Scalable Recommender System Using PySpark

Chao Huang, Lin Jiang, Shuo Yang, Han Xu
Data Science Institute, Columbia University, New York, 10025, US
ch3474, lj2493, sy2886, hx2282@columbia.edu

***Abstract*—In this project, we formulate our business problem as increasing retention rate of our customers, by providing them with personalized and accurate movie recommendations. Our target users are those with mid-level engagements. In order to solve this problem, we implemented two recommender systems using memory based and model based algorithms with PySpark. With extensive experiments, we not only fine tuned our model through cross validation, but also investigated how the performance will change with different parameter settings and different scale of dataset. In the end, we find the model with best performance and propose to do online testing. We also provide several ideas of improving the performance of the models and what we could add to make it a more productive recommender system.**

I. Introduction

This project is formulated under the scenario that we are an online digital media company providing movies for our customers. Our company aims to improve our profit by increasing our customers' stickiness and encouraging them to watch more movies they are interested in. The objective of this project is to focus on users who are neither very active nor inactive. We want to introduce a recommender system with the belief that well performed personalized and accurate recommendations for this group of customers. We choose to pay attention to this customer class because we think that active users have little potential for improving engagement and the reason for users being inactive is complex. Thus, we believe that it is more efficient to try improving the stickiness of users with mid-level engagements.

The recommender system should provide a reasonable number of recommended movies that the users are highly likely to watch, based on users' rating history. It should show improvements from the naive baselines. The model should also be scalable for long term use as we expend our business.

In the following sections, we will first conduct exploratory data analysis on our dataset in Section II. Then we will provide a brief introduction of the algorithms and implementation details of our models in Section III. By first introducing our experiment settings, we show our experiment results in Section IV. Finally, we conclude this project with several insightful findings and discuss what we can do next in Section V.

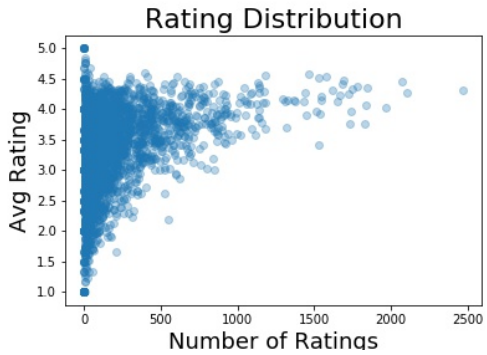


Fig. 1. Rating Distribution

II. Preliminaries

As initial data exploratory, we plot the average rating against the number of ratings to figure out the distribution of our data. We can easily observe that the majority of users tend to have ratings between 3 and 4.5, and only a few extreme ratings towards 1 and 5. This gives us the intuition that the top rated movies are not necessary the most popular, as they could be only rated by a minority of the users.

III. Our Proposed Method

In order to solve our business problem, we implemented two algorithms which are widely used both in academia and industry, namely k-Nearest-Neighbor (KNN for short) and Matrix Factorization (we refer to as MF or ALS_MF interchangeably in the following). In this section, we will explain how to implement these two algorithms using PySpark to make it scalable.

A. k-Nearest-Neighbor

In this project, we used item-based K-nearest neighbor collaborative filtering. The first step in this approach was to compute item to item similarity score. We used item-based approach instead of user-based one because first, item-based model is easier to scale since the number of movies tends to be fixed or grows a lot slower than the number of users, and second, the relationship between items are relatively static, which means we can reuse the similarity matrix.

In order to scale our model on large datasets, we utilized Spark Dataframe instead of pandas. As Spark dataframe has a data structure distributed over clusters, it's scalable and

TABLE I
STATISTICS OF THE DATASETS

dataset	#Users	#Items	#Interactions	Sparsity
ml-20m-5p	13807	1291	68969	99.61%
ml-20m-10p	13807	2534	135027	99.61%
ml-20m-20p	13807	5184	257487	99.64%
ml-20m-full	138493	27278	20000263	99.47%

can deal with large amount of data. However, there is no existing library implemented KNN algorithm using Spark, so we implemented it from scratch and tested its performance.

The routines of KNN model can be summarized in the following three steps,

- Self join the user-movie-rating table on "user" to calculate the inner product of two item vectors.
- Normalize the inner product using the norm of each item vector to get the similarity table of item pairs.
- Join the user-movie-rating table with the similarity table to find nearest neighbors and calculate inferred rating scores. Then by ranking scores, we can output the final Top-N recommendations.

