Recommender System using Low Rank SVD approximation

Apurv Priyam
apriyam3@gatech.edu
MS CSE-ISyE, 1st Year

Introduction

Recommender systems plays a vital role in success of e-commerce and internet-based companies such as Amazon, Netflix, YouTube etc. With the serious growth and overload of data, users and available options for a user, recommender systems have emerged as an important tool for recommending more useful options to users by providing information tailored for individual user. However, this era of 'big data' also poses many challenges like data sparsity and processing massive data more efficiently and accurately. In this project I will attempt to solve the recommendation system problem by reducing the dimension (using Singular Value Decomposition) and using online learning (incremental learning) based on SVD.

Recommender System is a method to learn individual preferences through machine learning techniques. Recommendations can be product recommendation (Amazon, Home Depot), friend recommendation (Facebook, LinkedIn), contents recommendation (Netflix, Pandora), application recommendation (iOS, android).

Literature review

With the rapid growth in data, users and items to choose, fast and efficient recommendation system has become a topic of interest. Many researches have been done to implement a faster approach by reducing the dimension. Zarzour, et al tried to achieve this by clustering similar users together based on their preferences and rating [1]. Reducing the dimension this way also helped in the sparsity issue by grouping the users and calculating their average ratings. Matrix factorization has proven to be effective method in reducing the dimensions of data while keeping meaningful information intact. Many other machine learning problems such as data compression, linear regression, topic modelling, background removal, page rank depend on matrix factorization techniques [7].

To make the recommender system scalable and efficient with newly generated data, these papers [2] and [4] tried to solve the incremental learning in recommendation system using SVD and [3] updated SVD for rank one matrix perturbation in an efficient way.

Methodology

In this project I will try to create a recommender system using different forms of low-rank matrix factorization and compare them with benchmarks (including user and item based collaborative filtering using cosine similarity on full rank matrix and matrix factorization learned through gradient descent). I will also aim to implement efficient incremental learning in case where either a new movie (or user) has been added in the system. Analysis of each of these methods will be conducted to compare performance in accuracy, speed, and

numerical stability against a benchmark method (SVD using gradient descent/cosine similarity on a complete ratings dataset). I am aiming to improve both accuracy and speed with respect to the benchmarks.

I will use dataset of user movie ratings [5] for this project. Since this will have many items, this matrix is expected to be sparse. Using low rank approximations to reduce the dimension space of data, I can potentially resolve the sparsity issue. We can extend the same approach to any kind of recommendation problem.

We want to update our recommender system if any new user, movie or reviews arrives. However, as the system grows, we cannot keep on running matrix factorization from the beginning, hence I will explore strategies like incremental learning to update the SVD in a more efficient way.

Algorithms

After filling the missing values and normalizing the rating matrix R, where R is a $m * n$ matrix with r_{ij} showing the rating user i has given to item j, I will try to build a robust recommender system using following low-rank approximation techniques:

1. Low-Rank SVD decomposition of R matrix and doing an inner product to predict the rating of user i and item j. It will have following steps:
    a. Compute SVD of R and obtain matrices $U, \Sigma \; and \; V$ of size $m * m, m * n \; and \; n * n$ respectively such that $R = U\Sigma V^T$. Perform the dimensionality reduction step by taking the top k values corresponding to k biggest entries in the Σ matrix.
    b. Calculate users and items matrices in smaller dimension space. $R = U_K \Sigma_K V_K^T = U_K \sqrt{\Sigma_K} \sqrt{\Sigma_K} V_K^T$. Here, $U_K \sqrt{\Sigma_K}$ represents the m users and $\sqrt{\Sigma_K} V_K^T$ represents the n items.
    c. To predict the rating of user i and item j, use inner product of $\left(U_K \sqrt{\Sigma_K}\right)[i, :] . \sqrt{\Sigma_K} V_K^T[:, j]$

2. Item-based filtering after reducing the dimension using SVD
    a. Same as above
    b. Same as above
    c. Take the $\sqrt{\Sigma_K} V_K^T$ matrix which represents the ratings of n movies by Kpseudo factors. Now using cosine similarity on this low dimensional item matrix, similarities between two items can be measured.
    d. For predicting the rating of a given user-item pair, weighted average can be taken of the closest items' rating where weights can be their similarity found in low dimension feature space.
3. User-based filtering after reducing the dimension using SVD
    a. Same as above, the only difference being the use $U_K \sqrt{\Sigma_K}$ (a low dimensional representation of m users given rating to K pseudo factors). Then user-based recommendation can be get by calculating similarities between users on this low dimensional matrix and calculating the weighted average.

