DS-GA 1004 Big Data Project: Goodreads Book Recommender System

Anu-Ujin Gerelt-Od
ago265

Paula Kiatkamolwong
kk4158

May 11, 2020

1 Overview

Recommender systems are a driving force for a variety of businesses such as retailers and entertainment services, and their popularity is only rising. On the business side, it allows the companies to offer a wide range of products, without limiting them to a confined physical space. As for the users, it makes it easier for them to make a decision when consuming a product by giving them a personal recommendation based on the choices made by users with similar interests.

This project explores a basic recommendation system implementation, with a comparison to single machine implementations extension. The data set that is used for this project was provided by Goodreads [1] - an online platform for readers to share their opinions on books they've read, and get recommendations for new books based on their preferences. The data consists of user ID, book ID, and the rating for each book, as well as Boolean-type factors such as whether the user read the book and/or left a review. As we have explicit feedback from the users in the form of a rating, we have used a Collaborative Filtering approach, which creates a recommendation based on the behavior history of the user as well as of users that have similar interests. The basis on which we split the data to ensure that this approach works is described in the Data Splitting subsection. Additionally, as the data matrix for this type of problem is usually very sparse, we have used an Alternating Least Squares method to process the data set and make predictions, details of which can be found in the Model and Experiments section. Lastly, we tested out an extension for a single-machine implementation of recommendation algorithms, to compare the time efficiency and their accuracy with our basic recommender system. The majority of the model building was completed on the cluster-computing framework - Apache Spark, supported by the Python API - PySpark [2] and the extension was built with LightFM [3].

2 Data Processing

2.1 Data Cleaning

For the building of the basic recommender system, we used the "goodreads_interactions.csv", extracted from the Goodreads website which contained information about 876,145 users, 2,360,655 books, and 228,648,342 interactions which is defined as books that users read, rated and/or reviewed. In order to make an effective recommender system, we cleaned the data set and removed entries that would not provide useful insights. For this purpose, we utilized the built in Spark SQL functions and created new DataFrames from the filtered data sets. This included the removal of interactions that had a rating of 0, which meant that the user did not leave a numerical rating, even if they had read the book. Later on, when we make predictions for the ratings of each book, the ones that did not have a rating could potentially result in lower and inaccurate predictions. This process brought down our sample size to 104,551,549 data points. Additionally, we grouped the books

that each user had read, and eliminated those that had a count of less than 10 - in other words, the ones that had fewer than 10 interactions. As our evaluation criteria was done on top 500 books, these entries would not have provided any information and it was better to cut down the sample size, in this case by about 110,000. In the end, our entire set consisted of 104,338,752 interactions, which is a little less than half of the original data set.

2.2 Data Splitting

The entire data set was split into three parts to be used for the building of the model. We went with the recommended size of 60% for training, 20% for validation and 20% for testing. Then, for the validation and test sets, we utilized Spark SQL's window function to group the interactions by users, created an index for each entry in the subsets and divided them into train and validation/test sets based on the value of their indices being even or odd. Finally, we wrote out the three data sets to parquet format for efficiency and ease of access. We repeated the whole process to downsample the data sets for testing the model in subsets of 1%, 5%, and 25% of the data as suggested by the project guideline.

3 Model and Experiments

The basic recommender system was built using an alternating least squares (ALS) matrix factorization method provided by PySpark's ML Recommendation module. This model works well when the data is sparse, and it factorizes the rating matrix into a product of two low-dimensional rectangular matrices based on users and items, and their latent factors. By creating these decomposed matrices, the ALS model allows for a better representation of all items in the set, regardless of popularity and results in well-personalized recommendations. The computation is expressed by

$$r_{ui} = \sum_{f=0}^{nfactors} H_{u,f} W_{f,i}$$

where r is the predicted rating for user u on item i based on $f = (0, n)$ latent factors and H and W are the user and item matrices, respectively [4]. With this method, we are able to run our model on different values for rank, which is the number of latent factors, and the L_2 regularization parameter for minimizing the loss function. Although the ALS implementation in Spark allows for the number of iterations to be set, we did not find the need to optimize on this as ALS tends to converge to a local minima on its own.

For hyperparameter tuning, we created a list of potential values for the $rank = [15, 20, 25, 30]$, regularization parameter $\lambda = [0.01, 0.05, 0.1, 0.25, 0.5]$, and joined them as a Python itertools object. Then, we created a grid search loop that would go through each pair of parameters and by using PySpark's ML Evaluation tool, report the root mean square error (RMSE) from the predicted and actual values. The results are reported in Table 1. We stored these in a list and kept a running tab of the lowest score and the corresponding model, saved as "best_rmse" and "best_model", along with the $rank$ and λ values. However, due to memory issues and in an effort not to overuse the cluster's resources, we periodically checked the RMSE values, and tuned the parameters accordingly. Early on for $rank = 15$, we observed that a regularization of $\lambda > 0.1$, in our case and $\lambda = 0.5$, resulted in an almost 116% increase in the error from 0.8484 to 0.9816, and similarly for values of $\lambda < 0.025$ results of which are not reported here. Thus, we narrowed our search to λ =0.05, 0.1 and 0.25. As for the rank, we found that results did not vary much with $\lambda = 0.1$ for $rank$ in range 15 to 30, so we stopped the grid search at $rank = 30$ with $\lambda = 0.1$ and were satisfied with our final values of $rank = 25$ and $\lambda = 0.05$, with a RMSE = 0.8233. Once the tuning was completed on the validation set, we tested out or optimal parameters on the test data set which returned RMSE = 0.8240.

