# Recommendation System Based on User-Book Interactions

**DS-GA1004 Final Project Report**

Anqi Zhang (`az983`) Yuan Ding (`yd1400`) Shuwen Shen (`ss13289`)
Center for Data Science, New York University

May 2020

## 1 Introduction

In this project, we used Apache Spark ML to develop and evaluate a book recommendation system with implicit feedback modeling. Alternating Least Square (ALS), a matrix factorization algorithm, is applied to generate the user and item embedding matrices. The data thathttps://www.overleaf.com/project/5eb70c9ee6574d0001b5a705 we used is from Goodreads dataset[1]. For this project, we used the rating scores as the user-book interactions to train the model and obtain all the evaluation results.

## 2 Implementation

### 2.1 Data splitting and subsampling

We relied on Pyspark on Dumbo to perform data processing. The original dataset contains 228,648,342 rows from 876,145 distinct users, each of which represents an interaction between a user and one specific book. Due to the limited working space on Dumbo, we decided to build a subsample from the original dataset by randomly selecting 1% unique users among all and take all of their interactions to make a miniature version of the data. The subsample contains 8,761 distinct users and 2,394,565 interactions, which accounts for 1.05% interactions of the whole dataset.

In terms of data splitting, users with fewer than 10 interactions were removed, as we believe that they may not provide sufficient data for evaluation. Among the remaining users in our data, after shuffling we selected 60% of users to form the training set, with all of their interactions. We then selected another 20% of users to form the validation set. For each user in the validation set, we randomly selected and put half of their interactions back to the training set, and the remainder is used as the validation set. We used the same process for the test set as for the validation set. In this way, we could make sure that our model does not predict items for users without history.

We developed a python script to accomplish the above process and ran it on Dumbo to derive the desired train, validation and test sets, so that they could be saved and accessed on HDFS for further usage. Plus, we also removed any items in the validation/test set that were not observed in the training set as we did not implement cold-start recommendation.

### 2.2 Model and Evaluation

Our recommendation model uses the alternating least squares (ALS) method from Spark. The latent factor representations for users and items were learned. In our recommendation system, we deal with a set of $M$ users $U = \{u_0, u_1, \ldots, u_{M-1}\}$. Each user ($u_i$) may have a set of $N_i$ ground truth relevant documents $D_i = \{d_0, d_1, \ldots, d_{N_i-1}\}$. We also have a list of $Q_i$ recommended documents $R_i = \{r_0, r_1, \ldots, r_{Q_i-1}\}$ in order of decreasing relevance.

The model evaluation is based on the top 500 predicted items generated by our system for each user after training process. Two additional evaluation metrics were selected to evaluate our model implementation [3].

**Precision at $k$** This metric measures of how many of the top-$k$ recommended documents are in the set of true relevant documents averaged across all users. In this metric, the order of the recommendations is not taken into account. The mathematical definition is as follows:

$$p(k) = \frac{1}{M} \sum_{i=0}^{M-1} \frac{1}{k} \sum_{j=0}^{\min(Q_i,k)-1} rel_{D_i}(R_i(j)) \quad \text{where} \quad rel_D(r) = \begin{cases} 1 & \text{if r} \in \text{D} \\ 0 & \text{otherwise} \end{cases}$$

**Mean Average Precision (MAP)** This metric is also a measure of how many of the recommended documents are in the set of true relevant documents. However, in this case, the order of the recommendations is taken into account, and this metric will impose higher penalty for highly relevant documents not being recommended. The mathematical definition is the following:

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

## 3 Results

**Hyperparameter Tuning** The hyperparameters that we tuned for ALS model were shown below. We tried $3 \times 3 \times 3 = 27$ combinations in total, and plots were made showing the tuning process (see Figure 1 and 2).

- `rank`: [5, 10, 20] - the number of latent factors
- `regParam`: [0.01, 0.1, 1] - the regularization parameter
- `alpha`: [1, 5, 10] - the parameter for implicit preference