For more implementation details, please refer to the code at src/model/KNN.py.

B. Matrix Factorization

For the MF, we utilized the existing ALS method in Spark recommendation module. However, this method may suffer from the "out of java heap space" error when the number of maxIter exceeds a certain number. One way to solve this problem is setting the check point directory of the current spark session but it's unclear whether this solution still works for larger datasets.

Also, we noticed that when training a MF model in implicit settings, negative sampling is a widely used technique to avoid overfitting and to better capture the final ranking metric during the training process. Therefore, we are curious about whether this techique may help when using explicit ratings for training. In our experiments, we fix the number of negative sample with each relevant rating in training set to 3 and call this model ALS_NEG.

IV. EXPERIMENTS

In order to check the correctness and efficacy of our implementations, we conduct experiments on three real-world datasets to answer the following questions.

- **Q1:** Are all models implemented correctly?
- **Q2:** How does our proposed method perform compared with the baseline models?
- **Q3:** How do the hyper parameters affect our models?
- **Q4:** Can our proposed method still work well for larger dataset?

In what follows, we first describe the experimental settings and then answer the above four questions.

A. Experimental Settings

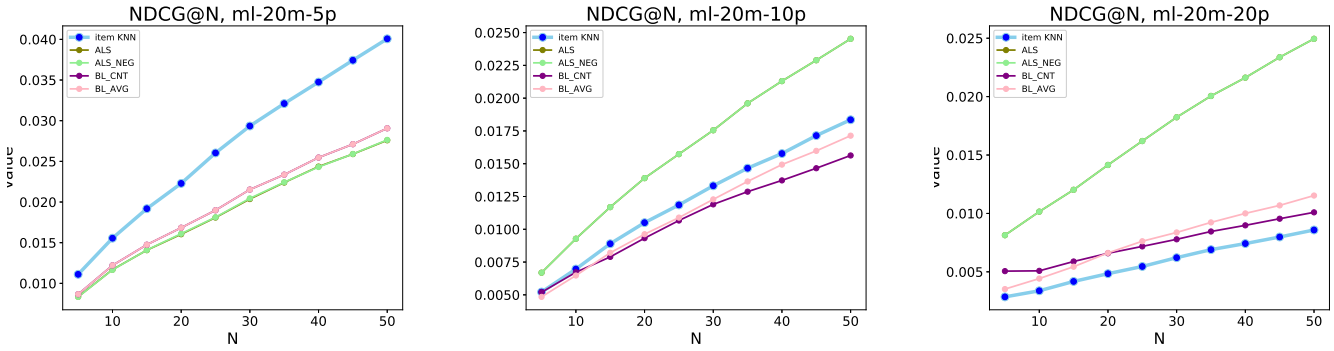
1) Datasets: We experimented with two real world datasets from movielens project[1], namely ml-1m dataset and ml-20m dataset. For the latter one, due to limited resources given for this experiment phase, we sampled three subsets with different number of interactions and guarantee that they can all fit in the memory under different models. For the sampling procedure, since our target customers are those with mid-level engagement with the websites, we filtered out users with more than 500 ratings in order to narrow our scope for this group of customers only. Furthermore, we sampled out 10% of the total customers in the original ml-20m dataset as our testing customer. And then, we sampled 5%, 10%, 20% of the total items respectively to get the sampled ratings. In the following, we will refer to them as **ml-20m-5p**, **ml-20m-10p** and **ml-20m-20p** respectively.

The statistics of these datasets are summarized in Table I. Compared with the original dataset, all three sampled datasets share similar sparsity level with the original dataset, so we may assume that the performance of our models in these small datasets are similar to that on the full dataset. And this is why we sample the dataset in a completely random manner instead of choosing popular items or users, so that we won't introduce any bias or assumptions through this sampling procedure.

2) Train Test Split: For the train-test split, we leave out 20% ratings from every dataset as the test set, and treat the others as the training set. The split is stratified to each user, in order to make user that almost every user exists both in the training set and in the test set. For cross-validation, we conducted 5-fold cross validation on the training set in order to find the optimal combination of hyper-parameters. After grid search, we will re-train every model with the optimal hyper-parameters and evaluate them on the left out test set.

