Collaborative Filtering on MovieLens 20M

COMS 6998-02: Advanced Machine Learning for Personalization
Homework 1

Kevin J. Wu - kjw2157

February 23, 2018

Abstract

In this homework I perform collaborative filtering on the MovieLens 20M dataset, making use of sparse matrix libraries and stochastic gradient descent in order to efficiently train and evaluate large-scale matrix factorizations. I explore the effect of low-dimensional matrix rank and regularization on training and testing data for two evaluation metrics?root mean-squared error (RMSE) and mean reciprocal rank (MRR)?specifically examining the bias-variance tradeoff and generalizability of low-rank matrix factorizations on held-out data.

1 Introduction

Collaborative filtering is a common technique used by recommendation systems to collapse user preferences over a set of items onto a low-dimensional subspace. Given a fixed number of users, u, and a fixed number of items, d, we can construct a user-item matrix, R, with dimensions u x d, whose entries r_{ij} contain user i's rating for item j, if it exists, and 0 otherwise.

Matrix factorization seeks to approximate R with a low-rank representation $\tilde{R}$, which can be written as the product of a u x r user matrix, V, and a d x r item matrix, W.

$$R \approx \tilde{R} = VW^T$$

Call the set of non-zero user-item pairs (i, j) for the observed ratings Ω. For a given low-rank representation r, the objective function for matrix factorization can be written as:

$$\min_{U,V} \sum_{(i,j)\in\Omega} (R_{i,j} - (VW^T)_{i,j})^2 + \lambda(||V||_F^2 + ||W||_F^2)$$

where λ is a tunable parameter that controls the degree of regularization. We can minimize this using standard gradient descent techniques (see Section 3 for details).

2 Dataset

The data for this project comes from the MovieLens 20M dataset (https://grouplens.org/datasets/movielens
The dataset contains approximately 20 million movie ratings, spanning 26,744 movies and 138,492 users. Each rating consists of a real-valued score from the set $\{0.5, 1, 1.5, 2, 2.5, 3, 3.5, 4, 4.5, 5\}$.

3 Code and Methods

3.1 Bias Terms

Bias terms for the global bias (μ), the user bias (b_v), and the item bias (b_w) were added to the matrix factorization formulation given in Section 1. Now, the matrix factorization formula for a given rating is expressed as the following:

$$r_{ij} \approx \tilde{r}_{ij} = \mu + b_{v,i} + b_{w,j} + v_i \cdot w_j^T$$

where μ is a scalar, b_v is a row-vector of length u, and b_w is a column vector of length d. Similarly, the modified objective function is:

$$\min_{V,W,b_w,b_v} \sum_{(i,j)\in\Omega} (r_{i,j} - (\mu + b_{v,i} + b_{w,j} + v_i \cdot w_j^T))^2 + \lambda(||V||_F^2 + ||W||_F^2 + ||b_v||_F^2 + ||b_w||_F^2)$$

Note that we include the bias terms b_v and b_w as part of the regularization.

Adding in bias terms adds additional degrees of freedom to the collaborative filtering model and resulted in an overall improvement in RMSE scores across different values of λ and rank r. In this assignment, the global bias μ is calculated as the empirical mean of all ratings in Ω, and b_v and b_w are calculated using gradient descent.

3.2 Gradient Descent: SGD, Momentum

Stochastic gradient descent (SGD) was used to train the collaborative filtering model, in large part due to the need to avoid performing large matrix multiplications make it feasible to train the model in reasonable time on a personal computer.

For a given (i, j) pair in the training set, Ω, gradients of the objective function J for the bias terms and user and item matrices are calculated as follows:

$$\frac{\partial J}{\partial v_i} = -(r_{i,j} - (\mu + b_{v,i} + b_{w,j} + v_i \cdot w_j^T))w_j + \lambda v_i$$
$$\frac{\partial J}{\partial w_j} = -(r_{i,j} - (\mu + b_{v,i} + b_{w,j} + v_i \cdot w_j^T))v_i + \lambda w_j$$
$$\frac{\partial J}{\partial b_{v,i}} = -(r_{i,j} - (\mu + b_{v,i} + b_{w,j} + v_i \cdot w_j^T)) + \lambda b_{v,i}$$
$$\frac{\partial J}{\partial b_{w,j}} = -(r_{i,j} - (\mu + b_{v,i} + b_{w,j} + v_i \cdot w_j^T)) + \lambda b_{w,j}$$

Adding in a momentum term was found to speed up convergence of SGD to a local minima in this experiment. For each parameter θ we maintain a momentum term, z_θ, which maintains an exponential moving average of past gradients.