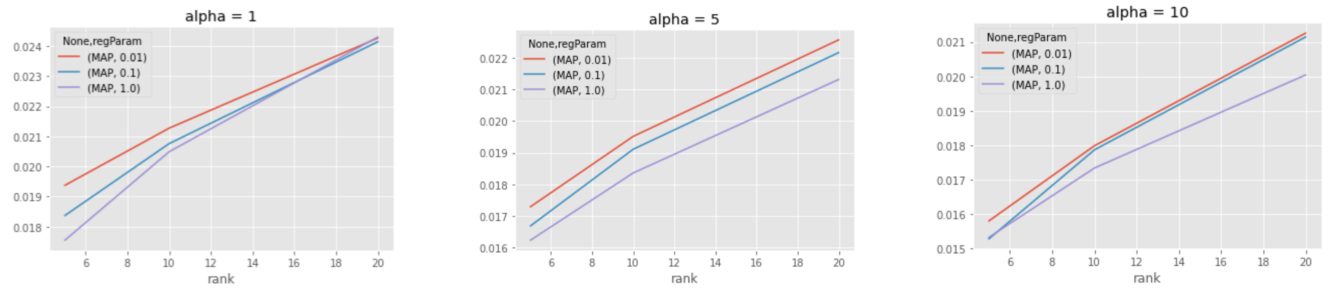


Figure 1: MAP

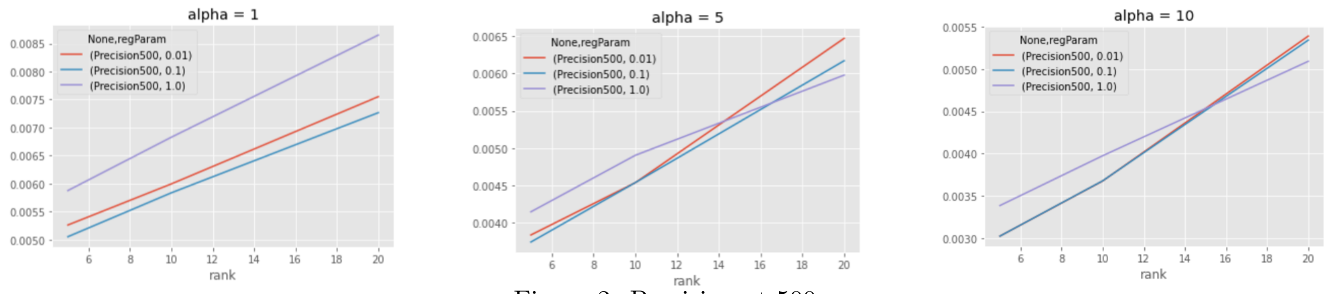


Figure 2: Precision-at-500

**Best Model Performance** After hyperparameter tuning, we were able to find the parameter combination of the best-performing model: `rank`=20, `regParam`=1, `alpha`=1. The results of the best model are shown below.

| | **MAP** | **Precision at $k$ = 500** |
|---|---|---|
| `Validation Set` | 0.0243 | 0.0087 |
| `Test Set` | 0.0252 | 0.0078 |

## 4 Extensions

### 4.1 Extension 1: Fast Search

Spatial data structure could be used for fast search to implement accelerated search at query time. In this project, we imported Annoy package to speed up our recommendation system. Annoy is an approximate nearest neighbours library. Random projections and spatial tree were utilized by Annoy to choose a random hyper-plane at every intermediate node in the tree. We also included the brute-force method for evaluating the efficiency gained by our spatial data structure.

To start with, we generated the latent factors of users and items using our best model trained on Dumbo and exported the result matrix saved as pickle files to our local machine. An Annoy tree was built and indexed by our item vector. Each internal node of the constructed binary tree represents a splitting hyperplane, which is chosen by sampling two points in the subset and taking the hyperplane equidistant from them[2].

The list of top 500 recommended books for each user was generated by brute-force algorithm by computing the dot product between the user and book items. These recommendation results are viewed as ground-truth values and serving as the benchmark to derive the recall scores when evaluating the performance of the Annoy package. We built forests of Annoy trees for different number of trees `n_trees` = [1, 5, 10, 30]. For each "forest" with fixed number of trees, we generated each point in the plot at different number of nodes to inspect during searching `search_k` and their corresponding recall scores.

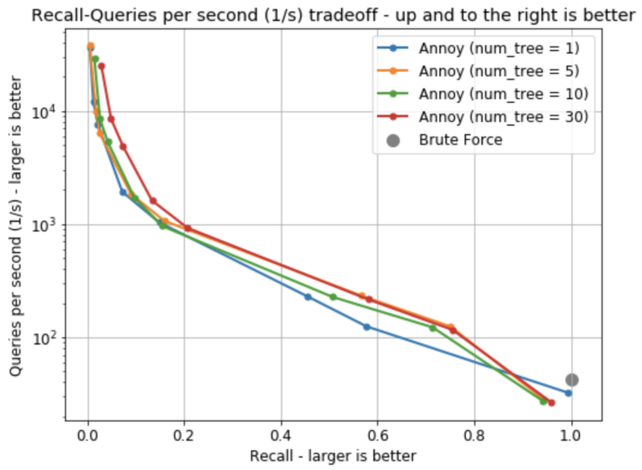


Figure 3: Fast search results

**Tradeoffs** There are two main parameters needed to tune Annoy: the number of trees `n_trees` and the number of nodes to inspect during searching `search_k`. From the above plot, we also observed several tradeoffs between parameters:

