Recommender System

DSGA 1004 Final Project Report

Akash Mishra
New York University
NY, USA
am11533@nyu.edu

Mayank Poddar
New York University
NY, USA
mp6021@nyu.edu

ABSTRACT

Almost all media services have a particular section where the system recommends things to you, such as a movie on Netflix, a product to buy on Amazon, a playlist in Spotify, a page to like on Facebook and so many others. We all have seen the “You might like this” section at least once in almost all services we daily use; the algorithm behind that is a Recommender System.

A Recommender System can be considered a subclass of information filtering system with ability to predict preferences of users. It derives from the dependency between the user and item centric activities.

1. Goal

The goal of this project is to build and evaluate a recommender system on small and large scale dataset from movielens [1]. We implement this system using pyspark’s ALS module. Along with this, we implemented a baseline popularity model for cold start strategy. As for the extension, we built and evaluated a model on LightFM and compared the performance with the model from ALS.

2. Baseline Implementation

Dataset: As instructed in the project guidelines, we have been using Movielens [1] dataset for our recommender system implementation. There are two variants of this dataset. First one contains around 100K data while the large one contains around 27 Mil data. We are explicitly using only ratings.csv which contains four columns - *userId, movieId, ratings and timestamp.*

Data Preprocessing: For this project, we used PySpark to process and build the model. To begin with, we add the dataset to *HDFS* using:

'hadoop fs - puts'

After adding the data to HDFS, we preprocess the data by loading it into a spark instance where we process, partition and sort the data so that further data wrangling can be done efficiently. Here, we have created 10 partitions for the whole dataset. Lastly, we divided the data into a train and test dataset with a ratio of 1:4. We saved the processed dataset into parquet files which are considered much more efficient in parallel computation. This is something we have also previously done in the Lab assignments.

The second robust approach to dataset splitting is as follows. We first take all the unique values of userId and split it into 60%, 20%, 20% for train, test, val respectively. Next, the val dataset and test dataset contain all the ratings data pertaining to their userId. As for train dataset, along with all the ratings data for train userIds, it also contains 40% of data from validation dataset and 40% of dataset from test dataset. This is done to ensure that the train dataset has seen all the users.

Baseline Popularity Model: Before starting with any complicated model, we started with a popularity model which would work well for cold start problems. Here we get a weighted rating for each movie in the dataset. We added a dampening factor ρ while calculating the global ratings so as to make sure to offset the rating if there is a low number of ratings for a given movie.

There are two approaches here:

1. Without User and Item Bias

$$P[i] \leftarrow (\sum_u R[u, i]) \; / \; (|R[:, i]| + \beta)$$

2. With User and Item Biases

- Model each interaction as a combination of **global**, **item**, and **user** terms:

$$R[u, i] \approx \mu + b[i] + b[u]$$

- The average rating over all interactions:

$$\mu = (\sum_{u,i} R[u, i]) / (|R| + \beta_g)$$

- Average (users) difference from μ for item i:

$$b[i] = (\sum_u R[u, i] - \mu) / (|R[:, i]| + \beta_i)$$

- Average (items) difference from $\mu + b[i]$ for user u:

$$b[u] = (\sum_i R[u, i] - \mu - b[i]) / (|R[u]| + \beta_u)$$

ALS: Alternating Least Squares (ALS) matrix factorisation 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.To personalize the recommendation, we used the ALS (Alternating Least Squares) method from PySpark's machine learning library that can run in parallel. We used a cross validation method from the PySpark evaluation library to hypertune the model and get the best param(using KFold CV). We set cold start strategy to 'drop' to ensure we don't get NaN evaluation metrics. The algorithm used is based on "Collaborative Filtering for Implicit Feedback Datasets"[4], adapted for the blocked approach used here.

$$\min_{U,V} \sum_{(i,j)\in\Omega} c_{ij}(p_{ij} - \langle U_i, V_j \rangle)^2 \qquad p_{ij} = \begin{cases} 1 & R_{ij} > 0 \\ 0 & R_{ij} = 0 \end{cases} \qquad c_{ij} = 1 + \alpha R_{ij}$$

Figure 1: Mathematical expression representing latent factor models

3. Extension

Comparison to single-machine implementation (using lightfm)

As an extension to our recommendation system project, we implemented a standalone implementation of the recommender system using lighfm[2]. We ran the model ALS and LightFm on both large and small dataset and evaluated the metrics and time taken which you can see in the code implementation also. As we will see the comparison between the ALS and LightFM in the result section. While we have used multiple metrics, primarily for this comparison we are using precision@K.

https://github.com/nyu-big-data/final-project-group_33

4. Evaluation

Hyper-parameter Tuning: What we do in model tuning is shape out model with the goal of achieving the best possible score in the unseen data i.e. validation data. We have used rmse as scorer function in the tuning for ALS while precision_at_k for lightfm model. We have used a built-in tuning method for PySpark[3] while for lightfm we used RandomizedSearchCV from scikit-learn library.

RMSE(Root Mean Squared Error): It's one of the most commonly used metrics for machine learning tasks.

$$\text{RMSE} = \sqrt{\frac{1}{|\hat{R}|} \sum_{\hat{r}_{ui} \in \hat{R}} (r_{ui} - \hat{r}_{ui})^2}.$$

Mean Average Precision(MAP): Average Precision depicts the fraction of higher-ranked items that are positive for each positive interaction. Score is calculated for each user and averaged over all users to get Mean average precision.