While performing above steps I will use cross-validation to get the optimal values of hyperparameters like the value of K, number of optimal neighborhood side while calculating the weighted average etc. The best result of all the above-mentioned methods will be compared with the following benchmarks:

1. Collaborative filtering using cosine similarity approach on complete rating matrix
2. Matrix factorization of R into two matrices $U\ and\ V$ of size $m * K\ and\ K * n$ respectively such that $R = UV$. The matrices will be calculated using gradient descent while minimizing the total error

Incremental updates of SVD for new information:

Recommendation systems utilizing SVD face a challenge of scalability, since rating data is often large and sparse. In order to overcome this, incremental algorithms can reduce running time when the ratings matrix is constantly growing.

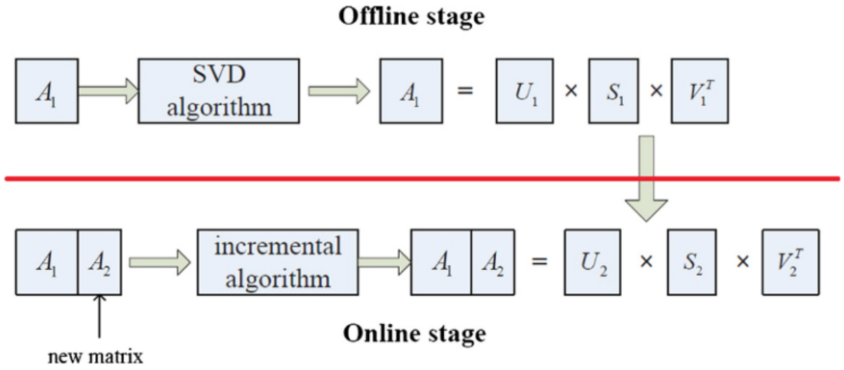

Figure: Flow chart of offline and online stage of updating SVD

For incremental SVD, consider an initial rating matrix $A = USV^T$. When A is updated with new information, I will use the properties of SVD to efficiently update U, S, and V without needing to recompute the entire decomposition. Let A_1 be new rating information received to be added to A and let $A' = [A\ A_1]$ then

$$A' = [USV^T, A_1] = U[S, U^T A_1]\begin{bmatrix} V^T & 0 \\ 0 & I \end{bmatrix} = UF\begin{bmatrix} V^T & 0 \\ 0 & I \end{bmatrix}$$

$$= U(U_F S_F V_F^T)\begin{bmatrix} V^T & 0 \\ 0 & I \end{bmatrix} = (UU_F)S_F\left(\begin{bmatrix} V^T & 0 \\ 0 & I \end{bmatrix} V_F\right)^T$$

Where F is defined as $[S, U^T A_1]$. Also, $SVD(A) = U_A S_A V_A^T$ where $U_A = (UU_F)$, $S\_A = S_F$, and $V_A^T = \left(\begin{bmatrix} V^T & 0 \\ 0 & I \end{bmatrix} V_F\right)^T$. Note that U_A and V_A^T remain orthogonal which allows to repeatedly perform this SVD update as new information comes in. Below is the pseudo-code for the incremental SVD algorithm.

Algorithm 1 Incremental SVD algorithm.

Input: matrices $A_1 \in \mathrm{R}^{m \times n_1}$, $A_2 \in \mathrm{R}^{m \times n_2}$; parameter $k \in Z^+$ satisfies $1 \leq k \leq \min\{m, n_1 + n_2\}$.
Output: matrices $U_k \in \mathrm{R}^{m \times k}$, $S_k \in \mathrm{R}^{k \times k}$, $V_k \in \mathrm{R}^{(n_1+n_2) \times k}$.

Using SVD on matrix A_1, we get three matrices U_1, S_1, V_1;
$F \leftarrow [S_1, U_1^T A_2]$, using SVD to decompose F and keep the rank-k approximation, we get three matrices U_F, S_F, V_F;
$b \leftarrow size(S_1, 2)$;
if $b < n_1$ **then**
    $V_k \leftarrow [V_1, zeros(n_1, n_2); zeros(n_2, b), eye(n_2)] \cdot V_F$;
else
    $V_k \leftarrow [V_1, zeros(n_1, n_2); zeros(n_2, n_1), eye(n_2)] \cdot V_F$;
end if
$U_k \leftarrow U_1 \cdot U_F$;
$S_k \leftarrow S_F$;
Return user eigenvector of $[A_1, A_2]$, i.e. $U_k\sqrt{S_k}(i)$ and item eigenvector of $[A_1, A_2]$, i.e. $\sqrt{S_k}V_k^T(j)$.

