A Collaborative Filtering Approach to Movie Recommender Systems

Sarvesh Patki
Center for Data Science
New York University
ssp6603@nyu.edu

Stephen Spivack
Center for Data Science
New York University
ss7726@nyu.edu

Abstract

In this paper, we design and implement a machine learning framework to recommend movies to users. Using the MovieLens 27M dataset, we first tested a baseline model which recommended the 100 most popular titles in terms of average rating to each user. Next, we tested a latent factor model which used matrix factorization – a state-of-the-art method for collaborative filtering – to make personalized recommendations. To extend our latent factor model, we performed a single-machine implementation and compared both the accuracy and overall computational efficiency between implementations. Our Github repository can be found here.

Keywords: recommender systems, ALS, matrix factorization, machine learning, big data, MovieLens

1. Introduction

Recommender systems are a subclass of machine learning algorithms that are built to predict the ratings or preferences of an individual user. Some of the more popular industry applications of this include media streaming, ecommerce and even online dating. Broadly, there are two approaches to recommender systems: content-based filtering and collaborative filtering. Content-based filtering systems learn the features of items that a user consumes in order to recommend similar – but novel – items for that user. For example, if the user of a music streaming service listens to music that is loud, has heavy bass and is high on "groove", the system might recommend other dance music that has similar auditory features for that user to consume. The MusicGenomeProject is one such large-scale effort based on this approach. Collaborative filtering systems, on the other hand, make recommendations for users by aggregating information about the preferences of all the other users in the database. Effectively, if User A and User B are similar, and User B consumes Product X and User A has not consumed Product X, the system should recommend Product X to User A. More intuitively, this is comparable to our music "soulmate" recommending that we listen to a given track for which we have no prior familiarity.

In this paper, we take a collaborative filtering approach to building and evaluating the performance of two movie recommender systems. The first approach is a baseline model which uses the overall popularity of a title to make recommendations. Here, movies with the highest average ratings are recommended to all users. This is not a personalized approach - all users are recommended the same movies. The second approach is a latent factor model which implements alternating least squares matrix factorization to decompose the utility matrix into two dense, reduced-dimensionality user and item matrices, which are embeddings in a common vector space. Matrix factorization is state-of-the-art for solving problems with sparse observations – such as the Netflix Prize [1] – and is well-suited for the purposes of this paper [4]. Finally, we will extend our latent factor model to a single-machine implementation and compare the accuracy and overall computational efficiency between implementations.

2. Related work

2.1. Matrix factorization for machine learning

Matrix factorization is a technique in linear algebra that decomposes a matrix into its constituent parts so that basic operations such as solving systems of linear equations, computing the inverse and so forth are more computationally efficient [3]. This technique has a toolbox of many routine algorithms such as QR decomposition, eigendecomposition and singular value decomposition. Matrix factorization is well-suited for problems with sparsely observed data and one such application with high industry value is building recommender systems. Here, the utility matrix – which consists of rows of users and columns of items – is decomposed into user and item features in a dense, lower dimensional latent space. When these user and item embeddings are matrix multiplied they produce some approximate solution that minimizes some error function – usually the mean-squared error. The origins of matrix factorization for recommender systems can be traced back to Simon Funk's 2006 blog post and are still considered state-of-the-art today.

2.2. Big data

One of the key challenges for recommender systems is handling the massive amounts of data inherent to the problem. To put this into perspective, Spotify has 82 million tracks and 422 million users, which presumably fills several hundred petabytes of disk space – though this statistic is not currently publicly accessible. Therefore, the parallel, distributed frameworks in the big data literature are crucial for optimizing the use of recommender systems in research and in application. Fortunately, the matrix factorization approach offers a trivially parallel solution where user and item embeddings are independent at the level of rows and columns, respectively, which works well when stochastic gradient descent is the algorithm used to optimize against the loss function [2]. Therefore, the distributed computing frameworks of today – most notably Apache PySpark – are well-suited for these kinds of problems.