To evaluate the recommender system model, we used the Recommend For User Subset function from PySpark as we did not have enough time to implement the Recommend For All Users function. The former function returns a recommendation for a small subset of users, in our case the test set, which as described before consists of 10% of the total data, and recommends k = 10 books for each user. We then

utilized the Ranking Metrics function from PySpark's MLLib Evaluation module and got a score of: MAP = 0.007479458649235268, Precision at k = 0.003048574214474758, NDCG at k = 0.011139540262473895.

	rank=15	rank=20	rank=25	rank=30
$\lambda = 0.05$	0.84847573845945784	0.8248559575629095	0.8233555832217416	0.8239379763987713
$\lambda = 0.1$	0.8276639588204272	0.8258210277376227	0.8243640405876608	NaN
$\lambda = 0.25$	0.8789013152839666	0.8791821957210181	0.8635750746637746	NaN
$\lambda = 0.5$	0.981633410146653	NaN	NaN	NaN

Table 1: Hyperparameter tuning results

4 Extension: Comparison to Single-Machine Implementations

LightFM is an open source python single-machine implementation of recommendation algorithms that also provides metrics for evaluating the performance of the model. For our project, we implemented a LightFM model with parameter k = 500 and the WARP ranking loss function. The program first reads train and test sets from parquet files, and converts them into Pandas DataFrames. These parquet files have been pre-processed and split to make sure that all users in the test set are contained in the train set as well. In order to train our model, we first have to convert the train and test DataFrames into COO matrices. The model is trained and evaluated on the test set. The table below (Figure 1) reports the comparison between Spark's parallel ALS model (using rank=10 and regParam=0.05) and LightFM model in terms of efficiency in model fitting time and accuracy for precision at k = 500.

Model	ALS Time	LightFM Time
1% of dataset		
Train	56 sec	157 sec
Total time (Train, Test, Evaluate)	729 sec (~12 min)	1591 sec (~27 min)
Precision at K (k=500)	0.00030	0.00013
5% of dataset		
Train	110 sec	1548 sec
Total time (Train, Test, Evaluate)	2071 sec (~35 min)	20,542 sec (~5 hr 42 min)
Precision at K (k=500)	0.00052	0.00016
25% of dataset		
Train	784 sec (~13 min)	Not available
Total time (Train, Test, Evaluate)	21773 sec (~6 hr 3 min)	Not available
Precision at K	0.00187	Not available

Figure 1: Time and accuracy comparison between ALS and LightFM models

Figure 2 shows a plot for time efficiency of the two models as a function of data set size. Training in both model is relatively fast. The bottle necks for both models occur during testing and evaluating. As the size of the data set grows, the total time required for the model to complete grows exponentially. For all sizes of the data set used, ALS model performs remarkably better than LightFM model in terms of both time efficiency and accuracy.

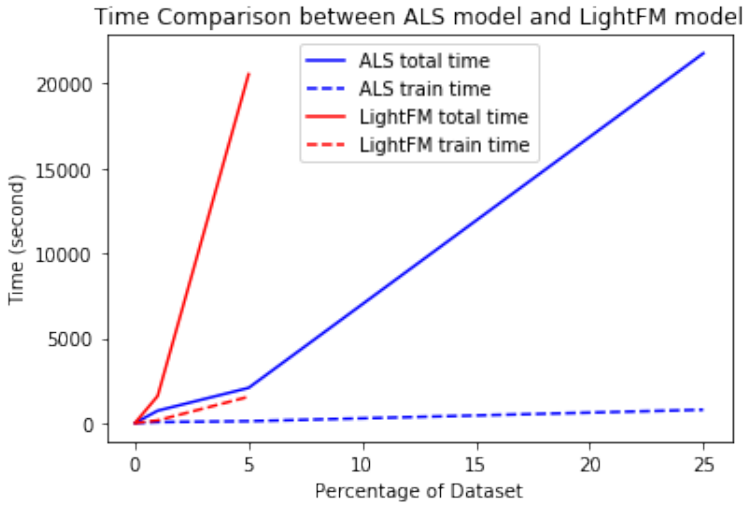

Figure 2: Time comparison between ALS and LightFM models as a function of data size

5 Contribution of Team Members

Both team members contributed to the data processing, modeling and experimenting process of the project. Anu-Ujin Gerelt-Od took lead in the data cleaning and splitting steps, and on parameter tuning. Paula Kiatkamolwong took lead in implementing the extension with LightFM.

References

[1] Goodreads dataset
`Mengting Wan, Julian McAuley, "Item Recommendation on Monotonic Behaviour Chains",` *RecSys'18.*
`Mengting Wan, Rishabh Misra, Ndapa Nakashole, Julian McAuley, "Fine-Grained Spoiler Detection from Large-Scale Review Corpora",` *ACL'19.*

[2] Apache Spark with Python API
`https://spark.apache.org/docs/latest/api/python/index.html`

[3] LightFM
`https://making.lyst.com/lightfm/docs/home.html`

[4] Kevin Liao, "Prototyping a Recommender System Step by Step Part 2" *TowardsDataScience*
`https://towardsdatascience.com/prototyping-a-recommender-system-step-by-step-part-2-alternating-least-square-als-matrix-4a76c58714a1`

[5] Nasir Safdari, "If You Can't Measure It, You Can't Improve It!!! How to Build a Scalable Recommender System for an e-commerce with LightFM in python" *TowardsDataScience*
`https://towardsdatascience.com/if-you-cant-measure-it-you-can-t-improve-it-5c059014faad`

[6] Arun Mathew Kurian, "How to build a Movie Recommender System in Python using LightFm" *TowardsDataScience*
`https://towardsdatascience.com/how-to-build-a-movie-recommender-system-in-python-using-lightfm-8fa49d7cbe3b`