- Each line in the plot could be used to visualize tradeoff between the recall and queries speed per second. We noticed that when the parameter `n_trees`= 30, the model achieves the best accuracy among others. As a result, a larger value of `n_trees` will give more accurate results, but larger indexes.
- For each line with fitted number of trees, we could easily conclude that when the recall score increases, the query speed decreases. As a result, a larger value of `search_k` will produce more accurate results, but will potentially take longer time to return.

**Efficiency Gains** Annoy can definitely implement a faster search than brute-force when recall is smaller than 0.9. More precisely, we are able to achieve a 3 times faster query speed using Annoy then brute-force at recall score of 0.8, which is pretty competent and promising. The reason that Annoy seems not performing very well at recall close to 1 is that Annoy will loop not only through the leaf nodes, but also through the intermediate nodes. Also, if very few similarity queries are made, Annoy will spend more time on initializing the indexer so it will take longer time than the brute force method to retrieve results. Therefore, if we are making many queries and a compromise in recall is allowed, Annoy is much more efficient than brute-force in this case.

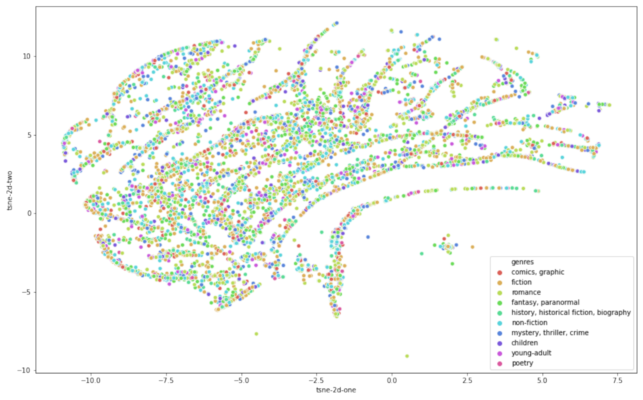
## 4.2 Extension 2: Exploration (Visualization)

The latent factors learned from the ALS model for both items and users (`itemFactors`, `userFactors`) are high-dimensional data and can be visualized using t-SNE. t-Distributed Stochastic Neighbor Embedding (t-SNE) is a technique for dimensionality reduction that is particularly well suited for the visualization of high-dimensional datasets. t-SNE minimizes the Kullback–Leibler divergence between two distributions: a distribution that measures pairwise similarities of the input objects and a distribution that measures pairwise similarities of the corresponding low-dimensional points in the embedding [4]. Contrary to PCA, it is not a mathematical technique but a probablistic one. We used the Scikit-Learn implementation of the t-SNE algorithm for visualization [5].

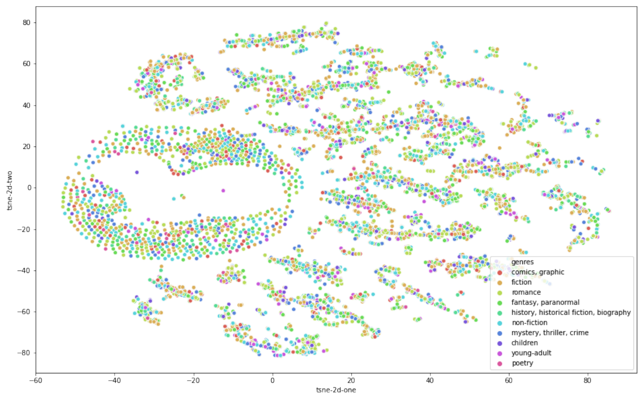
We extracted the user factors and item factors learned by our best model, and since we could not find additional information on users, we only visualized the items. After applying t-SNE algorithm to the item factors dataset, we created a scatter plot of the two resulting dimensions and colored each sample by its respective genres tag. The genres tags for each book are obtained from `goodread_book_genres_initial` [1]. It is a very fuzzy version of book genres and the tags are extracted from users' popular shelves by a simple keyword matching process, therefore a certain book can have multiple genres. For convenience, we tagged each book using only the most frequently occurring genre in all the records. The distribution of all ten genres are: fiction (20.98%), romance (17.05%), non-fiction (14.55%), fantasy, paranormal (13.68%), mystery, thriller, crime (9.29%), history, historical fiction, biography (7.05%), comics, graphic (5.28%), children (4.29%), young-adult (3.58%) and poetry (1.45%).

