What will you read next?
Book Recommendation System using Spark and LightFM

Nikhil Supekar (ns4486)
Param Shah (prs392)

1. <u>Implementation</u>

1.1. Preprocessing

We start by preprocessing the data to filter out records and columns that are not required in the basic recommendation system. The basic recommendation needs only the 'user_id', 'book_id', 'rating' columns hence the others were filtered out. As per the Goodreads dataset, users with 0 rating signify the absence of ratings. We believe that such interactions can be treated as good as missing interactions and hence are removed from the dataset.

We introduce parameterized sampling in the scripts to be able to generate small subsets of data for prototyping and testing algorithms before hitting Dumbo with huge datasets.
Interactions are then split into train, validation and test sets using 60:20:20 partitioning. Validation and test splits are made in such a way that each user has an equal number of entries in the training set and the validation set (same applies for the test set). This is achieved using windowing functionality in Spark SQL.

Splitting can leave some users in train, validation and test sets with very few interactions. We decide to drop these as they may not provide sufficient information for recommending other items. We drop all users who have less than 10 interactions in the train, validation and test sets after splitting.

Another side effect of splitting is having items in validation and test sets that were not observed during training. Since we are not using a content-based approach, such interactions won't add value to learning the latent factors and are hence dropped from the validation and test sets.

As a result of valid preprocessing, the dataset size reduces from 4 GB to 1.1 GB in train, 142 MB in validation and 142 MB in test sets.

1.2. Learning

We learn latent factors representations of users and books using the Alternating Least Squares (ALS) Algorithm. ALS learns a low rank matrix product representation of user-book utility matrix by alternating least squares optimization.

The ALS parameters we chose to tune are: rank, max iterations and regularization parameter.

We design the learning process as an independent parameterized downstream job to the preprocessing pipeline allowing for parallelization in a deployment architecture where the preprocessing pipeline and the learning pipeline could work using the consumer-producer paradigm.
Although the models in grid search can be trained in parallel, we perform a sequential grid search on the hyperparameters to ease the load on the Spark cluster and allow fair resource utilization while only searching for a modest set of parameters. In all we perform a search over 60 hyperparameter configurations.

The grid search parameter values range as follows:

Rank	10, 30, 50, 70, 90
Max Iterations	5, 10, 15
Regularization Parameter	0.01, 0.05, 0.1, 0.2

1.3. Spark tuning

We tune Spark parameters for optimized memory while maintaining fair resource utilization. We set the number of executors = 20, executor memory = 10 GB. We also use dynamic executor allocation to scale up and down resources as per Spark task requirements within a job. We run the job in client mode with driver cores = 4 and executor cores = 5 for task parallelization on the executors. All these properties can be controlled in the app.properties configuration file which is imported into every pyspark script.

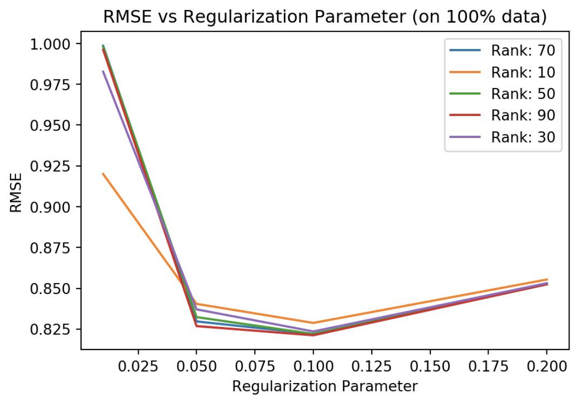
2. Evaluation

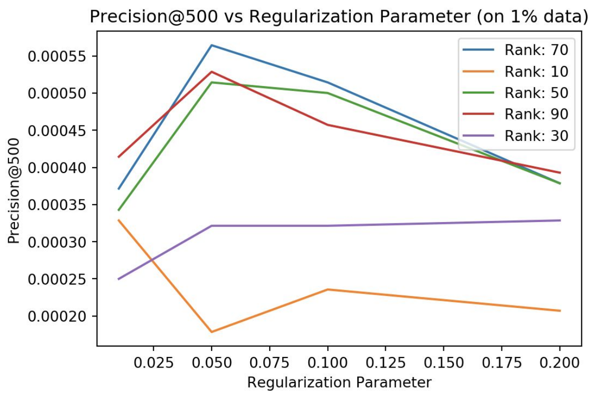
We evaluate the model using the following metrics: Root Mean Squared Error (RMSE), Mean Average Precision (MAP), Precision At k, Normalized Discounted Cumulative Gain (NDCG) at k
We want to make evaluations based on top 500 items, so we choose k = 500.

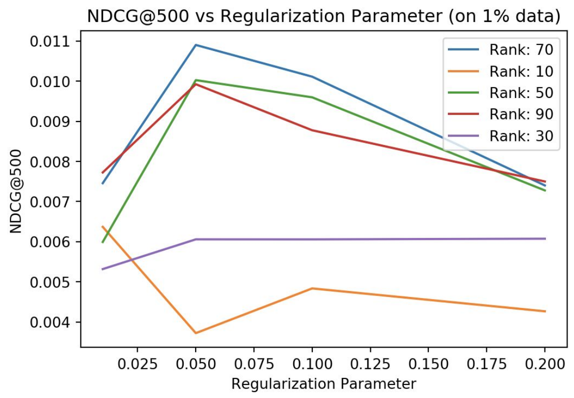
We observed that the RMSE Spark implementation is extremely optimized as is reported below for 100% of the dataset. However, the evaluation metrics do not seem to be well optimized for large datasets. For instance, in order to successfully compute a single MAP value for the simplest hyperparameter configuration, we required to launch 70 executors with 20 GB executor memory occupying over 1.5 TB out of 2.2 TB of the cluster space. This was met with much dismay from the HPC team and hence we decided to report these values only for 1% sample of the dataset.