3. Methods

3.1. Data

We used the classic 27M MovieLens dataset to train and evaluate both our recommender systems. This freely and publicly available benchmark dataset consists of self-reported movie ratings from 280,000 users across 58,000 titles. The dataset is represented as an *n* by *m* matrix, where *n* is the number of observed ratings across all users and *m* is a 4-dimensional vector of features – userId, movieId, rating and timestamp. Each rating is self-reported on a scale from 1 to 5 in 0.5 increments. See Table 1 for an example of ten records from the dataset. It is important to note that, as many of the ratings were not observed for a given user, the utility matrix is sparse (0.00166% is observed). We also report our results for the MovieLens 100k dataset, but this was mainly for dubugging purposes.

userId	movieId	rating	timestamp
1	307	3.5	1256677221
1	481	3.5	1256677456
1	1091	1.5	1256677471
1	1257	4.5	1256677460
1	1449	4.5	1256677264
1	1590	2.5	1256677236
1	1591	1.5	1256677475
1	2134	4.5	1256677464
1	2478	4.0	1256677239
1	2840	3.0	1256677500

Table 1. Ten records from the MovieLens 27M set.

3.2. Preprocessing

Preprocessing was performed using Pandas DataFrames. To partition our data into training, validation and test sets, we performed the following procedure: For the training set, we randomly sampled 60% of the observed ratings from every user. We chose to randomly sample instead of sort by timestamp – which is an available feature of this dataset – because movie ratings data are known to suffer from spatial autocorrelation effects that bias the stability of earlier ratings. For the validation set, we randomly selected half of the users and collected the remaining 40% of their observed ratings. For the test set, we collected the remaining 40% of the observed ratings from the remaining users. This procedure resulted in a 60%-20%-20% split where each user is represented in the training set as well as either the validation set or the test set. All three sets were saved as separate .csv files and directly ingested into our modeling scripts to increase the reproducibility [5] of our research.

3.3. Models

Modeling was performed using PySpark RDDs. All models were trained using the training data. Evaluation was performed using three ranking metrics – average precision, mean average precision and normalized discounted cumulative gain – from PySpark's machine learning library. Evaluation was performed on the validation data for hyperparameter tuning, as well as on the testing data to compute both the computational efficiency and accuracy of our models.

3.3.1 Baseline (Popularity)

First, we implemented a baseline model which uses the overall popularity of each movie title to make recommendations. Here, popularity is operationalized as the average user rating for a given title. Given a vector of ratings *R*

$$R = (R_1, R_2, ..., R_n)$$

where the size of *R* is greater than or equal to 30 – as per the central limit theorem – the baseline model computes the average rating for each movie *i* such that

$$P(i) = \frac{1}{R} \sum_{r \in R} r$$

where *P(i)* is the average rating for movie *i*. Once we computed *P(i)* for each movie title in the training set we sorted the values in descending order and recommended the top 100 to each user. We can consider this baseline model a benchmark against which we can compare more complex models.

3.3.2 Latent Factor (Personalized)

Second, we implemented a latent factor model – that is, the personalized model – which uses alternating least squares to decompose utility matrix *R* into two low-rank, dense matrices *U* and *V* which are embeddings of users and items into a common vector space. Here, parallelism is at the level of rows in *U* and columns in *V*. See Fig. 1 for a schematic of this model. The error function we are optimizing against takes the form

$$\min_{U,V} \sum_{(i,j) \in \Omega} (Rij - \langle Ui, Vj \rangle)^2$$

where the goal is to minimize the sum of the squared error with respect to utility matrix *R*.

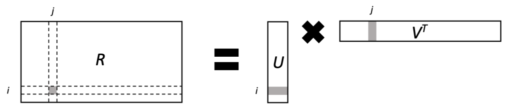

Figure 1. Schematic of the alternating least squares algorithm. Utility matrix R is decomposed into two dense, lower-rank representations U and V.

3.4. Experiments

All models were run over NYU's Peel (Hadoop) cluster by connection to a SparkSession. All scripts were written using Python, and dubugging was performed using the MovieLens 100k dataset. All models were trained using the training set, evaluated using the validation set to tune hyperparameters, and evaluted using the test set, where efficiency and accuracy were recorded for each. To extend this paper, we also trained and tested our latent factor model using LensKit, which is a single-machine implementation for recommendation systems. Here, we will compare the efficiency (training time) and accuracy between our PySpark and LensKit implementations.