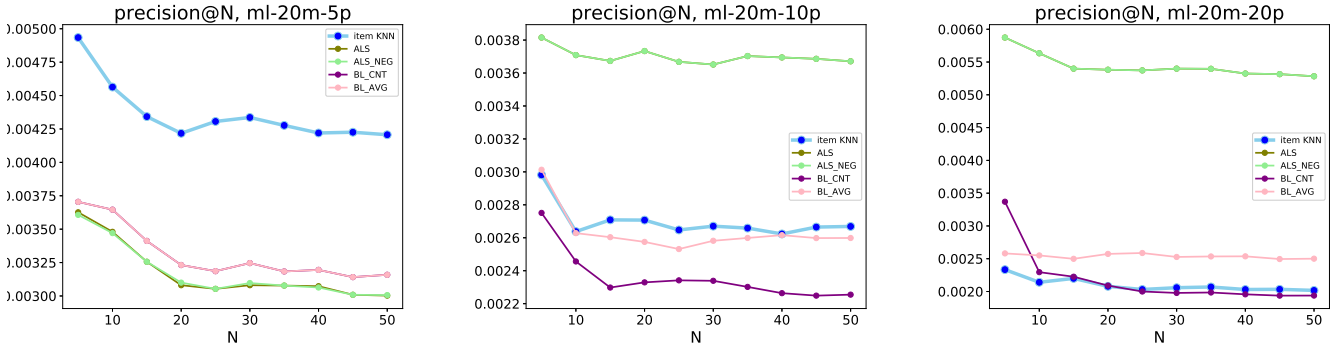
3) Evaluation Metric: In the evaluation stage, we evaluate our model with two groups of metrics, namely regression metric and ranking metric. For the first group, we calculate the **RMSE** (root mean squared error) between the prediction and ground truth of each rating in the test set. Although this metric can reflect how well our model fit the underlying pattern and generalize to unseen data, it doesn't make much sense in this business scenario. Since a smaller difference may lead to totally different recommendation results and it's hard to interpret how much error is fatal to the whole system. Therefore, after checking the correctness of our implementation, we evaluate our model on the second group of metrics. We mainly consider two ranking metrics, namely **Precision** and **NDCG** (Normalized Discounted Cumulative Gain). Precision evaluates that how many items among the top-N recommendations are truly relevant ones. Since Precision ignores the ordered positions where such hit happens, NDCG complements it by assigning a higher weight to those hits ranked higher in the recommendation list.

[1]https://grouplens.org/datasets/movielens/



(a) ml-20m-5p (b) ml-20m-10p (c) ml-20m-20p

Fig. 2. Top-N NDCG performance of different models



(a) ml-20m-5p (b) ml-20m-10p (c) ml-20m-20p

Fig. 3. Top-N Precision performance of different models

4) Baselines: We compared our models with two baselines without personalizations, namely **BL_CNT** and **BL_AVG**. The most intuitive recommendation is based on popularity. We measured the popularity for a movie based on either the total number of ratings (**BL_CNT**) or the average rated score (**BL_AVG**). For **BL_CNT**, we assigned 5 to the most frequently rated movie and 1 to the fewest rated movie. **BL_AVG** model might be bias for the movies which are only rated a few times. Recommendations to all users are the same for both baselines.

5) Parameter Settings: In order to fine tune our model, we search for the optimal combination of hyper-parameters through grid search. For KNN model, we explore the size of neighborhood (i.e., k) from 5 to 30 with a step length of 5. For MF models, we tune regularization term and number of latent factors through cross-validations. The grid search range for the optimal regularization term is $[0.005, 0.01, 0.05, 0.1, 0.5]$, and $[4, 8, 16, 32, 64]$ for the number of latent factors. The max number of iteration is fixed as 15, since a MF model optimized by ALS algorithm will always converge within 10 to 20 epochs.