3. Results

The results on the validation are described in the graphs below.







rank	num_iters	lambda	precision_500	RMSE_val	NDCG_500
70	10	0.05	0.000564286	0.829686622	0.010900939
90	10	0.05	0.000528571	0.826822738	0.009922817
70	10	0.1	0.000514286	0.822051921	0.010110986
70	15	0.05	0.000514286	0.827849194	0.010327937
50	10	0.05	0.000514286	0.832316928	0.010024756
50	10	0.1	0.0005	0.822008156	0.009596751
90	15	0.01	0.000485714	0.98197303	0.00900916
50	15	0.05	0.000464286	0.83069669	0.008878269

Top performing configurations on validation set

We observe that in general the best validation scores are for ranks 70, 90 and is indicative of the fact that the users and books are best represented in a 70 dimensional space. The best performance seems to be observed at 0.1 regularization parameter for RMSE and 0.05 for Precision at 500 and NDCG at 500. We also observe that NDCG and Precision at 500 give approximately the same performance plots. Some of the best performing models can be seen in the table above.

The best performing configuration on the validation set as per RMSE on the entire dataset is: Rank = 90, Max Iterations = 15, Regularization Parameter = 0.1 with a RMSE value of 0.8206.
The best performing configuration on the validation set as per Precision At 500 on 1% dataset is: Rank = 70, Max Iterations = 15, Regularization Parameter = 0.1 with a Precision At 500 value of 0.00056.

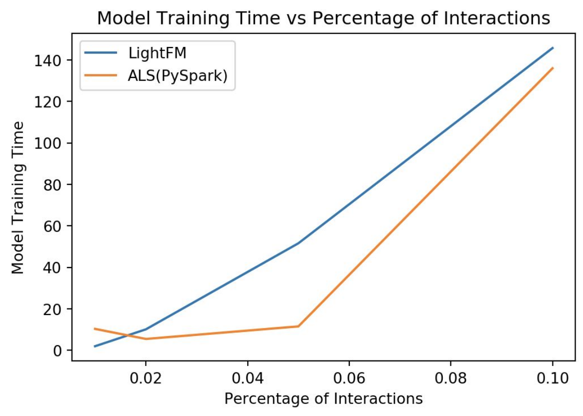
Since Precision at 500 is more representative of the ordering of items and well suited for recommendation systems, we make the model selection based on this value. A list of top performing models as per Precision At 500 can be seen above along with comparison with RMSE and NDCG.

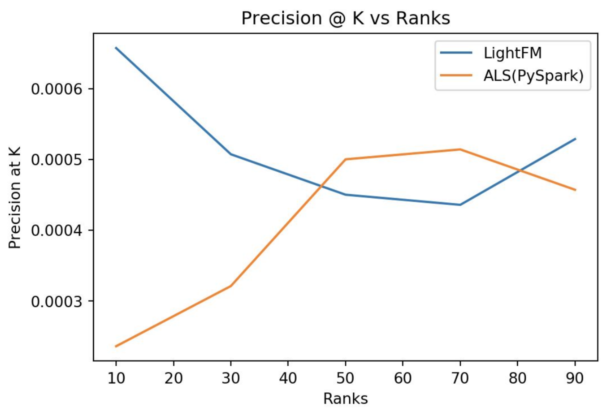
The test score of the best Precision At 500 configuration is [0.00051] for the configuration: Rank = 70, Max Iterations = 10, Regularization parameter = 0.05.

4. <u>Extensions</u>

We choose to compare collaborative filtering based ALS model with another model - LightFM - on a single machine implementation to see how parallelization benefits ALS in Spark. LightFM is a hybrid recommender that uses collaborative and content based filtering for making recommendations. The matrix factorisation represents users and items as linear combinations of their content features' latent factors. Because we wanted to compare the two models solely based on their performance on parallel vs single machine implementation, we chose to skip the content based input for LightFM. In our implementation we used the WARP loss that performed better than all the other loss functions implemented in the library.
We started by working on a very small dataset (size 0.01) on our local machine. We used the explicit ratings provided in the dataset to make our model predict scores for given user-interaction pairs. We quickly realized that the computation for a smaller dataset was faster than PySpark, but in contradiction, as the dataset size increased, the time taken for the model started increasing exponentially. It became impossible to train larger models on a single machine, hence we decided to restrict the sample set sizes for performance analysis to 1%, 2%, 5%, 10% of user-interaction data.
We observe that on a small dataset, the fit time for the model is linear in dataset size. However, for a large dataset, Spark scales much better than LightFM as LightFM is not even able to fit in a feasible time on large datasets on single machines. We observe that precision at 500 is fit differently by ALS and LightFM and gives a better performance on different configurations. We believe that this might be due to the differences in both algorithms. More importantly, ALS on Spark takes regularization into account whereas LightFM provides no controllable regularization parameter.

The test performance (precision at 500) for the best model using LightFM is 0.000664. The configuration of this model is : Epochs = 10, Rank = 10, Learning Rate = 0.05.





Contributions
Nikhil Supekar - Spark preprocessing, Spark tuning, Spark Ranking metrics, System Design
Param Shah - Spark RMSE evaluation, LightFM tuning, LightFM metrics, LightFM vs Spark Comparison

References

1. Collaborative Filtering - Spark 2.2.0 Documentation
2. LightFM 1.15 documentation
3. LightFM Hybrid Recommendation system
4. Research Paper - Metadata Embeddings for User and Item Cold-start Recommendations