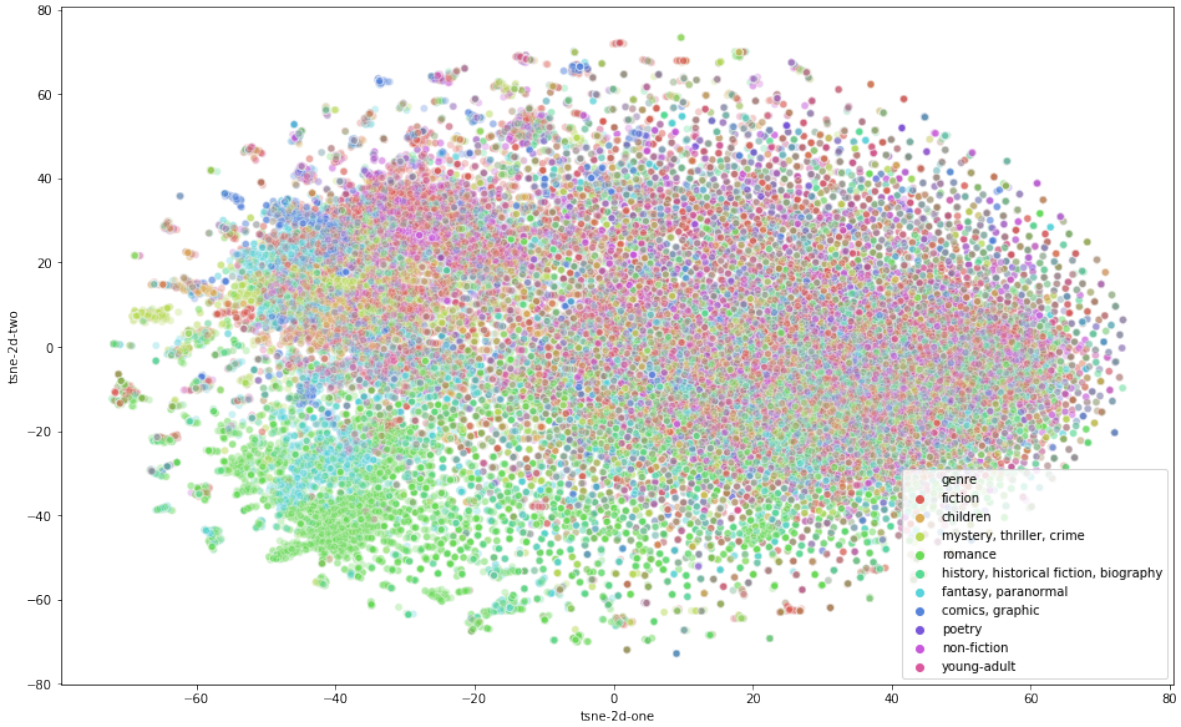
		rank			
		25	50	75	100
regParam	0.001	n/a	n/a	n/a	0.438339
	0.005	n/a	n/a	n/a	0.356027
	0.008	n/a	n/a	n/a	0.340187
	0.01	0.510964	0.423332	0.369227	0.334788
	0.03	n/a	n/a	0.393554	0.366703
	0.05	0.537001	0.480016	0.453154	0.437030
	0.1	0.616482	0.598991	0.592621	n/a