Computing SVD

1. **Jacobi Transformation**

I implemented my own version of SVD using Jacobi transformations as described in the textbook (Van and Golub) under section 8.6.4.

The algorithm does iterative Jacobi transformations on the symmetrical matrix (A'A) with orthogonal metrices to diagonalize it. I did one-sided Jacobi algorithm which do pairwise column orthogonalizations iteratively. Successive rotations overwrite the zeros in previous iterations to non-zero values. However, they got smaller and smaller in each iteration. At the same time columns of AV keep getting towards orthogonal, where V are the product of transformations metrices.

The U and S matrices are obtained by scaling the columns of AV. Each diagonal element of sigma is calculated as the length i.e. 2 norms of the corresponding column in AV. U is defined as $\frac{AV}{S}$. Now, permutations can be done to keep the values across diagonal sorted in the decreasing order.

2. **Bidiagonalization using Householder matrix**

I also converted the matrix to a bidiagonal matrix using QR decomposition via householder matrix. I applied householder to column and then to corresponding row to ameliorate the entries to zero. I sent this matrix to above Jacobi SVD. The Jacobi SVD performed much faster, took almost half of the time. But QR decomposition is slow, hence overall time taken in this process was more than the plain Jacobi Transformation.

Pseudo Code for Jacobi Transformation

```
function computeSVD(A, threshold)
    V = eye(size(A,2))
    while error > threshold
        for i in 1 to n
            for j in 1 to n
                J(i,j, θ) ← Jacobian( A(:, i), A(:, j) )
                error ← cos( angle between i and j of AᵀA )
                A ← A*J
                V ← V*J
    % here A is overwritten by A*V which has orthogonal columns
    for i in 1 to n
        S(i, i) ← norm(A(:, i))
        U(:, i) ← A(:,i) / S(i,i)

    return [U, S, V]
```

```
function Jacobian(x, y)
    % finding symmetric matrix
    % of these columns
    a ← x'*x
    b ← y*y
    c ← x'*y

    % finding cos(θ) and sin(θ) to
    % diagonalize the matrix
    β ← (b − a) / (2*c)
    t ← sign(β) / (|β| + √(1+β²))
    cos(θ) ← 1 / √(1+t²)
    sin(θ) ← c*t
    J = [cos(θ)  sin(θ); −sin(θ)  cos(θ)]
    return J
```

Data

The dataset used for analysis is the movieLens dataset collected by GroupLens Research. The movie rating dataset contains 100,000 ratings from 943 users over 1682 movie titles. The data was collected between 1997 and 1998 and are on a scale from 1 to 5 (integral) stars.

Results

The aim of the experiments is to find the optimal parameters settings for the recommender systems. After getting these parameters, different models will be compared in terms of error (RMSE) and run time.

Low rank for approximating the matrix plays a vital in efficiency of recommender system. Higher rank can reduce the stability and increase the over-fitting while lower rank will not capture the latent relations between the users and items. Based on RMSE vs Low Rank plot for different models I selected the Low Rank which minimized the RMSE on validation dataset. Figure 1 shows the RMSE on train and validation data and time (in sec) for different values of Low Rank.

