Book Recommendation System using PySpark

Diwen Lu, Denglin Jiang, Xiaocheng Li

1 Overview

The goal of this project is to build and evaluate a realistic and large scale recommendation system using the Goodreads dataset in Spark and Hadoop Distributed File System (HDFS). We study using Alternating Least Square algorithm in a model-based collaborative filtering model how hyperparameters rank and regParam affect the quality of a recommendation system measured by RMSE, precision@k, and mean average precision. As two extensions, it visualizes the learned book representations projected onto 2-dimensional space, and it explores the impact of spatial data structure (partition trees) on the query time when searching for nearest neighbors.

2 Data Processing

2.1 Data Statistics & Cleaning

For a data overview, please refer to Appendix A. The data in `goodreads_interactions.csv` contains a lot of informationless interactions. Among $\sim$ 2.3B interactions, only $\sim$ 1.1B were rated by a user who actually read the book. And among these authentic interactions, $\sim$ 1.05B have a valid non-zero rating. As the rating is only considered to be valid if it is in {1,2,3,4,5}, we filtered out all interactions with the rating 0 in data cleaning after reading data from the source directory. Then taking into consideration the number of unique users and number of total valid interactions we would be left with, we only kept users who have more than 10 unique interactions because it is difficult to learn an effective distributed representation of a user from only a few ratings. In the end, we are left with $\sim$ 0.7M users and $\sim$ 1B interactions as our full dataset.

2.2 Data Splitting

In an explicit recommendation system, we cannot evaluate a user or an item if that user or item doesn't appear in the training set (cold start problem), because we have no information about that user and item. To prevent such issues, we sample by user as we must guarantee each unique user that appears in validation and test set must also appear in the training set. We find all unique users in the entire data set and split them into 60%, 20%, 20% ratios and then inner join the three groups of users with the entire data set respectively to include the corresponding interactions. Then in each 20% set we take half of interactions for each user and put those interactions back to 60% set, in such a way we can guarantee we never encounter unseen users in validation and test. The cold items can be avoided by setting `coldStartStrategy='drop'` in ALS initialization.

3 Model and Experiments

3.1 Model Description

Our baseline is a model-based Collaborative Filtering model using Alternating Least Square (ALS) algorithm, which attempts to fit the rating matrix R as the product of two matrices U and V, representing users and items respectively, that minimize the mean square error by iteratively and interchangeably optimizing one matrix, U or V, by setting the other fixed.

$$(R^*, V^*) = \underset{(R,V)}{\operatorname{argmin}} \|R - U^T V\|^2 + \lambda(\|U\|^2 + \|V\|^2)$$

Our hyper-parameters include the rank of U and V (rank), and regularization parameter (regParam) λ. Rank controls the fineness of user and item representation. Higher rank is generally thought to give richer but more complex distributed representation. regParam controls the magnitude of latent factor representation, aiming to prevent over-fitting. We set `maxIter = 10` for all our experiments. For rank, we tune over $[5, 10, 20, 30]$ incrementally, and for regParam λ, we follow exponential growth of base 10 over $[.001, .01, .1, 1]$.

3.2 Evaluation Metrics

We use 3 metrics to do model selection: root mean squared error (RMSE), precision@k (p@k), and mean average precision (mAP). RMSE simply measures the closeness of the predicted rating to the ground truth rating. p@k and mAP are two measures that assess how good is our recommendation, whether a user would consume the items that are recommended by the trained recommendation system. p@k calculates how many of the top k recommended items are in a user's ground truth consumption list, ignoring the order of recommendation and the order of ground-truth items. mAP is a further refinement over p@k in that it considers order of recommendation into account. The item that appears in the ground truth consumption list and early in the recommendation list would be weighted heavier than the item that appears in the ground truth consumption list but later in the recommendation list. So a good mAP score implies a user would be fed items that he loves the most.