For every iteration t of SGD, evaluated at sample (i, j), the momentum updates and gradient updates are given by:

$$z_{v_i}^t = \gamma z_{v_i}^{t-1} + \eta \frac{\partial J}{\partial v_i} \quad , \quad v_i = v_i - z_{v_i}^t$$
$$z_{w_j}^t = \gamma z_{w_j}^{t-1} + \eta \frac{\partial J}{\partial w_j} \quad , \quad w_j = w_j - z_{w_j}^t$$
$$z_{b_{v,i}}^t = \gamma z_{b_{v,i}}^{t-1} + \eta \frac{\partial J}{\partial b_{v,i}} \quad , \quad b_{v,i} = b_{v,i} - z_{b_{v,i}}^t$$
$$z_{b_{w,j}}^t = \gamma z_{b_{w,j}}^{t-1} + \eta \frac{\partial J}{\partial b_{w,j}} \quad , \quad b_{w,j} = b_{w,j} - z_{b_{w,j}}^t$$

3.3 Python Implementation

Implementing collaborative filtering in an efficient way proved to be an engineering challenge. For this assignment, all calculations were done using Python and the SciPy and NumPy numerical programming packages.

3.3.1 Sparse Matrices

The full user-item matrix is a 138,492 by 26,744 matrix (containing over 3.7 billion entries!); a full, dense representation of such a matrix would take up over 30 GB, which is more than most computers can store in RAM. However, we can take advantage of the sparseness of the data in order to avoid storing the user-item matrix in its entirety. Specifically, the SciPy COO format for sparse matrix representation was used to store the full dataset, along with the test and training splits. The COO matrix format stores sparse matrices as an array of (i, j, v) triplets, where v is the value of the non-zero entry at row i, column j.

3.3.2 Train/Test Split

The full ratings dataset was split into training and testing datasets of approximately equal size. In order to ensure that all users and all movies appeared in the training data and in the testing data, the following procedure was used to generate the split:

1. Randomly split users into two equally sized sets, $users_a$ and $users_b$.
2. Randomly split items into two equally sized sets, $items_a$ and $items_b$.
3. Create training set by concatenating the ratings of $users_a$ on $items_a$ with the ratings of $users_b$ on $items_b$.
4. Create testing set by concatenating the ratings of $users_a$ on $items_b$ with the ratings of $users_b$ on $items_a$.

Again, this was all done without the need to convert the full user-item matrix to a dense matrix representation. Conversions to SciPy CSR format were used in order to efficiently index row-wise into the user-item matrix, and similarly, conversions to SciPy CSC format were used in order to efficiently index column-wise.

3.3.3 SGD

For SGD, the learning rate (η) was initialized to 0.01 and γ, the momentum parameter, was initialized to 0.5. After every epoch, η was either maintained or decreased by a factor of 10 depending on the change in the objective function from the previous epoch. γ was gradually increased after every epoch. In general, training was terminated after the objective function failed to show significant improvement from the previous iteration. However, in practice, other termination conditions were necessary, which will be described in Section 4.4.

3.3.4 Computational Resources

Training was by far the biggest computational bottleneck in this entire process. On average, the matrix factorization model took 25-30 minutes to train for one set of hyperparameters, with a single epoch of SGD (one full pass through the training data) taking around 4 minutes. RMSE computation on the training and test data averaged around 100 seconds each, and MRR computation averaged around 160 seconds. One small hack that was used to speed up training was instead of calculating the loss function over the entire training data after every epoch, a random sample of 100k ratings was used to calculate the loss.

Initially all computations were performed on a single core on a MacBook Pro equipped with a 2.2 GHZ quad-core Intel Core i7 processor and 16 GB of RAM. Later, a Google Cloud compute instance was set up on which multiple hyperparameter searches were executed in parallel in order to generate confidence intervals around the evaluation metrics. The Google Cloud instance type used was `n1-standard-8`, an 8 vCPU instance with 30GB of memory.