The results are shown below. First we randomly selected 5% of items (21,407) to visualize (Figure 4(a), Figure 4(b)), and we found although t-SNE can make items into clusters after 1000 iterations, there is no clear distinction between different genres. Then we selected items in five genres which we think are not highly-correlated (fiction, non-fiction, comics, graphic, children and poetry) and trained again (Figure 4(c), Figure 4(d)).We observed that one single cluster still contains books of different genres. The conclusions are:

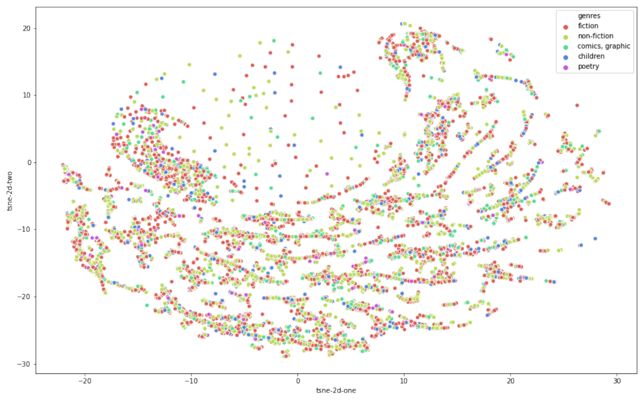
- The items tend to form lines rather than clusters when the number of iterations in t-SNE is not large enough, more iterations can give better visualization results (300 vs 1000).
- The learned item factors implied very little on the genres of books, or the recommendation system does not simply rely on genres to recommend books. This makes sense because people usually like or hate a book based on its content/author/popularity instead of the category. The genres is just an objective feature without indicating more on the quality of a book. Also, in reality a book can definitely have multiple categories (e.g. ”Harry Potter” can belongs to fiction, fantasy and also children genres), therefore the overlapping among different genres clusters is reasonable.



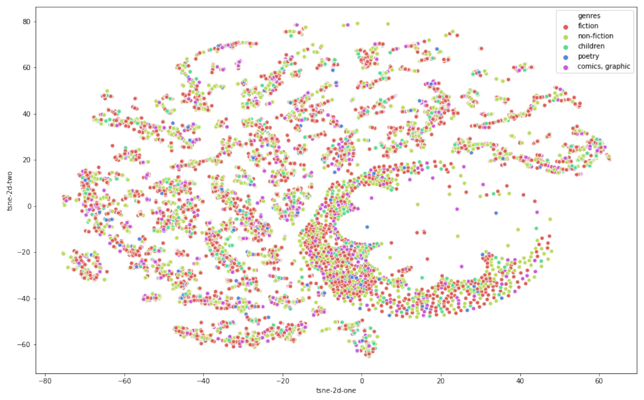
(a) All genres after 300 iterations



(b) All genres after 1000 iterations



(c) Five selected genres after 300 iterations



(d) Five selected genres after 1000 iterations

Figure 4: The Visualization of items based on learned item factors from ALS and genres tags using T-SNE

## 5 Contributions

Anqi Zhang: Baseline Model, Evaluation, Extension 1, Report
Yuan Ding: Data splitting, Baseline Model, Evaluation, Extension 2, Report
Shuwen Shen: Data splitting and subsampling, Baseline Model, Extension 1, Report

## References

[1] Goodreads dataset, collected by Mengting Wan, Julian McAuley, ”Item Recommendation on Monotonic Behavior Chains”, RecSys 2018, `https://sites.google.com/eng.ucsd.edu/ucsdbookgraph/home`

[2] The principle of how Annoy works follows the GitHub homepage of Annoy package: `https://github.com/spotify/annoy`

[3] The definition of the precision at k and MAP follows the documentation of Apache Spark of ranking systems, `https://spark.apache.org/docs/latest/mllib-evaluation-metrics.html#ranking-systems`

[4] L.J.P. van der Maaten and G.E. Hinton. Visualizing High-Dimensional Data Using t-SNE. Journal of Machine Learning Research 9(Nov): 2579-2605, 2008. `http://lvdmaaten.github.io/tsne/`

[5] The introduction of the Scikit-Learn implementation of t-SNE algorithm: `https://scikit-learn.org/stable/modules/generated/sklearn.manifold.TSNE.html`