DS-GA 1004 Big Data Final Project – Goodreads Recommender System

Devansh Singh, ds6137@nyu.edu

1. Overview

Goodreads is a social networking website that allows users to share information about books that they are reading. A Goodreads user maintains bookshelves of books that he has read or is currently reading. He can rate books that he has already read and can also write reviews for books.

In this final project, the Goodreads dataset has been used to implement an automatic recommender system that uses prior knowledge of a user's rating for some books to suggest new titles to add to his collection. The recommender system uses the Alternating Least Square (ALS) method at its core, which is essentially a matrix factorization algorithm.

We then evaluate the performance of this recommender system based on various kind of metrics like Mean Average Precision, Normalized Discounted Cumulative Gain etc.

For improving the time, it takes our basic recommendation system to generate results, we then use a spatial data structure using the Spotify's annoy python library to accelerate search at query time. A basic summary of the Goodreads dataset:

Total number of unique users: 876,145
Total number of unique books: 2,360,650
Total number of interactions: 228,648,342
Avg. number of interactions per user: 260.98

2. Data Processing

2.1 Data Filtering and Subsampling

For the purpose of building and testing the recommendation system, we consider only those users and their corresponding interactions, **that have at least 10 interactions present in the complete dataset.** This has been done to make sure that each user in the validation and test set have at least some prior history while the model is being trained. With users having less than 10 interactions, it gets difficult to prepare the data for training, test etc. Once all these such users have been identified, we consider interactions of such users only and avoid all other interactions.

Number of unique users that have at least 10 interactions: 772,730
Total number of interactions of such users: 228,248,124

Since, the complete dataset is too big for prototyping the model, we initially used 1% of all the filtered users and their corresponding interactions. Then in the next stage, we used 5% of all the filtered users and their corresponding interactions for hyper parameter tuning, generating the best possible model and using this same baseline model for comparison with the annoy extension. Throughout the project, **we have been able to fit and use only 5% of the total filtered interactions** due to lack of available shared cluster resources.
For subsampling, **we randomly selected 5% of filtered users (38,575) first and then collected all of their corresponding interactions.**

2.2 Data Splitting (Preparation of training, validation and test datasets)

After filtering and subsampling the users, we split them into train, validation and test users in a 60:20:20 ratio. This gives us 3 separate mutually exclusive lists consisting of users for training, validation and testing sets. Using the user_id's in these 3 lists, we then find their corresponding interactions. For every user in the validation and testing datasets, we take half of their interactions and append them to the training interactions dataset and the remaining half are kept in the validation and training sets itself for evaluation and tuning purposes. This makes sure that all the users have some prior history while the model is being trained. This is achieved by first sorting the validation and test datasets based on user_id's and then assigning index to each of the row of these datasets. Then every even numbered row is added to the training dataset and every odd numbered row is kept in the validation or training dataset itself. For 5% subsampling and after creation of final train, validation and test datasets following are the results of our splits:

Table 1: Count of distinct users and total number of interactions for train, validation and test datasets.

	Training Dataset	**Validation Dataset**	**Test Dataset**
No. of unique Users	38575	7723	7786
Total no. of Interactions	9158842	1138381	1151608

It's important here to note that training dataset consists of all the user_id's from the 5% subsample and also half of the interactions for each user from both test and validation sets. The script data_subsample_split.py is used for subsampling and creating the final train, validation and test datasets. The dataset are stored in parquet format as 'train_data_final.parquet', 'val_data_final.parquet' and 'test_data_final.parquet'.

3. Model, Experiments and Evaluations

We create our recommendation model using the ALS method in Spark's to learn latent factor representations for users and items. Alternating Least Square (ALS) is a matrix factorization algorithm and it runs itself in a parallel fashion. ALS is implemented in Apache Spark ML and built for large-scale collaborative filtering problems. ALS attempts to estimate the ratings matrix R as the product of two lower-rank matrices, X and Y, i.e. X * Yt = R. Typically these approximations are called 'factor' matrices. The general approach is iterative. During each iteration, one of the factor matrices is held constant, while the other is solved for using least squares. The newly solved factor matrix is then held constant while solving for the other factor matrix.

Just like other machine learning algorithms, ALS has its own set of hyper-parameters. We probably want to tune its hyper-parameters via hold-out validation (on validation dataset) so that we get a good model without overfitting. There are mainly two hyper parameters in Alternating Least Square (ALS):

- **rank**: the number of latent factors in the model (defaults to 10)
- **regParam**: the regularization parameter in ALS (defaults to 1.0)

maxIter for all the experiments was kept constant at its default value (10).