3.4 Evaluation Metrics

Matrix factorizations were evaluated using root mean-squared error (RMSE) and mean reciprocal rank (MRR), described below.

3.4.1 RMSE

The root mean-squared error for user matrix V and item matrix W on a held-out test set of user-item ratings, Ω_{test}, is calculated as:

$$RMSE_{test} = \sqrt{\frac{1}{|\Omega_{test}|} \sum_{(i,j) \in \Omega_{test}} (r_{i,j} - (\mu + b_{v,i} + b_{w,j} + v_i \cdot w_j^T))^2}$$

3.4.2 MRR

For a given user i, the mean reciprocal rank is calculated over the set of movies in Ω_{test} that user i ranked 3 or higher. For user i, call this set of movies Ω_i. If $|\Omega_i| > 0$, the MRR is calculated as:

$$MRR_i = \frac{1}{|\Omega_i|} \sum_{jin\Omega_i} \frac{1}{rank_i, j}$$

where $rank_j$ is the ranking of item j for user i, among the predicted scores for user i's held-out ratings. In other words, for each user's i's ratings in Ω_{test}, the predicted rating $\tilde{r_{i,j}}$ was calculated and ranked in descending order in order to generate $rank_{i,j}$.

4 Results

4.1 Grid Search

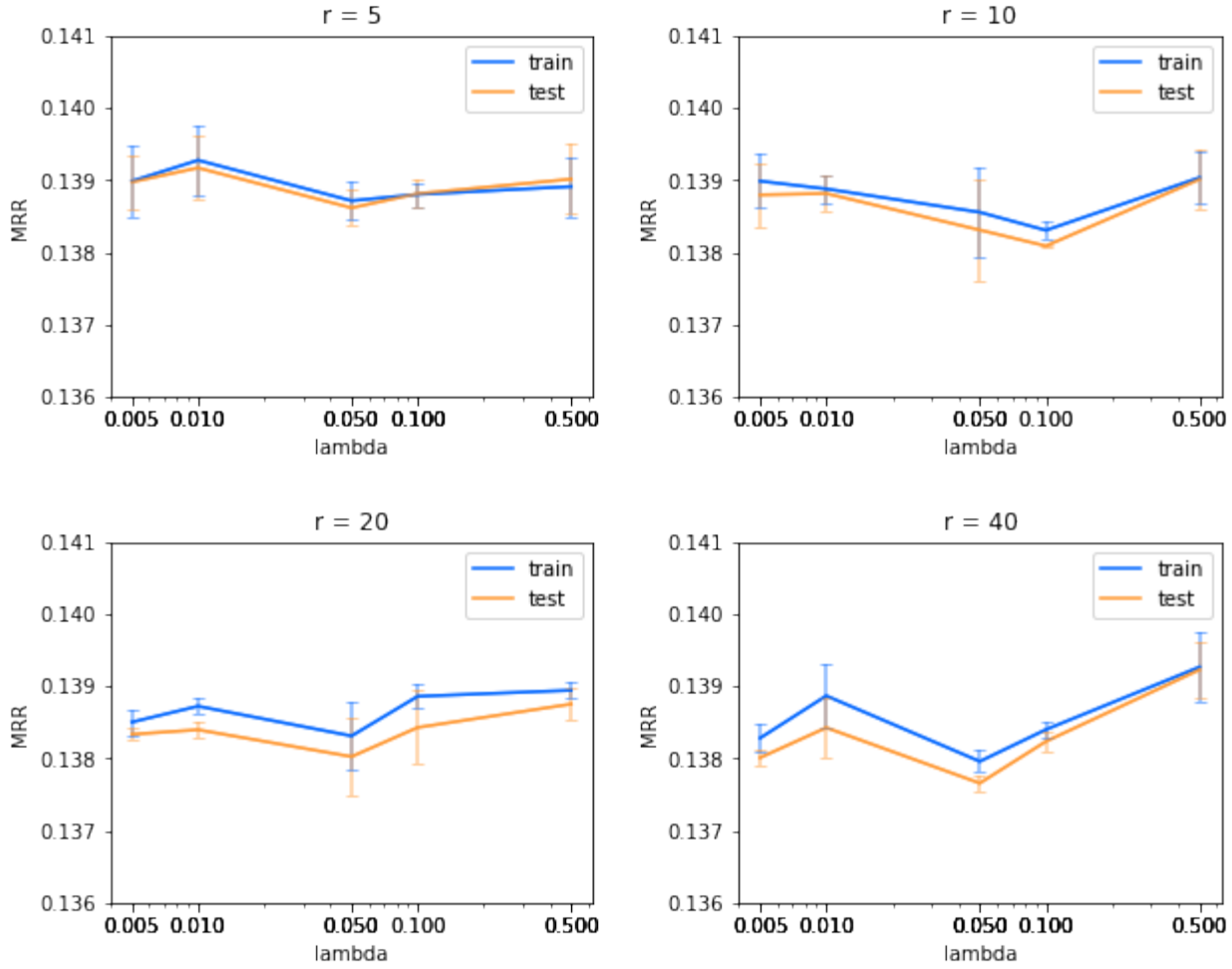
To examine the effect of matrix rank and L2 regularization on the model's performance, I performed a grid search over 4 different values of the rank variable r (5, 10, 20, and 40), and 5 different values of λ (0.005, 0.10, 0.50, 0.1, and 0.5). The two evaluation metrics were computed over various random training/testing splits, and the results are shown in Figures 1 and 2. Results on training data are shown in blue and results on test data are showed in orange. The vertical bars represent double-sided confidence intervals at the 95% level, calculated using the standard errors of the evaluation metric over multiple random splits of the data.