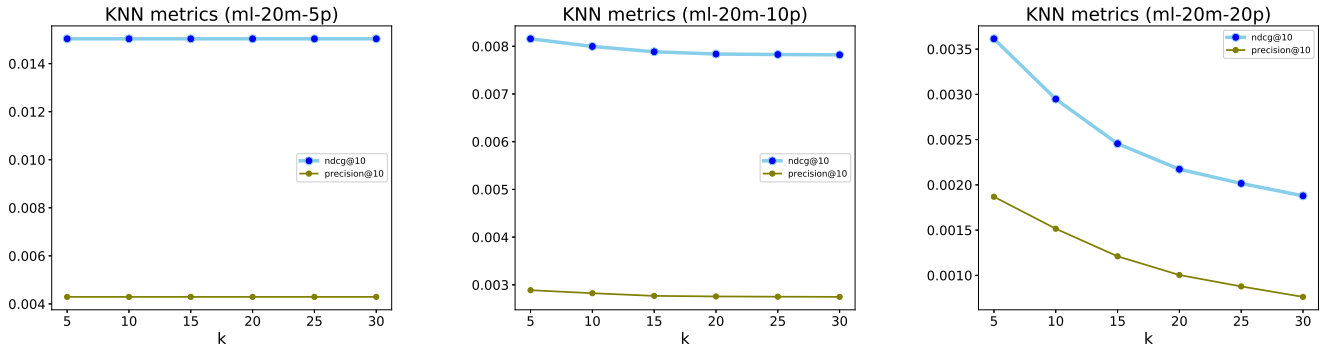
B. Model Correctness (Q1)

Since we implemented our KNN model from scratch, so the first thing we want to show is the correctness of our implementation. In this section, we first run our model on ml-1m dataset without any sampling. Also we just roughly fit our model using a set of default parameters without fine tuning and compare it with the benchmark found on the surprise package[2]. We get RMSE of 0.9328 when training a KNN model with $k = 20$ which is comparable with the k-NN benchmark with 0.923. From the results, our implementation of KNN and achieves similar RMSE performance with the benchmark, which indicates the correctness of our implementation.

C. Top-N Performance (Q2)

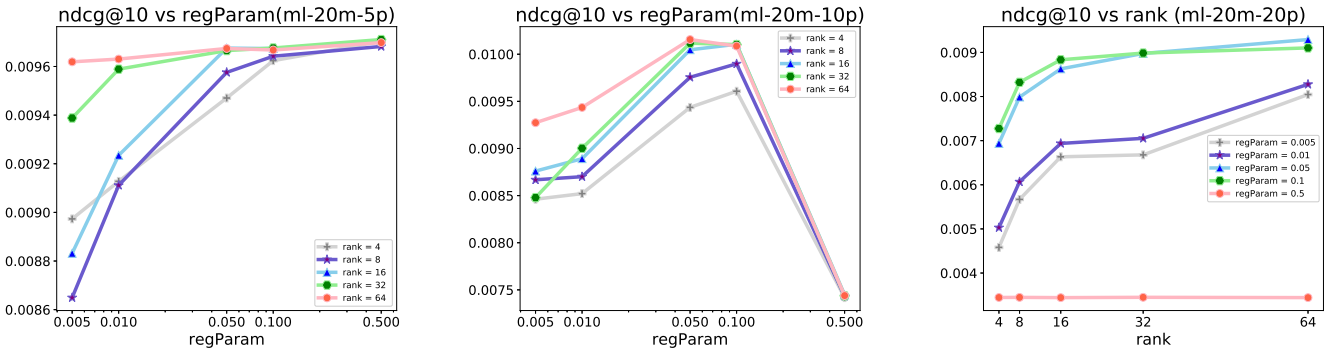
In our business setting, we may only recommend limited number of items to each user each time, due to the limited space on webpage, though showing more recommendations always leads to higher probability of finding what our customers exactly want. So we want to investigate how the performance of our model changes with regard to the number of displayed recommendations.

[2]http://surpriselib.com/



(a) ml-20m-5p (b) ml-20m-10p (c) ml-20m-20p

Fig. 4. The effect of hyper-parameters on KNN model, focusing on "k", using NDCG as the metric.



(a) ml-20m-5p (b) ml-20m-10p (c) ml-20m-20p

Fig. 5. The effect of hyper-parameters on the ALS_MF model, focusing on "regParam", using NDCG as the metric.