3.3 Grid Search

In validation, we take top $k = 500$ recommended items per user and when we compare them with the ground-truth items of that user, we only consider the items with a predicted or actual rating ≥ 3, as we want to only recommend books that we think a reader would rate at least "Okay". Our measure of "Goodness" of book rating is based on the following subjective cutoff.

1	2	3	4	5
Awful	Fairly bad	Okay	Will enjoy	Must read

Table 1: Rating Interpretation

Based on our experiments, a single run over validation using 100% data takes around 11 $\sim$ 13 hours using 10GB memory for spark driver and executor. The bottleneck is at executing `model.recommendForUserSubset` to get top 500 recommended books for each user and constructing the `RankingMetrics` object where we need to combine the recommended books and the ground-truth validation books into an RDD. Thus we make the decision to only use 100% data to do grid search for RMSE and use 25% data for p@k and mAP. The following tables show the grid search validation quality.

$rank \backslash \lambda$	.001	.01	.1	1
5	1.04	.888	.835	1.31
10	1.18	.923	.828	1.31
20	1.35	.956	.824	1.31
30	1.42	.976	.822	1.31

Table 2: RMSE (100% data)

$rank \backslash \lambda$	.001	.01	.1	1
5	.555	.815	.659	.769
10	1.42	2.94	1.89	.630
20	6.96	13.6	6.18	.821
30	12.8	21.2	9.37	.792

Table 3: p@k $\times 10^{-5}$ (25% data)

$rank \backslash \lambda$	.001	.01	.1	1
5	.189	.0438	.124	.096
10	.445	.263	.225	.068
20	1.09	2.87	1.04	.119
30	5.19	7.09	2.24	.168

Table 4: mAP $\times 10^{-5}$ (25% data)

Our grid search shows that under different metrics, we get different optimal hyper-parameter settings. Among 3 metrics, we consider RMSE to be the most reliable one because ALS tries to find the representations of users and items such that their dot product can match the rating matrix as close as possible. p@k and mAP all suffer from the fact that different users have different number of books in the validation set, ranging from a few to a few thousand, and both metrics can't capture such difference. But still, they can reflect the general trend of recommendation quality change over grid search. The optimal regParam λ we found lie around 0.1, giving us lowest RMSE and highest p@k and mAP. There is no determined value for rank in our search, because a higher rank tends to give us a higher quality given the optimal regParam. We stopped searching at rank = 30 as there is a trade-off between rank and train and evaluation time performance.

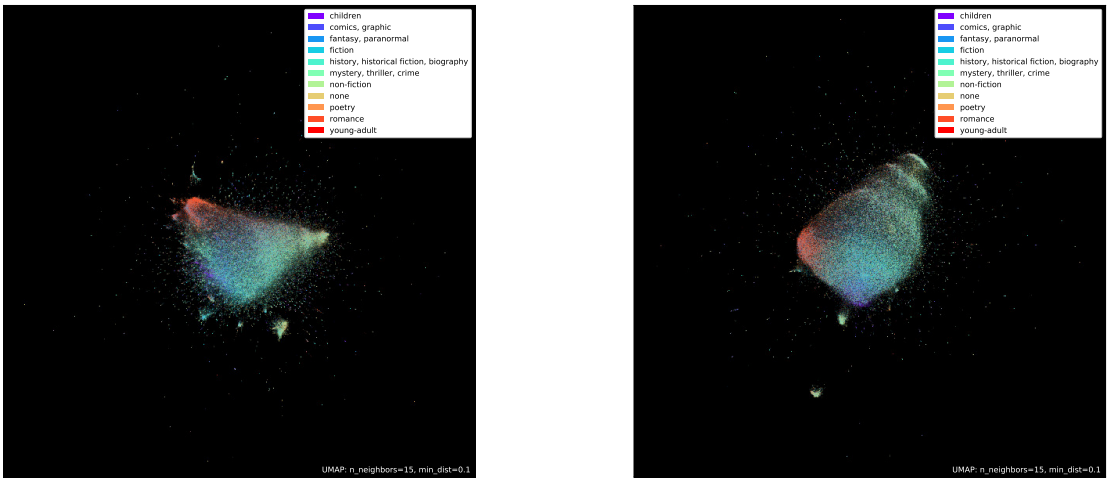
3.4 Test Result

We decide to use rank = 30 and regParam = .01 as our optimal hyper-parameter. Though regParam = .1 is the optimal in terms of RMSE, p@k and mAP are more meaningful metrics in this scenario. The test result from 25% sub-sampled data is RMSE = 1.09, p@k = 19.6×10^{-5}, mAP = 5.76×10^{-5}.