Table 1 reports a set of summary statistics for the grid search procedures.

4.2 MRR Analysis

First, one can see that MRR remained relatively low across all sets of hyperparameters (Figure 1), and across training and test data The average MRR value observed in this ex-

Figure 1: Effect of Varying Rank and Regularization on MRR

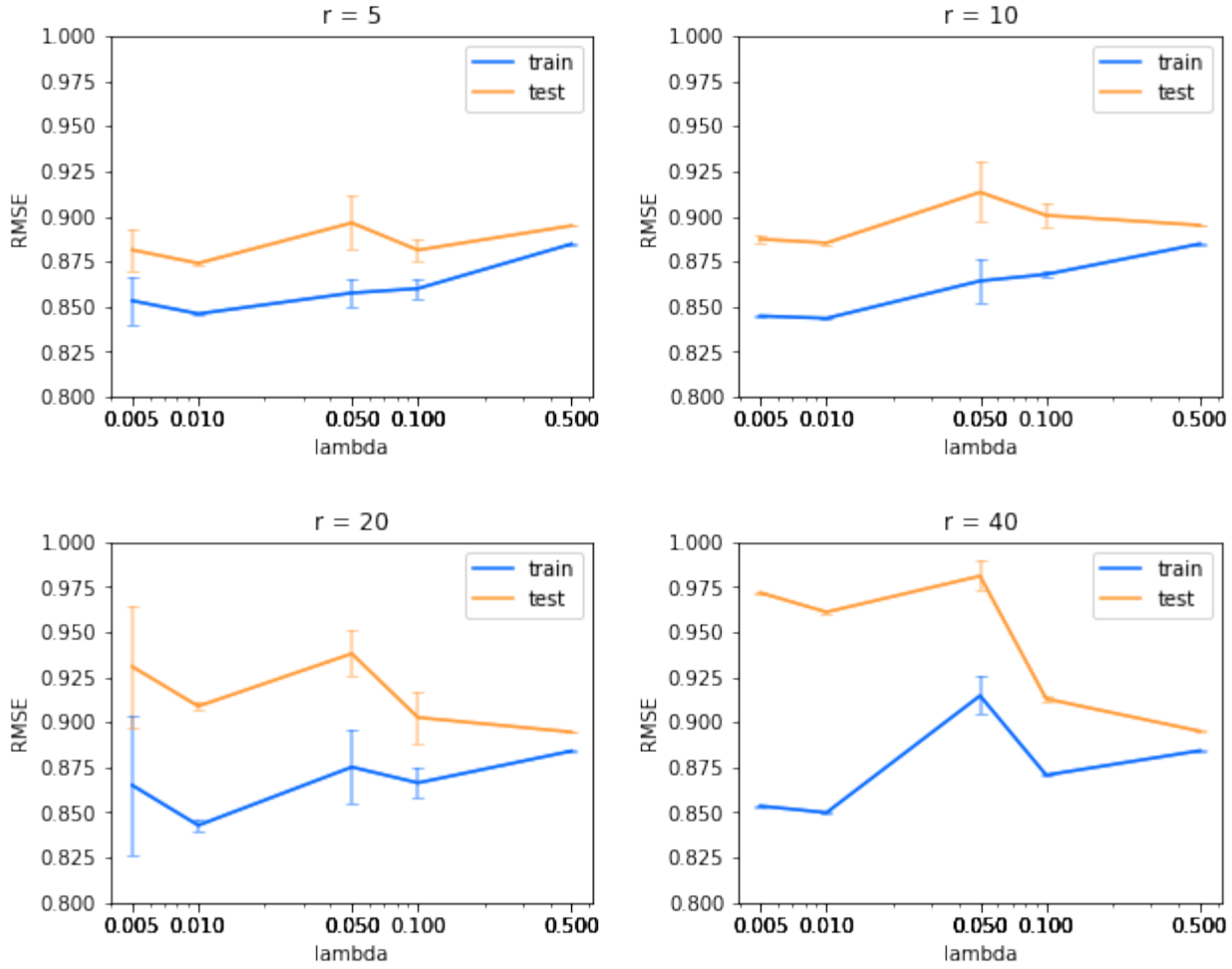


periment, 0.138, corresponds roughly to an average predicted ranking of just over 7 for a "highly-rated" (score > 3.0) item for a given user. This is not ideal; one initial hypothesis for the relatively low MRR is the fact that it is not directly being optimized for in the objective function.

4.3 RMSE Analysis

In Figure 2, we observe slightly more success by examining RMSE, where nearly all combinations of hyperparameters yielded a RMSE of less than 1. In general, lower-rank representations tended to achieve slightly lower error on the test dataset, with an optimal RMSE of 0.875 and 0.88 for $r = 5$ and $r = 10$, as opposed to an optimal RMSE of 0.9 for $r \geq 20$. The optimal training error is relatively stable around 0.85 for all ranks, however, suggesting that low-rank representations can be equally as expressive as higher-rank representations on the training data without overfitting.

Figure 2: Effect of Varying Rank and Regularization on RMSE



Moreover, noticeable variations in RMSE were observed across λ and rank values. The first noticeable pattern is the effect of increasing λ on error. As the regularization parameter is increased, RMSE on training data increases noticeably in both small- and large-rank factorizations. However, the RMSE of the test data either remains relatively constant or decreases in λ, suggests that L2 regularization is "working" by encouraging a more generalizable, lower-variance matrix-factorization representation of the data.