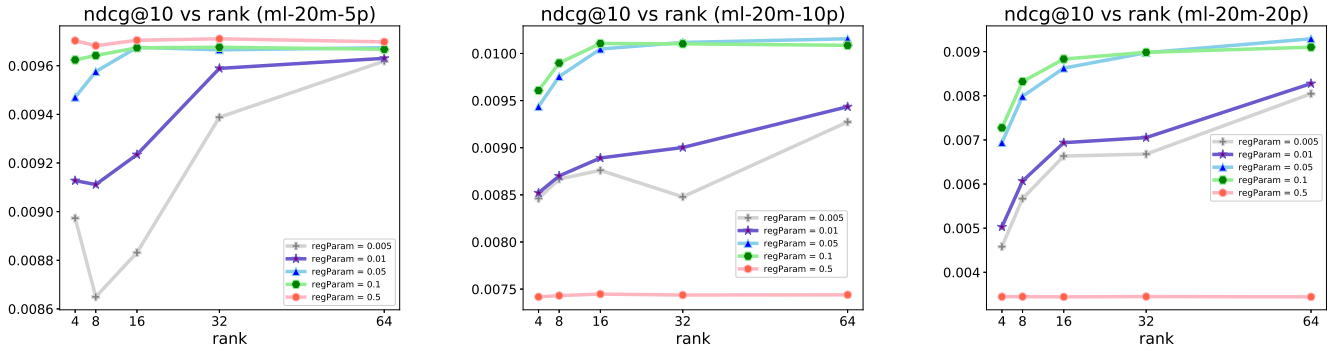
We compare the top-N recommendation performance with the baselines, by setting N from 5 to 50 with step 5. Note that the final ranking is based on all items in this dataset, which still makes top-50 recommendations quite challenging. We present the performance of NDCG@N from Figure 2(a) to Figure 2(c) for our methods and baseline methods, and Precision@N counterparts from Figure 3(a) to Figure 3(c). Note that the performance may vary from dataset, but the relative performance between models in a specfic dataset is what we really care. Here are our findings,

- For most datasets, MF models obtain the best performance in terms of NDCG@K and Precision@K as compared to all baselines. A special case would be the ml-20m-5p dataset. In this dataset, KNN model is the best and two baseline method surpass the MF models. Since this dataset includes the smallest number of training items, the MF models may suffer from the overfitting problem. For larger dataset, MF models show significant performance boost because of its higher model capacity.
- When varying the number of N in Top-N recommendations, NDCG tends to increase as N increases and Precision seems to fall at the same time. It's a natural phenomenon since it's more likely to include false-positive recommendations when we include those movies ranked low in the list. However, when N is larger than 20, the change in Precision would be flat. It implies that perhaps it's more robust to recommend more than 20 movies to a user at a time in order to avoid missing those movies ranked on the boundary of the Top-N recommendations.
- For two MF models, the one with negative sampling doesn't seem to have a significant difference with the other. It implies that using explicit rating records to train a MF model gives the model a wider range to learn the final parameters compared with binary records in the implicit settings. Since there is more information included in ratings, it's not necessary to perform negative sampling to avoid overfitting.

D. Impact of Hyper-parameters (Q3)

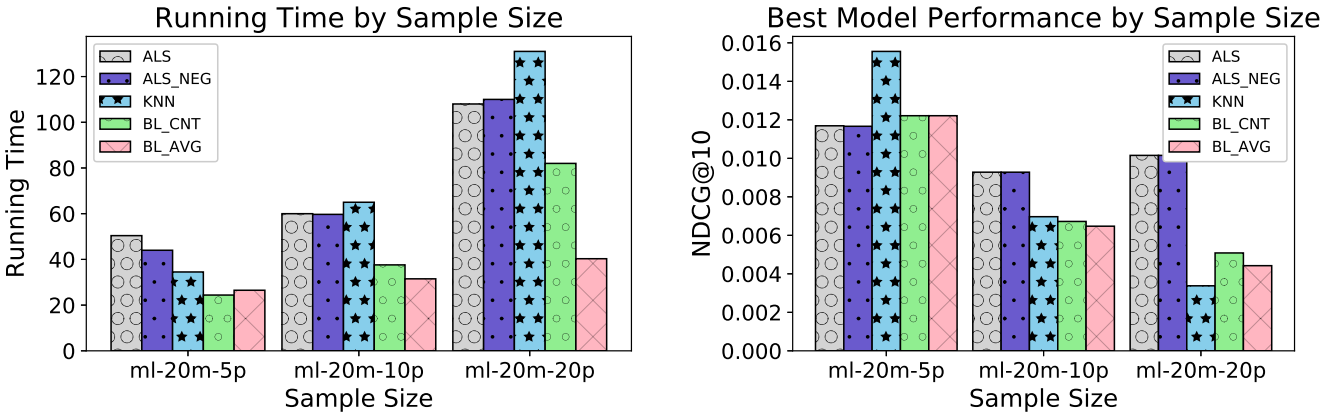
In order to understand how the hyper-parameters change the performance of our model, we plot the results of cross validations. Since whether we choose NDCG as the evaluation metric or we choose Precision, the final patterns are similar. So we only display the results with NDCG here.