We evaluate our model based **on the top 500 item recommendations** it generates for all of the considered user. The function recommendForAllUsers(numItems), where numItems=500, is used for generating these recommendations for each user, for all users. For tuning of these hyper parameters and for evaluation on test data, we use the following 3 ranking metrics from Spark's RankingMetrics() module:

1. **Mean Average Precision (MAP):** It is a measure of how many of the recommended documents are in the set of true relevant documents, where the recommendations order is taken into account.
2. **Precision at 500:** It is a measure of how many of the first k recommended documents are in the set of true relevant documents averaged across all users. The order of the recommendations is not taken into account.
3. **Normalized Discounted Cumulative Gain at 500 (NDCG):** It is similar to Precision at k (500) but here the order of the recommendations is taken into account.

The range of hyper parameters for tuning are shown below. These were selected based on the default values and the outputs of previously failed/incomplete implementations of the model. In total, combination of 16 values were used for tuning.

- regParam: [0.01, 0.1, 1, 10]
- rank: [10,20,30,100]

The complete results for tuning are shown in Table 2 below. We can see that the best set of hyper parameters corresponds to regParam=0.1 and rank=100. We also explored results for rank over 100 but have excluded them from the final report because they gave marginal improvements in results which is not justified for the increase in computation time. training_testing_hyper.py is used to perform tuning on validation dataset.

Table 2: Results of Hyper-Parameter tuning on validation dataset.

S. No.	regParam	rank	Mean Average Precision (MAP)	Precision at 500	NDCG at 500
1.	0.01	10	0.05688379617379462	0.05699613402061853	0.232294510036262
2.	0.01	20	0.06647694512582333	0.0607951030927835	0.2526906213617303
3.	0.01	30	0.06995276676930012	0.06275000000000004	0.26094971426462016
4.	0.01	100	0.0779068059301285	0.06554381443298968	0.2760692367517368
5.	0.1	10	0.05553411907067125	0.05689948453608248	0.23003849428055154
6.	0.1	20	0.06399612885438496	0.06058247422680413	0.24911467899897724
7.	0.1	30	0.06790688907851657	0.06238659793814437	0.25768587028195894
8.	**0.1**	**100**	**0.07921910393695457**	**0.06609278350515467**	**0.2803379454668494**
9.	1.0	10	0.0435106399924005	0.05047164948453609	0.19787403343577037
10.	1.0	20	0.04622665512676925	0.05277963917525772	0.20671481059227348
11.	1.0	30	0.04893912665116224	0.054699742268041224	0.21569470213943875
12.	1.0	100	0.05545814769187221	0.05773840206185567	0.23471796113988538
13.	10	10	0.00001225554083191524	0.00017396907216494844	0.000315842110354375
14.	10	20	0.00000857642886540526	0.0001520618556701031	0.00027933488065131365
15.	10	30	0.00001365477618632965	0.00017010309278350512	0.00031500190518525637
16.	10	100	0.00001382382550471401	0.00022422680412371136	0.0004416555475140825

From the results, it can also be noted that for ranks below 100 irrespective of the values of other parameters, performance of these other models is substantially low.

We then used these best set of hyper-parameters to evaluate our final model against the test data set. The results on the test data set have been summarized in the table below:

Table 3: Results of evaluation of the tuned model on test dataset.

regParam	rank	Mean Average Precision (MAP)	Precision at 500	NDCG at 500
0.1	100	0.07945678209879109	0.06493799877225294	0.2830140366190343

testing_for_tuned.py is used for evaluation on the test dataset and it uses the best tuned model generated in the previous step for this purpose.

The basic recommendation model could further be improved by training on the whole dataset (and not just the 5% subsample) and to refine the parameters further for finding more precise and optimal set of values.

4. Extension: Fast Search using Spotify's annoy library

For our extension, we implement the Fast Search at query time using a spatial tree structure. We do so by using the Annoy (Approximate Nearest Neighbors Oh Yeah) library. It is a binary tree-based implementation of ANN that reduces the query search time from *O(n)* (brute-force) to *O(log n).* While generating recommendations for all users in our basic system, the biggest hurdle is that it takes a lot of time for generating these recommendations for each user, for all users. The binary-tree implementation of annoy helps reduce the time it takes to generate these recommendations drastically.

Annoy starts with picking two points randomly and then splitting the hyperplane equidistant from those two points. It then keeps splitting the subspace recursively. Based on this splitting, a binary tree starts to take shape. The same recursive procedure is followed until there are at most K items left in each node. We end up with a binary tree that partitions the space. The nice thing is that, points that are close to each other in the space are more likely to be close to each other in the tree. In other words, if two points are close to each other in the space, it's unlikely that any hyperplane will cut them apart. To search for any point in this space, we can traverse the binary tree from the root. Every intermediate node (the small squares in the tree above) defines a hyperplane, so we can figure out what side of the hyperplane we need to go on and that defines if we go down to the left or right child node. Searching for a point can be done in logarithmic time since that is the height of the tree.