4 Extensions

4.1 UMAP Visualization

We choose to use Uniform Manifold Approximation and Projection (UMAP) to visualize how item representations are distributed in the learned space by genres. We get genres information for each book from `goodreads_book_genres_initial.json` available for Goodreads website. Since a book corresponds to many genres, we take the genre with highest number of instances to be that book's genre label. There are $\sim$ 2.3M books in the full dataset. But due to the limited memory on personal computer, UMAP can only fit $\sim$ .4M 30-dimensional book representations or $\sim$ .4M 10-dimensional book representations in a reasonable time. From Figure 1, clustering doesn't seem to benefit from a finer representation. rank = 10 is sufficient to capture the clustering behavior of books representations. In both plots we can observe that different genres all cluster together but are inseparable. Such pattern is expected as one book can often belong to many genres. But we can see there are still observable patterns within this huge clustering. We can vaguely see the boundaries among reddish, greenish, bluish, and purplish blocks. There are even several homogeneous-colored smaller clusterings that are clearly separated from the biggest one. Figure 1 is a solid proof that our ALS model learns books of different genres in an effective way.



(a) rank = 30, regParam = .01, size = .4M

(b) rank = 10, regParam = .01, size = .4M

Figure 1: UMAP visualization

4.2 Approximate Nearest Neighbor Search: Annoy

Spark ALS `recommendForUserSubset` API uses exhaustive methods for nearest neighbor search to find top k recommendation items for each user. However, for lots of applications, this is way too slow. In a full data setting with nearly 290 million points, we need something faster. Annoy is a faster way to do nearest neighbor search with a unique feature of MMAP, where we can map a user's latent factor very quickly into RAM, realizing it on the page cache in linux or in any kernel.

The implementation of Annoy requires two steps: training an Annoy index and searching. In training, through `AnnoyIndex(rank, metric)` we build an Annoy index using binary trees. We recursively split data sets into two random halves until only k items are left in each leaf. Note that this takes only $\frac{n}{k}$ memory instead of n. Our goal is to have two points preserve their distance in this binary tree as in the original space. Hopefully, if two points are similar to each other in the original space, they are likely to end up in the same branch of the binary tree (1). In the evaluation stage, we search through this Annoy index tree by starting with the root for the query point. Then, through `get_nns_by_item` we navigate down the tree until we reach the leaf. The k points that are closest to the query point would be the Approximate Nearest Neighbors, also the k items to recommend. Additional tricks of priority queue and building a forest of trees are used to improve time performance in the implementation by Spotify (2).

We applied Spotify's implementation (see details of Annoy installation on Dumbo in Appendix B) on our 25% sub-sampled data with different selections of hyper-parameters. To make results comparable, we adopt the same distance measure, dot product, as is default in Spark ALS exhaustive search, and find top 500 approximate nearest neighbors.