For KNN model, we focus on the number of neighbors. As shown in Figure 4, for the largest dataset, ml-20m-20p, the



(a) ml-20m-5p (b) ml-20m-10p (c) ml-20m-20p

Fig. 6. The effect of hyper-parameters on the ALS_MF model, focusing on "rank", using NDCG as the metric.



(a) Running Time (b) Best Model Performance

Fig. 7. The effect of data size on the running time and best model performance, using NDCG@10 as the metric.

performance decreases with k, and for the other two datasets, the performances are not very sensitive to the choice of k. One possible explanation for this may be that the largest dataset contains the most interactions, so expanding the neighborhood may introduce noise into the final prediction. While for the other two, each user may not have as many as 30 interaction records in total, so expanding the neighborhood will not bring any differences.

For MF models, we focus on the regularization terms and number of factors (rank). We have the following findings,

- For the impact of regularization term, among the three datasets, two larger datasets share similar patterns while the pattern of the smallest one is somehow different. When the number of factor is fixed, for ml-20m-5p dataset, increasing the regularization term will always improve the performance of MF model. It's reasonable since the dataset is quite small, and large regularization term can help prevent overfitting. For larger dataset, there is an optimal setting for regularization term which is neither too big nor too small.
- For the impact of rank, we focus on the best performing ones with optimal settings of regularization terms. For these models, after rank is great than 16, increasing rank doesn't seem to have a significant effect of improving the performance. And it may even make the performance worse. This phenomenon is reasonable since large number of factors may lead to overfitting.

E. Impact of Data Size (Q4)

In this section, we study how well our models scale with datasets of different sizes. We run all experiments on a Ubuntu 16.04 server, with i5-9400F CPU (2.9GHz, 6 cores) and 16G DDR4 memory. All running time reported in Figure? is in the unit of second. Data sparsity and data size influences both running time and model performances. To be comparable, we used the same models as in ml-20m-20p for running time comparison. KNN is the most sensitive to data size. KNN is relative faster than ALS models in ml-20m-5p, while it is the most time consuming model as sample size increases. This is reasonable since KNN is a memory based model, and calculating item similarity matrix is a time-consuming task. Note that training KNN and MF model on the full ml-20m

dataset will both lead to out of memory error. For future work, we can try using approximate neighbor finding strategy to alleviate the memory problem of KNN and try using mini-batch stochastic gradient descent on MF model to fit a larger dataset to memory. For performance comparison, NDCG is used to compare our best models' performance under different data sizes. ALS models have more robust performance with different data size. KNN performs well for smaller sample size and its performance drops fast as sample size increases.

V. Conclusion and future work

In this project, we use both traditional memory based and model based algorithms. Through extensive experiments, we study the effect of hyper-parameters and data size and the pros and cons of each model. We find that Matrix Factorization has the best performance. Our models are also scalable on datasets with different sizes. We believe that a scalable recommender system using Matrix Factorization is robust enough for online testing. We propose to implement further online testing to evaluate its performance.

In the future, in order to make a more productive model, we think there are several things we can try to improve the performance and robustness of our model,

- Adapt two phase recommendation models to filter out irrelevant movies before training and evaluating the final ranking model. This is a widely used strategy in the industry. By constructing a recall module, we can first filtering out a small subset of movies that our users may be interested in and then feed the data to the ranking model. In this way, we can not only save the time of training models but also improve the final recommendation accuracy, since we are now training on a denser subset of the original dataset.
- Combine more content based features to help solve the sparsity problems. As both KNN and MF models suffer terribly from the lack of user item interactions, providing content features of items may be an feasible way to solve this problem.