Collaborative-filter Based Recommender System for Movies

Big Data Final Project Report

Sasha Richardson
Netid: sfr9746

Wonkwon Lee
Netid: wl2733

Abstract

In this project, we will develop and evaluate a collaborative filter recommender system using the MovieLens dataset(s). Our recommender system is implemented by Spark's alternating least squares (ALS) method to learn latent factor representations for users and items. In addition, we compare the performance of the parallel ALS model to a single-machine implementation using lightfm. Here is the GitHub repository for this project.

1 Introduction

The world is transforming from the age of knowledge towards the age of recommendations. Many companies have already successfully adopted recommendation systems, as a way of boosting their bottom line. Recommendation systems have made their way into our day-to-day online surfing and have become unavoidable in any online user's journey.

Collaborative Filtering is the most common technique used when it comes to building intelligent recommender systems that can learn to give better recommendations as more information about users is collected.

In this project, by using Spark's alternating least squares (ALS) method, we intend to develop a latent factor model to decompose the user-item interaction matrix into a product of user factor matrix U and item factor matrix V that consist of embeddings for users and items in a common vector space. Before implementing the sophisticated model, a popularity baseline model is developed.

2 Popularity Baseline Model

A baseline model is one we use to provide a first cut, easy, non-sophisticated solution to the problem. In much of the use cases for recommender systems, recommending the same list of most popular items to all users gives a tough-to-beat baseline. The implementation of the baseline model is conducted on local environment by some basic dataframe computations.

2.1 MovieLens Dataset

The MovieLens dataset is a benchmark dataset, consists of 25 million ratings and one million tag applications applied to 62,000 movies by 162,000 users. The *ratings.csv* has four columns, userId, movieId, rating, and timestamp. Each row represents an interaction between the user and movies. The *movies.csv* has three columns, movieId, title, and genres. All datasets are stored in NYU's HPC environment.

2.2 Data Preprocessing

The dataset to be used in the implementation was generated by merging the files *ratings.csv* and *movies.csv*, on the column *'MovieId'*.

For efficient implementation, the raw CSV-format data is converted to parquet format. Also, rating with zero scores and users with less than 10 ratings are filtered out. To split the dataset into train, test, and validation sets, *randomSplit* function is used to randomly split the original dataset. We set the split ratio as $0.6/0.2/0.2$, and partitioned the data into train/test/validation sets.

The test and validation sets are two disjoint datasets, but they should overlap with the training set. Hence, a portion of each user's rating will appear in the training set to prevent the user's embedding be purely random when evaluated in the test set. In addition to all user-item interactions of the training set, half interactions from the test and validation set are added to the training set. We use *zipWithIndex* function to divide test and validation interactions, and only half of each interaction is added to the training set. The other half of the interactions are kept as test and validation sets.

2.3 Implementation

The popularity baseline model was implemented using an item-based approach; specifically, the KNN model was used to train the baseline model. KNN measures the distance between similar movies to suggest recommendations of top-K nearest neighbors.

Finding the Nearest Neighbors use the unsupervised algorithms with *sklearn.neighbors*. The algorithm we use to compute the nearest neighbors is "brute", and we specify "metric=cosine" so that the algorithm will calculate the cosine similarity between rating vectors. To implement the KNN-based model, we first need to create a pivot table. In the pivot table, the index will be 'title', columns will be 'userId', and values in the table will be 'rating' of each user for the movies. The sparse matrix is created for the pivot table and fills the empty values with zeros.

Finally, we fit the model.