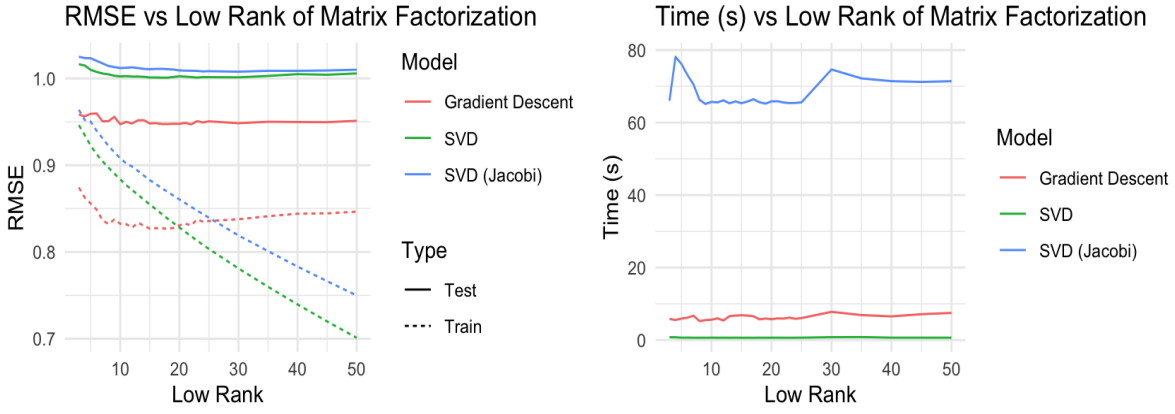

Figure 1: RMSE and Time for different values of Low Rank.

For the Item or User based recommender system enhanced with SVD, ideal number of neighbors for collaborative filtering was found. Figure 2 shows the experimental result for varying Low Rank and number of neighbors. One interesting observation was that the performance gets worse as I increased the neighborhood size with the worst performance at the top right corner of the heatmap. User based model required lower Low Rank value as compared to Item Based probably indicating user behavior can be explained with lower number of latent features.

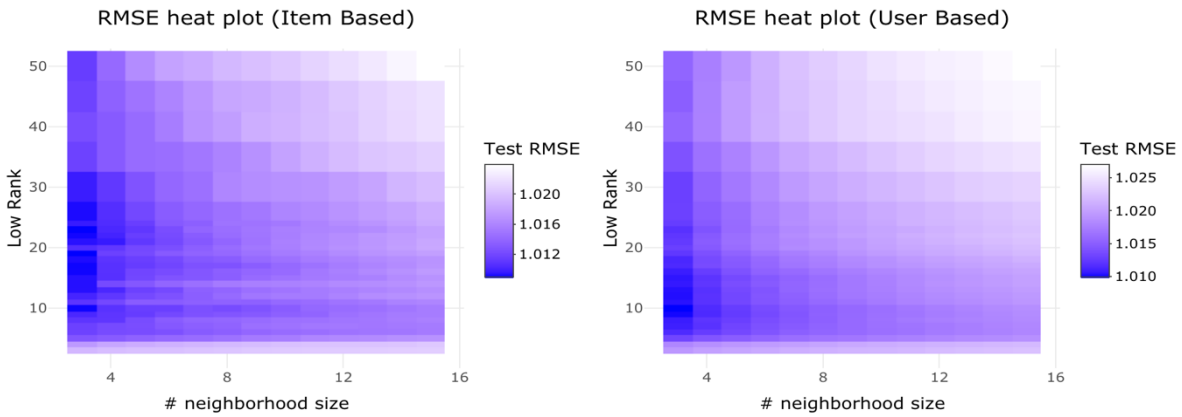

Figure 2: RMSE on test data for different values of Low Rank and neighborhood size for Collaborative filtering enhanced using SVD

I also followed similar approach to get the best neighborhood size for simple Item and User based collaborative filtering. As expected, as I increased the neighborhood size the train error increased but it made the model more stable with lower test error. After optimal neighborhood size the test error also started increasing.

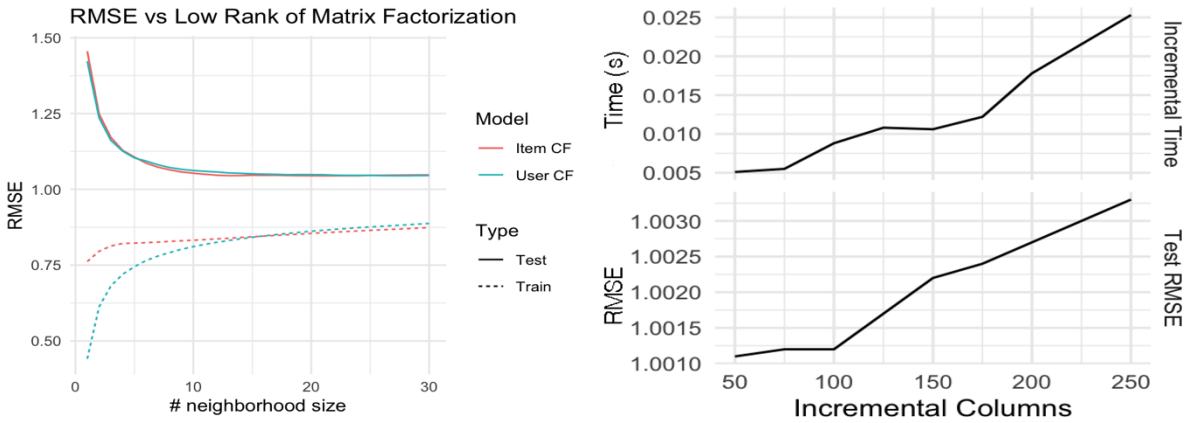

Figure 3: (Left) RMSE for different values of neighborhood size for collaborative filtering
(Right) RMSE on test data and incremental time taken by model for different incremental data size