We can also observe this tradeoff between bias and variance of our fitted model as we vary the rank r. While training data $RMSE$ is constant around 0.85 across ranks, the RMSE of the test data predictions is over 0.90 and as high as 1.0 when $r \geq 20$. However, we can see that regularization in the form of higher values of λ is able to correct for much of this overfitting; with $\lambda = 0.5$, the RMSE of both testing and training data predictions converge to 0.87-0.88.

Finally, we observe the very low variation in RMSE scores for matrix factorizations of rank 40, across all values of the regularizer. This is surprising, as increasing the underlying

matrix rank should result in a more overfit model; in fact, one should expect to see a greater variance in RMSE on both the test and training data for higher rank factorizations. One possible explanation for this is a failure of the gradient descent procedure to find a global minima, or escape local minima (where somehow, these local minima consistently appear across multiple random folds).

The following section discusses some of the obstacles encountered with the gradient descent procedure and possible improvements.

4.4 SGD Termination

One caveat to these figures is that while an effort was made to encourage SGD to converge to a local minima in all cases, this was not always possible as the due to floating point errors in NumPy. This was generally an issue with higher-rank representations (specifically, $r = 40$), but it was also apparent across all ranks with higher values of λ, as the regularization seemed to be encouraging many of the entires in the user and item matrices to saturate. In order to get around this issue, some of the gradient descent procedures were terminated early before any floating point overflow errors were encountered.

The effect of this "early stopping" is especially apparent in the large kink in the training and test error graphs for $r = 40$, $\lambda = 0.05$. One can speculate that the true RMSE across training data might lie between 0.925 and 0.95 for the test data, and between 0.85 and 0.875 for the training data.