Mean Average Precision At K: MAP@K works similarly to Mean average precision but we only consider the first K interaction for getting the score.

$$\text{AP@N} = \frac{1}{\min(m, N)} \sum_{k=1}^{N} P(k) \cdot rel(k).$$

Precision and Recall at K: Precision and Recall at cutoff K, P(K) and r(K), are simply the precision and recall calculated by considering only the subset of your recommendations from rank 1 through K. It's all about focusing on the rank ordering of relevant items.

recommender system precision: $$P = \frac{\text{\# of our recommendations that are relevant}}{\text{\# of items we recommended}}$$

recommender system recall: $$r = \frac{\text{\# of our recommendations that are relevant}}{\text{\# of all the possible relevant items}}$$

Normalized Discounted Cumulative Gain (NDCG) at K:

Discounted Cumulative Gain involves discounting the rating scores with the log of the corresponding position.

$$DCG = \sum_{i=1}^{n} \frac{2^{relevance_i} - 1}{log_2(i+1)}$$

However, the issue with DCG is that the number of recommendations served may vary for every user. Hence, we need to normalize the metric to account for it. The NDCG is the DCG with a normalization factor in the denominator which is just the DCG of ideal order.

$$NDCG = \frac{DCG}{iDCG}$$

AUC Score: This score represents how often a positive interaction ranks ahead of a negative interaction.

Reciprocal Rank: It is the inverse of the first position where we get the positive interaction.

Even though we have used mostly all available metrics in the PySpark and LightFM library, for comparison we have relied on precisionK as it will give us the quantification of all the correct items in the top K recommendations.

We have evaluated the metrics for K=100

5. Results

We have evaluated the metrics for K=100

5.1 Baseline Popularity

5.1.1 Only Global Bias

data	map@K	precision@K	ndcg@K	recall@K	RMSE	MAE
val	0.1674	0.2657	0.3072	0.2146	2.8901	2.6847
test	0.1332	0.2497	0.3097	0.2513	2.7169	2.5017

5.1.2 Global, item and user bias present

data	map@K	precision@K	ndcg@K	recall@K	RMSE	MAE
val	0.1162	0.2306	0.2797	0.1875	0.9102	0.7173
test	0.1053	0.2125	0.2581	0.2077	0.9265	0.7279

The thing to note here is that even though the RMSE and MAE have reduced a lot, almost by ~67%, the ranking metrics have not significantly improved, even reduced in some cases. This cements the point that adding user and item bias might not necessarily improve the baseline model, but it is much more interpretable because it gives us the ability to fine tune global, user and item biases.

5.2 ALS (Alternating Least Squares):

We have trained the model using CrossValidator and ParamGridBuilder to get the optimal parameters with the scorer function being RMSE. We used KFold Cross validation (K=5) for improved results. Lastly, we have used this set of hyper-parameters to optimize performance.

$$regularizationParams = [.01, .05, .1, .2]$$

$$latent\_ranks = [10, 50, 100, 150]$$

data	map@K	precision@K	ndcg@K	recall@K	RMSE	MAE
small	0.0039	0.0454	0.0413	0.0163	0.9631	0.7525
large	0.0021	0.0338	0.0253	0.0099	0.861	0.6797

Best Param

Small Data :- $Rank: 50 \ RegParam: 0.1$

Large Data :- $Rank: 150 \ RegParam: 0.05$

It took us 850 seconds to train on the small dataset while around 100 minutes to train on the large dataset.

5.3 LightFM

As previously discussed, we have used cross validation and hypertuned the parameters in the LightFM model using RandomizedSearchCV from scikit-learn library. For both the large and small dataset, we got these optimum parameters for fitting the model.

We used a gamma distribution with a=1.2 for learning rate optimization and loss function = ["bpr", "warp", "warp-kos","logistic"].

Best Param for both dataset:

$lr = 0.1841, \ Loss = bpr, \ no\_components = 21$

Main Takeaway: Time taken for a small dataset is around 320 seconds. This is lower compared to ALS. But when we look into the large dataset, we see that ALS using PySpark completes the model training in just 1.5 hours while LightFM takes around 4 Hours to train the whole model. We can attribute to the parallel computation that it becomes much more effective as that dataset size becomes quite large.

Lastly, when we check the accuracies we can see that ALS has better precisionAtK which we are using to score the recommendation ranking for the user in the movielens dataset.

Dataset	evalset	precision @K	auc	recall@ K	reciprocal rank
small	train	0.4974	0.9979	0.0902	0.7111
small	test	0.0587	0.8943	0.0388	0.1609
larget	train	0.3918	0.9617	0.0616	0.82
large	test	0.0311	0.8538	0.0311	0.278

Infrastructure: We have used Peel and dataproc from GCP for all the spark jobs. While for LightFM model training, 16vCPU GCP Compute Engine was used to run the large and small from movielens.

6. Contribution and Collaboration

Akash Mishra: ALS, Extension 1

Mayank Poddar : Baseline Popularity, Extension 1

7. REFERENCES

[1] Dataset from Movielens

[2] LightFM https://github.com/lyst/lightfm

[3] PySpark https://spark.apache.org/docs/latest/api/python

[4] Collaborative Filtering for Implicit Feedback Datasets https://ieeexplore.ieee.org/document/4781121