2.4 Results

```
Recommendations for Airplane! (1980):

1: Young Frankenstein (1974), with distance of 0.5565768755917464:
2: Princess Bride, The (1987), with distance of 0.557216384183876:
3: Groundhog Day (1993), with distance of 0.5682846361615 81:
4: Terminator, The (1984), with distance of 0.5833878362143257:
5: Star Wars: Episode V - The Empire Strikes Back (1980), with distance of 0.5848412867808372:
6: Back to the Future (1985), with distance of 0.5859048876760784:
7: Ferris Bueller's Day Off (1986), with distance of 0.5900226372390311:
8: Star Wars: Episode IV - A New Hope (1977), with distance of 0.603322859877855 9:
9: Aliens (1986), with distance of 0.6051973774087109:
10: Big (1988), with distance of 0.6074123959181021:
```

```
Recommendations for userId 11:

1: movieId 176, with distance of 0.6460420995567477:
2: movieId 486, with distance of 0.7044585254811317:
3: movieId 133, with distance of 0.7045454545454546:
4: movieId 602, with distance of 0.7073192634993539:
5: movieId 93, with distance of 0.7142094682854392:
6: movieId 235, with distance of 0.719926303448362:
7: movieId 229, with distance of 0.7259779660536856:
8: movieId 81, with distance of 0.7261468098918477:
9: movieId 485, with distance of 0.726905219663981:
10: movieId 33, with distance of 0.7315346713854005:
```

The baseline popularity model prints out the recommended items based on their popularity. The latent factors between the user and movie are not considered in the

baseline model. The recommendations for all users are saved as *baseline.csv*, and are later called on the *Spark* environment for evaluation.

2.5 Evaluation

After constructing the popularity model with basic dataframe computations, recommendations for all users are stored as a csv file. The baseline data is then loaded on the *Spark* environment for evaluation.

After we evaluated the list of recommended movies, we quickly identified two obvious limitations in our KNN approach: one is the 'popularity bias', and the other is the 'item cold-start problem'. Also, there will be another limitation, 'scalability issue', if the underlying training data is too big to fit in one machine

3 Latent Factor Representations

Matrix factorization is a factorization of a matrix into a product of matrices. In the case of collaborative filtering, matrix factorization algorithms work by decomposing the user-item interaction matrix into the product of two lower dimensionality rectangular matrices. One matrix can be seen as the user matrix where rows represent users and columns are latent factors. The other matrix is the item matrix where rows are latent factors and columns represent items.

Alternating Least Square (ALS) is also a matrix factorization algorithm and it runs itself in a parallel fashion. ALS is implemented in Apache *Spark* ML and built for a large-scale collaborative filtering problems. ALS is good at solving the scalability and sparseness of the rating data, and it's simple and scales well to very large datasets.

3.1 Implementation

The basic ideas behind ALS are:

- Factorize a big matrix into two small matrix (A = Users * Items)
- Use two loss functions for gradient descent
- Alternative gradient descent between Users and Items matrices back and forth

Hyperparameter tuning in Alternating Least Square:

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

The latent factor model is implemented using *PySpark.ml* package, a *Spark* DataFrame-based machine learning API. We train the latent model on the preprocessed training set and use ALS algorithm to fit our recommender system. The ALS algorithm factors a user-item interaction matrix into user factor matrix U and item

matrix V. The algorithm learns the latent factors between user and item and also tunes the factor weights. In addition, ALS is suitable for a parallel environment, which maximizes the efficiency of model tuning and implementation.

3.2 Evaluation Metrics

The recommender system is evaluated by *PySpark.MLlib* package, a *Spark* RDD-based API. The following metrics are used for evaluation:

- MAP: Mean Average Precision
- MAP @ K: Mean Average Precision of top K recommended items
- NDCG: Normalized Discounted Cumulative Gain of top K recommended items

The evaluation is based on predictions of the top 100 recommended items for each user. Furthermore, the model fitting time is recorded for each iteration to evaluate the efficiency. The ranking metric in *PySpark.MLlib* contains various evaluation metrics, appropriate for our project. Since *PySpark.MLlib* is an RDD-based API, we We decide to record the three metrics, mean average precision, mean average precision at K=100, and NDCG at K=100.

3.3 Hyperparameter Tuning

Rank	Regularization Parameter	Max Iterations
50	0.05	20

We use a grid search approach on the validation set to tune three hyperparameters: max iterations, regularization parameter, and ranks. Mean average precision is used for hyperparameter tuning metrics to optimize the model. For each iteration loop on the validation set, we record the three hyperparameters and the three evaluation metrics, MAP, MAP @100, and NDCG @100. The best model is then saved for efficient access and evaluation.