Given more time, the immediate solution to this would be to allow user-item matrix values to saturate and clipping these values to 0. Another approach is to experiment with more adaptive learning rate decay schedule (perhaps on a per-parameter basis using a method such as Adagrad) that might allow gradient descent to converge more quickly and to a better (and less numerically problematic) local optimum.

5 Conclusion

In this report, I describe the general framework for matrix factorization of user-item preferences and show how large-scale collaborative filtering may be done in high-level numerical programming languages in Python without the use of specialized computer hardware (albeit with some difficulty). I run a grid search over hyperparameters and discuss the bias-variance tradeoff of matrix rank and regularization by analyzing the effect of these parameters on the RMSE on training and held-out data. Finally, I discuss some of the challenges of performing gradient descent on a dataset of this size and some irregularities in the final results.

The code for preprocessing the data, training matrix factorization using gradient descent, and running a grid search over hyperparameters is available and documented in the files `datahandler.py`, `modeleval.py`, `sgd.py`, and `main.py`. Figures were generated using

		RMSE (train)			MRR (train)			RMSE (test)			MRR (test)		
rank	**lambda**	*mean*	*sd*	*n*	*mean*	*sd*	*n*	*mean*	*sd*	*n*	*mean*	*sd*	*n*
	0.005	0.8533	0.0134	4	0.139	0.0005	4	0.8814	0.0121	4	0.139	0.0004	4
	0.01	0.8461	0.0004	4	0.1393	0.0005	4	0.874	0.0008	4	0.1392	0.0004	4
5.0	0.05	0.8576	0.0089	5	0.1387	0.0003	5	0.8964	0.0167	5	0.1386	0.0003	5
	0.1	0.86	0.0064	5	0.1388	0.0002	5	0.8814	0.0068	5	0.1388	0.0002	5
	0.5	0.8847	0.0002	4	0.1389	0.0004	4	0.8948	0.0001	4	0.139	0.0005	4
	0.005	0.8449	0.0007	4	0.139	0.0004	4	0.8874	0.0019	4	0.1388	0.0004	4
	0.01	0.8436	0.0005	4	0.1389	0.0002	4	0.8853	0.001	4	0.1388	0.0002	4
10.0	0.05	0.8643	0.0137	5	0.1386	0.0007	5	0.9134	0.0189	5	0.1383	0.0008	5
	0.1	0.8679	0.0011	2	0.1383	0.0001	2	0.9006	0.0047	2	0.1381	0.0	2
	0.5	0.8848	0.0006	3	0.139	0.0003	3	0.8951	0.0002	3	0.139	0.0004	3
	0.005	0.8651	0.0394	4	0.1385	0.0002	4	0.9308	0.0346	4	0.1383	0.0001	4
	0.01	0.8429	0.0034	4	0.1387	0.0001	4	0.9088	0.0024	4	0.1384	0.0001	4
20.0	0.05	0.8751	0.0236	5	0.1383	0.0005	5	0.9379	0.0143	5	0.138	0.0006	5
	0.1	0.8664	0.006	2	0.1389	0.0001	2	0.9026	0.0105	2	0.1384	0.0004	2
	0.5	0.8839	0.0003	4	0.1389	0.0001	4	0.8945	0.0002	4	0.1389	0.0001	4
	0.005	0.8536	0.0004	4	0.1383	0.0002	4	0.9717	0.0006	4	0.138	0.0001	4
	0.01	0.8499	0.0001	4	0.1389	0.0005	4	0.961	0.0008	4	0.1384	0.0004	4
40.0	0.05	0.9147	0.0151	8	0.138	0.0002	8	0.9809	0.012	8	0.1377	0.0002	8
	0.1	0.8707	0.0006	5	0.1384	0.0001	5	0.913	0.0015	5	0.1382	0.0002	5
	0.5	0.8842	0.0004	4	0.1393	0.0005	4	0.8951	0.0003	4	0.1392	0.0004	4

Table 1: Grid Search Results. Mean and standard deviation of RMSE and MRR on training and test data, along with the number of random splits tried (n).

`matplotlib`, and the code and the saved grid search results used for generating the final figures and table can be found in `Visualizations.ipynb`.