Table 3: RMSE for different hyperparameters on 10% of users

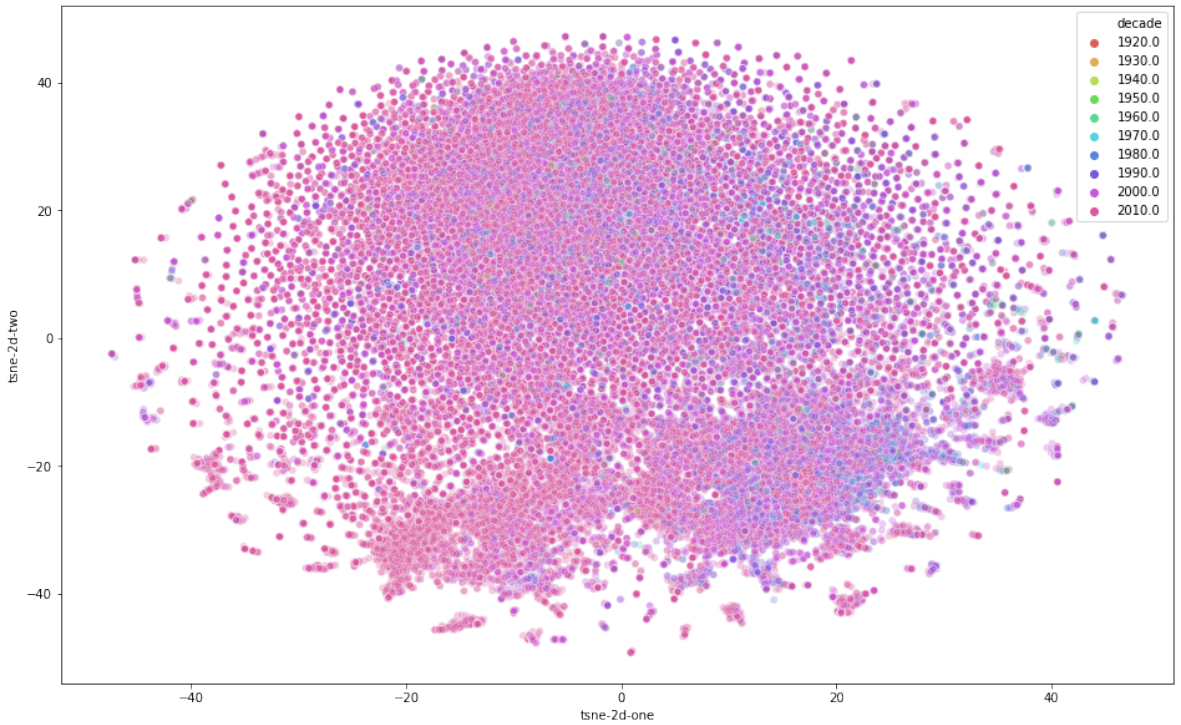
		rank			
		25	50	75	100
regParam	0.001	n/a	n/a	n/a	0.015084
	0.005	n/a	n/a	n/a	0.022082
	0.008	n/a	n/a	n/a	0.023242
	0.01	0.004214	0.010282	0.017538	0.023448
	0.03	n/a	n/a	0.021474	0.023919
	0.05	0.007438	0.016774	0.022274	0.024204
	0.1	0.006150	0.009668	0.010839	n/a

Table 4: Precision @ 500 for different hyperparameters on 10% of users

A.2 t-SNE Embedded Space Visualization, Grouped by Book Genre



A.3 t-SNE Embedded Space Visualization, Grouped by Publication Decade



References

[1] *UCSD Goodreads Book Graph Dataset*. 2017. URL: https://sites.google.com/eng.ucsd.edu/ucsdbookgraph/home.

[2] Mengting Wan and Julian J. McAuley. "Item recommendation on monotonic behavior chains". In: *Proceedings of the 12th ACM Conference on Recommender Systems, RecSys 2018, Vancouver, BC, Canada, October 2-7, 2018*. Ed. by Sole Pera et al. ACM, 2018, pp. 86–94. DOI: 10.1145/3240323.3240369. URL: https://doi.org/10.1145/3240323.3240369.

[3] Mengting Wan et al. "Fine-Grained Spoiler Detection from Large-Scale Review Corpora". In: *Proceedings of the 57th Conference of the Association for Computational Linguistics, ACL 2019, Florence, Italy, July 28- August 2, 2019, Volume 1: Long Papers*. Ed. by Anna Korhonen, David R. Traum, and Lluis Màrquez. Association for Computational Linguistics, 2019, pp. 2605–2610. DOI: 10.18653/v1/p19-1248. URL: https://doi.org/10.18653/v1/p19-1248.

[4] *UCSD Goodreads Book Graph Poetry Dataset*. 2017. URL: https://sites.google.com/eng.ucsd.edu/ucsdbookgraph/home?authuser=0#h.p_kBNa-srdzjOX.

[5] *Evaluation Metrics - RDD-based API - Ranking Systems*. 2020. URL: https://spark.apache.org/docs/latest/mllib-evaluation-metrics.html#ranking-systems.