3.4 Results

Dataset	MAP	MAP @100	NDCG @100	Time
Validation	0.066084	0.077952	0.228651	27.112112
Test	0.074107	0.092508	0.251395	51.607537

The model performance with optimal hyperparameters on validation and test sets is recorded in the table. All other evaluation is recorded in *Report.md* in GitHub repository. The optimal hyperparameter values are rank of 50, regularization parameter of 0.05, and max iterations of 20.

4 Extension

4.1 Single Machine Implementation

We extended our project to compare *Spark*'s parallel ALS model to a single-machine implementation. For the single machine implementation, the *LightFM* package is used. We record evaluation score and model fitting time as a function of data set size.

4.2 Implementation

We started by creating an interaction matrix where rows represent each user and columns represent each movie id, with ratings as values. For this task, we used *pivot_table* from the *Pandas* package. Then, to build a matrix factorization model, we used the *csr_matrix* function from the scipy.sparse package. The input dataframe is transformed into CSR sparse matrix that can be used for matrix operations. We set the split ratio as $0.8/0.2$ and split it into train and test sets, corresponding with the data splitting method used for the *Spark* ALS model. We then build the *LightFM* model and then train the model.

4.3 Performance comparison with Spark's ALS model

We ran the model with both BPR and WARP ranking losses on the small and large datasets, with the same hyperparameter combinations as in *Spark* ALS.

Spark ALS and *LightFM* produce roughly the same magnitude of ranking metrics. There's an only a minor difference between *LightFM* WARP and BPR methods. Furthermore, comparisons between *LightFM* and *Spark* ALS, on precision at k and time, over ranks, and regularization parameters are relatively the same. With regards to time, implementation time varies as ALS supports a parallel environment, and *LightFM* does not. It is important to note that the overload on the cluster can affect the running speed, so comparing the performance based on evaluation time is difficult.

4.4 Limitation

LightFM has a limitation running on large datasets. Transforming the dataset with *LightFM* package creates a sparse matrix, which can cause dimension restrictions. Hence, we use a down-sampled dataset, which is also used for ALS implementation. In addition, the evaluation metrics in lightfm are limited compared to *Spark* recommendation evaluations or *sklearn* metrics. Also, there is no option to run training or inference on the GPU with *LightFM*, so it is challenging to scale up to the parallel environment.

5 Discussion

In this project, we have implemented a popularity baseline model, ALS latent factor model, and single machine implementation using *LightFM*, and have evaluated the

performance. Yet, there were some difficulties while conducting the project, and also there could be improvements.

The Peel cluster is a shared source, and an excessive number of tasks can trigger a bottleneck issue. Hence, measuring and evaluating the running speed is challenging for this project. Also, different libraries use different evaluation metrics, so it is rather hard to properly compare the performance, without implementing our own metrics. For example, *LightFM* does not support as many metrics as *PySpark*, and the evaluation of the baseline KNN model is not sufficient without importing it to the *Spark* environment. Fortunately, we observed better evaluation results with the ALS model.

Finally, the hyperparameter tuning with mean average precision takes a longer time than tuning with RMSE, in terms of memory and speed. We optimized the model with RMSE at first and later changed to optimizing with MAP. Multiple calling of *recommendForUserSubset* functions can easily run out of memory and slow down the speed. Downsampling the dataset into different ratios like 10%, 30%, or 50% could increase the efficiency.

6 Contribution

- Data processing and Partitioning: Sasha Richardson, Wonkwon Lee
- Baseline Popularity Model Implementation: Sasha Richardson, Wonkwon Lee
- ALS Model Implementation: Wonkwon Lee
- LightFM Single Machine Implementation: Sasha Richardson
- Project Report: Sasha Richardson, Wonkwon Lee

7 References

- MovieLens Dataset
- Spark Collaborative Filter
- Spark Ranking Metrics
- Data Partition
- LightFM
- Kaggle Popularity Baseline