In our experiment, we compare the time it takes our basic recommender system with the best set of hyper-parameters and the annoy implementation to generate recommendations for each and every user.
We were able to implement the annoy extension currently on our local system only and not on dumbo cluster due to library installation issues. We tried to generate recommendations for all the users in our test data set, but due to

high computational costs our local machine did not allow us to do so. So, we limited ourselves to 800 unique users from our test dataset and generated top 500 recommendations using both the systems for each of these 800 users and compared the time taken. Table 4 below shows the time it took for our basic recommender system.

Table 4: Time taken and precision by the basic recommender system to generate 500 recommendations for 800 of our users in the test data set.

regParam	rank	Time Taken (sec)	Precision at 500	Mean Average Precision (MAP)
0.1	100	72.46092319488525	0.0664575	0.08108238635824877

There are just two main parameters needed to tune Annoy: the number of trees n_trees and the number of nodes to inspect during searching search_k.

n_trees is provided during build time and affects the build time and the index size. A larger value will give more accurate results, but larger indexes. **search_k** is provided in runtime and affects the search performance. A larger value will give more accurate results but will take longer time to return.

We use the following range and combinations of n_trees and search_k to find the combination that provides the most optimal run-time. The following table 5 summarizes our results:

n_trees : [10, 20, 40, 50] , search_k : [-1, 10, 50, 100]

Table 5: Tuning of n_trees and search_k parameters on 800 of our users in the test data set.

n_trees	search_k	Time Taken (sec)	Precision at 500	Mean Average Precision (MAP) ($x10^{-6}$)
10	-1	3.655941963195801	0.00013750000000000006	2.307559477208712
10	10	3.51990008354187	0.00013750000000000006	2.3075594772087115
10	50	3.253026247024536	0.00013750000000000004	2.307559477208711
10	**100**	**3.246181011199951**	**0.0001375000000000001**	**2.3075594772087115**
20	-1	3.547536849975586	0.00013750000000000004	2.384414371787422
20	10	3.555121898651123	0.00013750000000000004	2.384414371787422
20	50	3.4355199337005615	0.00013750000000000006	2.384414371787422
20	100	3.472327947616577	0.00013750000000000006	2.3844143717874223
40	-1	4.1015541553497314	0.00013000000000000002	2.3049397686497475
40	10	4.116975784301758	0.00013000000000000002	2.304939768649748
40	50	4.629157781600952	0.00013000000000000007	2.304939768649748
40	100	4.5987389087677	0.00013000000000000002	2.304939768649748
50	-1	4.855401039123535	0.00013000000000000002	2.3022370366719375
50	10	4.6285529136657715	0.00013000000000000002	2.302237036671937
50	50	4.546988010406494	0.00013000000000000002	2.3022370366719367
50	100	4.774848937988281	0.00013000000000000002	2.302237036671937

We see that the least time taken by the system is 3.25 seconds at n_trees=10 and search_k=100. This is a reduction of **almost 95%** as compared to the brute force search approach of our basic recommender system. However, the accuracy of the recommendations is significantly lower than the ones achieved by our basic recommender model. Looking at the results in the above table, it can easily be said that there is a tradeoff between better accuracy and speed.

Since, we performed our experiment on a set of only 800 test users it, the results might change when the same setup considers a larger test user set. With increased computational power, we can also test for higher values of search_k (like 5000, 6000 etc.) which will give more accurate results but will take longer time to execute.

extension.py is used to run the experiment comparing the runtime of basic recommender system with the annoy implementation. We also looked at other libraries, like nmslib, but found annoy implementation to be the simplest and most intuitive one.

Appendix

Contribution of team member:

Since, I (Devansh Singh) am completing this project alone, all the python scripts, implementation of basic recommender system, hyper parameter tuning, implementation of annoy extension and writing of this report have been performed solely by me.

Please refer to the “script_execution_steps.pdf” document in the Github repository for a detailed explanation about the purpose and how to execute of each of the scripts present in the repository.

References:

1. Annoy GitHub: https://github.com/spotify/annoy

2. How Annoy works: https://erikbern.com/2015/10/01/nearest-neighbors-and-vector-models-part-2-how-to-search-in-high-dimensional-spaces.html

3. ALS:https://towardsdatascience.com/prototyping-a-recommender-system-step-by-step-part-2-alternating-least-square-als-matrix-4a76c58714a1

4. NDCG:https://towardsdatascience.com/evaluate-your-recommendation-engine-using-ndcg-759a851452d1

5. RankingMetrics: https://spark.apache.org/docs/latest/mllib-evaluation-metrics.html#ranking-systems