4. Results

4.1. Baseline model

We trained our baseline model using the training set and evaluated it using just the validation set. Because there are no hyperparameters to tune for this model, this approach is justified. We evaluated our model using the following ranking metrics: average precision, mean average precision and normalized discounted cumulative gain. See Table 2 for a summary of our results.

size	prec	map	ndcg
100k	8.19e-4	5.96e-5	9.20e-4
27M	2.31e-6	1.54e-7	2.64e-6

Table 2. Baseline results for small and full datasets. First column is dataset. Second column is average precision. Third column is mean average precision. Fourth column is NDCG.

Looking at our results, we can see that in general our model performed better on the small dataset compared to the full dataset. This is true across all three ranking metrics we used for model evaluation. The best score – 0.00092 or 0.092% – was observed for the 100k set and NDCG metric. Overall, this is quite low – as expected – and we consider these baseline results a benchmark against we can compare our personalized model.

4.2. Latent factor model

4.2.1 Hyperparameter tuning with validation set

Before tuning our hyperparameters, we implemented the alternating least squares algorithm on both sets for *maxIter* = [5, 10, 15]. The rank and regularization terms were clamped at 10 and 0.01, respectively. See Table 3 for a summary of the results. For the small set, our results suggest that the model converges at *maxIter* = 10. However, for the full set our model converges at *maxIter* = 5. Therefore, to remain conservative, we decided to implement a *maxIter* = 10 for all our subsequent modeling both for hyperparameter tuning and for testing.

maxIter	prec	map	ndcg
5	0.015279	0.002059	0.199799
10	0.153770	0.001905	0.020395
15	0.015377	0.002058	0.019979

Table 3. Results on small set for our alternating least squares model for maxIter = [5, 10, 15].

maxIter	prec	map	ndcg
5	1.44808-5	2.32169e-6	2.02039e-5
10	1.45620e-5	2.30293e-6	2.01085e-5
15	1.45039e-5	2.30394e-6	2.01034e-5

Table 4. Results on full set for our alternating least squares model for maxIter = [5, 10, 15].

To tune our hyperparameters, we permuted matrix rank and regularization term across 20 experiments for the small set and 15 experiments for the full set. The small set had *rank* = [25, 50, 100, 200, 300] and *lambda* = [0.0001, 0.001, 0.01, 0.1]. The large set had *rank* = [50, 100, 200] and *lambda* = [0.001, 0.005, 0.01, 0.05, 0.1]. *maxIter* was clamped at 10 and the total run time was recorded across all experiments for both datasets.

Table 5 provides an overview of the results for the small dataset. In general, all scores increased as the rank parameter increased. This was expected, as *rank* controls the number of parameters and provides overall flexibility to the model – too many parameters will lead to overfitting. The regularization term appears to approximate an inverse-U function, meaning a value somewhere between 0.001 and 0.01 is the sweet spot for this implementation of alternating least squares. Overall, normalized discounted cumulative gain provides the highest score, followed by average precision and mean average precision. Our best was a NDGC score of 0.11405 for *rank* = 300 and *lambda* = 0.0001. We predict that this particular rank value will be suitable for testing, but that the regularization term will have to be increased an order of magnitude or two.

Table 6 provides an overview of the results for the full set. Similar to what was observed for the small set, in general all scores increased as the rank parameter increased – see Fig 2. Moreover, it appears that this effect is modulated by the specific ranking metric in the same way that we observed for the small set. However, it is important to keep on mind that

rank	lambda	prec	map	ndcg
25	0.0001	0.02085	0.00312	0.02713
25	0.001	0.02468	0.00453	0.03477
25	0.01	0.02714	0.00446	0.03795
25	0.1	0.02331	0.00380	0.03666
50	0.0001	0.03977	0.00853	0.05821
50	0.001	0.04154	0.00969	0.06476
50	0.01	0.04422	0.01038	0.06924
50	0.1	0.10028	0.00634	0.04781
100	0.0001	0.05396	0.01527	0.08592
100	0.001	0.05524	0.01479	0.08722
100	0.01	0.05511	0.01519	0.08949
100	0.1	0.03127	0.00705	0.05162
200	0.0001	0.06190	0.02055	0.10381
200	0.001	0.06383	0.02056	0.10679
200	0.01	0.06288	0.02020	0.10672
200	0.1	0.03213	0.00781	0.05490
300	0.0001	0.06455	0.02477	**0.11405**
300	0.001	0.06636	0.02297	0.11250
300	0.01	0.06344	0.02018	0.10796
300	0.1	0.03288	0.00819	0.055877