Once I selected the best parameters of all the models, I compared their result. Table 1 shows the comparison with optimal low rank and neighborhood size.

	Low Rank	Neighborhood Size	Train RMSE	Test RMSE	Train Time	Prediction Time
Gradient Descent	22	-	0.8307	0.9411	5.92	0.03
SVD based	18	-	0.8385	1.0009	0.65	0.03
Item Based CF using SVD	19	3	0.8693	1.0089	0.74	0.22
User Based CF using SVD	10	3	0.9069	1.0099	1.09	0.40
Item Based CF	-	21	0.8569	1.0447	13.09*	3.11
User Based CF	-	28	0.8829	1.0448	14.02*	2.77

Table 1: Model comparison (* time may be inflated due to suboptimal code)

I used incremental learning by updating the SVD matrix for online learning. I used the Low Rank = 18 (optimal from previous experiments for SVD based model). Table 2 shows experiments results for different values of incremental data size (number of movies added for incremental learning). As I increased the incremental training size, both the test error and incremental training time increases (Figure 3).

# Incremental Columns	Train RMSE	Test RMSE (% Increase from Original RMSE: 1.0009)	Incremental Train Time (% of original train time: 0.65 s)
50	0.8389	1.0011 (0.02 %)	0.005 (0.79 %)
75	0.8388	1.0012 (0.03 %)	0.006 (0.85 %)
100	0.8388	1.0012 (0.03 %)	0.009 (1.36 %)
125	0.8391	1.0017 (0.08 %)	0.011 (1.66 %)
150	0.8393	1.0022 (0.13 %)	0.011 (1.63 %)
175	0.8402	1.0024 (0.15 %)	0.012 (1.88 %)
200	0.8406	1.0027 (0.18 %)	0.018 (2.74 %)
250	0.8414	1.0033 (0.24 %)	0.025 (3.90 %)

Table 2: Comparing RMSE and Incremental Learning time with learning on full data. Incremental learning includes recalculating SVD matrices after adding the new columns (movies).

I also compared the SVD based recommender system (using MATLAB's `svd` function) with my own SVD function. The process became slow, but I got almost the same accuracy (Figure 1).

Conclusions

After implementing various methods to solve the recommender system problem, I noticed that Regularized SVD learnt through gradient descent performed best in terms of error (had lowest RMSE) while SVD based factorization was the fastest. However, my self-implemented Jacobi SVD took more time than inbuilt SVD but had similar RMSE. Moreover, scenarios where extra items/users are added in the system, using incremental SVD saved a considerable amount of time (~98% time for adding 150 new movies) while keeping the RMSE almost the same. User and item based collaborative filtering also showed scope of improvement in terms of items/users uniqueness. We can conclude that since Gradient Descent and SVD lie on extreme ends in terms of error and time performance, there is a trade-off between using one over the other and it depends on the problem in hand.

Reference

[1] Zarzour, Hafed & Al-Sharif, Ziad & Al-Ayyoub, Mahmoud & Jararweh, Yaser. (2018). A new collaborative filtering recommendation algorithm based on dimensionality reduction and clustering techniques. 102-106. 10.1109/IACS.2018.8355449.

[2] https://www.cse.wustl.edu/~zhang/teaching/cs517/Spring12/CourseProjects/incremental%20svd%20missing%20value.pdf

[3] Gandhi, Ratnik & Rajgor, Amoli. (2017). Updating Singular Value Decomposition for Rank One Matrix Perturbation. https://arxiv.org/abs/1707.08369

[4] http://glaros.dtc.umn.edu/gkhome/fetch/papers/sarwar_SVD.pdf

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

[6] http://eigentaste.berkeley.edu/dataset/

[7] https://heartbeat.fritz.ai/applications-of-matrix-decompositions-for-machine-learning-f1986d03571a

[8] https://en.wikipedia.org/wiki/Jacobi_eigenvalue_algorithm