$rank \backslash \lambda$	.001	.01	.1	1
5	04:15:20	04:39:32	03:54:47	04:00:42
10	04:05:50	04:14:17	04:41:42	04:25:54
20	05:33:15	03:17:45	05:49:27	04:20:57
30	03:13:53	03:26:14	04:21:26	04:04:15

Table 5: Brute-force validation time (25% data)

$rank \backslash \lambda$	.001	.01	.1	1
5	00:21:01	00:36:45	00:14:41	00:17:10
10	00:19:43	00:18:03	00:14:16	00:11:19
20	00:22:04	00:20:21	00:18:00	00:14:10
30	00:16:54	00:19:19	00:22:48	00:20:43

Table 6: Annoy validation time (25% data)

$rank \backslash \lambda$	.001	.01	.1	1
5	1.06	1.80	1.92	.390
10	2.39	4.98	5.88	.389
20	7.70	14.6	12.1	.529
30	11.9	23.0	32.7	.047

Table 7: Annoy validation p@k $\times 10^{-5}$ (25% data)

$rank \backslash \lambda$	.001	.01	.1	1
5	.065	.157	.129	.060
10	.510	.469	.455	.029
20	.843	.210	1.55	.124
30	3.66	6.23	5.03	.075

Table 8: Annoy validation mAP $\times 10^{-5}$ (25% data)

Theoretically, by searching for approximate nearest neighbors, Annoy trades accuracy (quality) for performance (time), reducing time complexity to $\mathcal{O}(logn)$. In terms of performance, Annoy is able to accelerate searching by about 12 to 15 times empirically on average by comparing Table 5 and Table 6. We imagine the performance would improve much more as rank increases. As for accuracy (quality), we don't observe too much loss. On the contrary, we are surprised to get a bit higher p@k and mAP scores under small rank settings. An explicable reason behind this might be the poor representative quality of our latent vectors resulting from the small rank values, which might impair expression of distance measures. However, as the value of rank increases, these metrics draw near those of brute force search. When rank = 30, we are able to get similar quality metrics from Annoy search to Exhaustive search (Table 7 $(30, .01)$ to Table 3 $(30, .01)$) and we even get way better p@k at $(30, .1)$ in Table 7. The reason we could achieve better performance as well as similar or even better accuracy might lie in our parameter settings. In 25% sub-sampled data, we have 34588 distinct users, and by setting `build(n_trees =30)`, `search_k = 500`, we end up looking at 15000 users in total, covering at least $\frac{1}{3}$ of total users. These parameter values could help us maintain accuracy while improving searching time enormously.

References

[1] E. Bernhardsson, "Personal blog on nearest neighbors and vector models – part 2 – algorithms and data structures," 2015.

[2] E. Bernhardsson, "Annoy." `https://github.com/spotify/annoy`.

Appendix A Data Description

The data (`goodreads_interactions.csv`, `user_id_map.csv`, `book_id_map.csv`) used in the baseline recommendation system has been uploaded on Dumbo by instructor. We only use `goodreads_interactions.csv` for modeling in which users' ids and books' ids have been translated to consecutive integers, `user_id` and `book_id`, for the ease of understanding and computation. The column `user_id` and `book_id` in the file `goodreads_interactions.csv` respectively correspond to the column `user_id_csv` in the file `user_id_map.csv` and the column `book_id_csv` in the file `book_id_map.csv`, where the true desensitized ids are stored.

user_id	book_id	is_read	rating	is_reviewed
0	948	1	5	0
0	947	1	5	1
...	...	...	...	...

Table 9: `goodreads_interactions.csv`

user_id_csv	user_id
0	8842281e1d13...
1	72fb0d0087d2...
...	...

Table 10: `user_id_map.csv`

book_id_csv	book_id
0	34684622
1	34536488
...	...

Table 11: `book_id_map.csv`

Appendix B Annoy Installation on Dumbo

Three steps are required to successfully install Annoy on Dumbo.

1. create and activate a virtual environment venv via conda
   ```
   $module load anaconda3
   $conda create -n venv python=3.6
   $source activate ~/.conda/envs/venv
   ```
2. install Annoy using conda on venv and additional required libraries for this Python in venv
   ```
   $conda install -c conda-forge python-annoy
   ```
3. change pyspark PYTHON PATH to this Python3.6 in venv
   ```
   $export PYSPARK_PYTHON=~/.conda/envs/venv/bin/python
   $export PYSPARK_DRIVER_PYTHON=~/.conda/envs/venv/bin/python
   ```

Then we are done!