Table 5. Results for ALS model hyperparameter tuning on validation set for small dataset.

if the rank parameter is took high the model will overfit the data and generalize poorly to the test set. As such, due to the long training times on the full dataset, we decided to remain conservative with a max rank of 200. As far as the regularization term is concerned, the relationship between score and lambda is an inverted-U function, suggesting that lambda = 0.05 is the optimal value – see Fig 3. This is what we will use to evaluate our model on the test set.

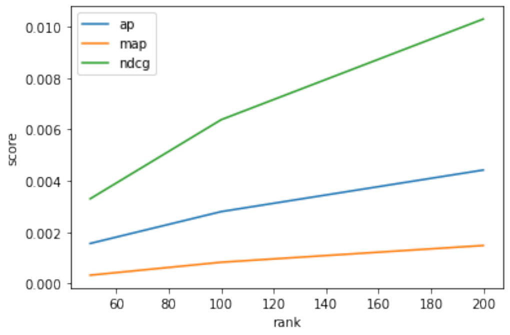

Figure 2. Score as a function of rank. Lambda clamped at 0.05. Blue line is average precision. Orange line is mean average precision. Green line is NDCG.

rank	lambda	prec	map	ndcg
50	0.001	6.51e-5	5.32e-6	8.18e-5
50	0.005	7.64e-5	4.69e-6	1.01e-4
50	0.01	2.20e-4	3.02e-5	4.06e-4
50	0.05	1.56e-3	3.24e-4	3.30e-3
50	0.1	2.24e-4	4.17e-5	5.04e-4
100	0.001	1.77e-4	1.42e-5	2.22e-4
100	0.005	3.78e-4	1.15e-4	8.08e-4
100	0.01	8.68e-4	2.94e-4	1.98e-3
100	0.05	2.80e-3	8.28e-4	6.38e-3
100	0.1	2.76e-4	4.72e-5	6.11e-4
200	0.001	6.96e-4	9.99e-5	1.03e-3
200	0.005	1.13e-3	4.15e-4	2.51e-3
200	0.01	2.02e-3	6.24e-4	4.46e-3
200	0.05	4.42e-3	1.48e-3	**1.03e-2**
200	0.1	3.09e-4	5.54e-5	6.86e-4

Table 6. Results for ALS model hyperparameter tuning on validation set for large dataset.

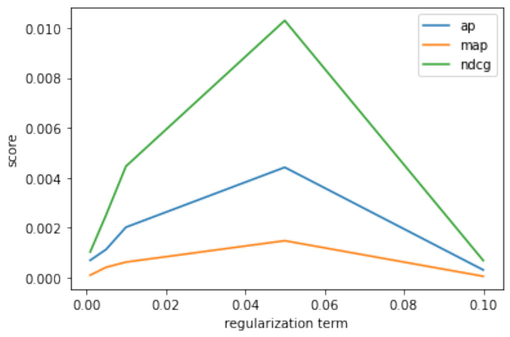

Figure 3. Score as a function of regularization term. Rank clamped at 200. Blue line is average precision. Orange line is mean average precision. Green line is NDCG.

4.2.2 Evaluation of model using test set

We evaluated our model on both the small and full test sets using *maxIter* = 10, *lambda* = 0.05 and *rank* = 300. Tab 7 provides a summary of our results.

size	prec	map	ndcg	run time
small	0.060721	0.024555	0.116333	517.5s
full	0.005111	0.001673	0.011998	4506.1s

Table 7. Results of our ALS Model. First column is dataset. Second column is average precision. Third column is mean average precision. Fourth column is NDCG. Fifth column is total run time.

In general, our model performed better on the small set than on the full set across all three ranking metrics. Moreover, the small set was about an order of magnitude faster than our full set. In general, normalized discounted cumulative gain yielded the highest score for both the small and full sets, with scores of 0.116 and 0.012, respectively.

4.3. PySpark vs. LensKit Analysis

To extend this project, we used LensKit to perform a single-machine implementation of our latent factor model. Specifically, we implemented the alternating least squares algorithm for both datasets and recorded the NDGC score and the run time, which acts as a proxy for overall computational efficiency. This allows us to compare our single-machine LensKit implementation to our cluster-based PySpark implementation. Table 8 provides a summary of our results for the LensKit implementation.

size	ndcg score	run time
small	0.096559	11.3s
full	0.03942	1446.8s

Table 8. Results of our LensKit Model. First column is dataset. Second column is NDCG score. Third column is run time.

For the small set, our LensKit ALS model yielded a NDCG score of 0.097, with a total run time of 11.3 seconds. For the full set, our model yielded a NDCG score of 0.039 with a total tun time of 1446.8 seconds, which is notably worse in terms of accuracy and overall performance compared to the small set. Compared to the cluster-based PySpark approach, the single-machine LensKit implementation performed better both in terms of accuracy and computational efficiency. In our discussion section, we will touch upon these two key findings: that the model performs better on the smaller set, and that LensKit was superior to PySpark in terms of accuracy and computational efficiency.

5. Discussion

In this paper, we took a collaborative filtering approach to movie recommender systems. Using the classic MovieLens dataset, we implemented a baseline popularity model as well as a personalized latent factor model. Moreover, we implemented a single-machine LensKit model to compare against our cluster-based PySpark model in terms of accuracy and computational efficiency. In general, we found that the baseline popularity model did not perform as well as the personalized latent factor model, which we expected to be the case. The whole purpose of the popularity model is to establish a baseline against which we can compare more complex models, such as the latent factor model we implemented here. In this regard, our results for the baseline model serve as a manipulation check.

The results of our latent factor model provided us with the key finding that it performs better in terms of accuracy and computational efficiency for the small set compared to the large set. This latter finding is not too surprising given that run time typically increases as a function of sample size, so it makes sense that the small set was significantly faster than the large set. However, the finding that the model has higher accuracy for the small set than for the large set is not as intuitive. However, the reason for this can be answered using basic linear algebra. In general, for large problems with sparsely observed data, model accuracy is expected to decrease as a function of sparsity – assuming both the model and hyperparameters are identical across dataset sizes. This makes sense, as very large matrices with very sparse observations will be more difficult to impute than smaller matrices with relatively less sparsity. In our case, we used the same model and hyperparameters for both the small and full sets, which provides evidence for this claim. However, we note a limitation in the approach to our hyperparameter tuning. Because the analyses performed in this paper were computed across a shared cluster, we could not feasibly test a wider range of hyperparameters. Therefore, we cannot conclude with certain whether we actually obtained the optimal hyperparameters for our alternating least squares model.

The results of our LensKit ALS model also provide us with the key finding that the single-machine implementation is superior to the cluster-based implementation, at least for the MovieLens 100k and 27M sets. Because the MovieLens 27M set is roughly 250MB, it is possible that it is not large enough to see any benefits from distributed, parallel computing. Future work could follow the same approach outlined in this paper using a much larger dataset. We predict that once the dataset reaches tens of gigabytes or more, the cluster-based approach will outperform the single-machine approach both in terms of accuracy and computational efficiency. That being said, for relatively small problems with a dataset that is on the order of a gigabyte or less, the single-machine implementation is the better option.

6. Contributions

Sarvesh Patki - data preprocessing, ALS model, extension (lenskit)
Stephen Spivack - data preprocessing, baseline model, hyperparameter tuning on ALS

References

[1] James Bennett, Stan Lanning, et al. The netflix prize. In *Proceedings of KDD cup and workshop*, volume 2007, page 35. Citeseer, 2007. 1

[2] Rainer Gemulla, Erik Nijkamp, Peter J Haas, and Yannis Sismanis. Large-scale matrix factorization with distributed stochastic gradient descent. In *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 69–77, 2011. 2

[3] Anshul Gupta, George Karypis, and Vipin Kumar. Highly scalable parallel algorithms for sparse matrix factorization. *IEEE Transactions on Parallel and Distributed systems*, 8(5):502–520, 1997. 1

[4] Yehuda Koren, Robert Bell, and Chris Volinsky. Matrix factorization techniques for recommender systems. *Computer*, 42(8):30–37, 2009. 1

[5] Marcia McNutt. Reproducibility, 2014. 2