Latent Matrix Factorization for Movie recommendation

Kilol Gupta (kg2719)[1]

[1]Columbia University, SEAS

kilol.gupta@columbia.edu

Abstract. *This report is part of my project on latent matrix factorization for movie recommendation for the Advanced Machine Learning for Personalization class. The report will outline the specific details of this assignment which aimed at building a movie recommenda-tion engine that performs collaborative filtering using the famous MovieLens 20M dataset. More specifically, the recommendation engine performs matrix factorization and completion by learning 2 matrices namely V and W which are essentially decompositions of our dataset (formulated as a big sparse matrix) via low rank factorization.*

Index

1. Literature Review

Building recommendation systems is a challenging task, and collaborative filtering is one of the techniques that has proved to aid us in accomplishing this task. Traditionally speaking, user-based and item-based collaborative filtering techniques are computationally and theoretically simple to implement and understand, but matrix factorization is more effective because of the fact that it allows us to discover latent features that are representative of the user-item (user-movie in this case) interaction.

The idea is fairly simple. As the name suggests, our motive is to factorize the matrix at hand into two or more matrices (of smaller sizes) such that you get the original matrix back when you multiply these matrices.

For our task, we have to use this process to predict user ratings of the movies based on the ratings that we have. Rating set forms a matrix (users $\times$ movies) which we factorize into V (users $\times$ rank) and W (movies $\times$ rank) where V is representative of user-data and W is representative of movie data. We try to achieve the optimal values of these two matrices such that the product VW^T is as close to X as possible, or in other words, the ratings from X are close to predicted ratings that are obtained by taking a dot product of the i_{th} row (user) of V with the j_{th} column of W (transposed).

We can use any of the several optimization algorithms like adam, rmsprop, gradient descent etc. I have adopted the stochastic gradient descent algorithm to optimize the cost function.

Cost function for this problem looks like below:

$$E = \sum_{(i,j)\in\Omega_{train}} (R_{i,j} - (VW^T)_{i,j})^2 + \lambda(||V||_F^2 + ||W||_F^2)$$

Let's denote the overall error/cost as 'E'. The first term is the squared error per i,j pair (i: user, j: movie). This error is the difference between actual rating which user i gave to movie j and the predicted rating. We do this for only those ratings that appear in the training data set.

The second term is a **regularization** term that ensures that the parameters that we are trying to learn i.e. values in V and W matrices don't overfit to the training data and generalize well to the unseen examples. This particular is the L2 regularization as it adds the frobenius norm (euclidean norm for matrices) to the cost function. One can also use L1 norm as well to achieve L1-regularization.

Stochastic Gradient Descent: often shortened to SGD, it is also known as incremental gradient descent. It is a stochastic approximation of the gradient descent optimization and iterative method for minimizing an objective function. Briefly explaining the gradient descent process, this optimization algorithm tries to look for the direction in which we should move our parameter values to observe a reduction in the cost/objective function. It does so by calculating the gradient at a point and subtracting the product of gradient and learning rate from the parameter values. One disadvantage to this is that we might end up at a local optima and be stuck there. To overcome this, random initialization of the parameter values often helps. Tuning the learning rate appropriately also guides how good/bad our algorithm is working. High learning rate means that we are taking wider steps and might miss the optimum value, and low learning rate means that we are taking small steps and hence it might take us a really long time to converge.

To confirm that our optimization algorithm is working as expected, one can draw learning curves which shows the direction in which our algorithm is proceeding, and we can tune our learning rate accordingly.

$$\frac{\partial E}{\partial V_i} = -2(R_{i,j} - (VW^T)_{i,j})(W_j) + 2\lambda V_i$$

$$\frac{\partial E}{\partial W_j} = -2(R_{i,j} - (VW^T)_{i,j})(V_i) + 2\lambda W_j$$

Subtracting the above gradients from V[i] and W[j] respectively after multiplying the gradients with alpha/learning rate, we get the following update rule.

$$V_i = V_i + \alpha[2(R_{i,j} - (VW^T)_{i,j})(W_j) - 2\lambda V_i]$$

$$W_j = W_j + \alpha[2(R_{i,j} - (VW^T)_{i,j})(V_i) - 2\lambda W_j]$$

Programmatically, we should ensure that these updates happen simultaneously.

For testing the performance of our model, we split our dataset into training and test set and optimize our cost function by making updates while looking through the ratings in the training set. The test set is then used to evaluate our learnt model which in this case is V and W matrices. The evaluation metrics used are **Average RMSE** which stands for Average Root mean square error and **MRR** which stands for Mean Reciprocal Rank. the formulas for both are as follows:

$$RMSE = \frac{\sqrt{\sum_{(i,j)\in\Omega_{test}} (R_{i,j} - (VW^T)_{i,j})^2}}{\sqrt{size(\Omega_{test})}}$$

In the above equation, Ω_{test} refers to the test set of ratings which consists of the held out ratings used for evaluating the performance of our learnt model.

The way we split data into test and training, we ensure that every user is there in the training and test as well. Let U be the set of all users. Let Ω_u be the set of movies that are in the Ω_{test} which user u has rated 3 or higher. $rank_i$ is the rank given by user u to movie i. MRR is then defined as follows:

$$MRR = \frac{\sum_{u\in U} \frac{1}{|\Omega_u|} \sum_{i\in\Omega_u} \frac{1}{rank_i}}{size(U)}$$

Finally, the model is trained using different values of rank (number of latent features we are trying to learn) and lambda (the regularization constant). These are called as hyperparameters as opposed to parameters which stands for the values in V and W matrices that we are learning by optimizing the aforementioned cost function.

2. Dataset and its representation

This section describes the dataset, the splitting technique, splitting ratio, how I have loaded the data into my programming environment, the data structure that I have used to store it and multiple random folds of the dataset.

The MovieLens 20M data-set contains 20000263 ratings across 27278 movies as generated by 138493 users between January 09, 1995 to March 31, 2015. All selected users have rated at least 20 movies. The dataset of concern to us exists in movies.csv and ratings.csv. movies.csv stores movie id and movie name along with a few other columns. The ratings.csv file stores movie id, user id, the rating and timestamp.

Because of the size of our dataset, I decided to split the data into training and test

set at the csv level rather than loading the data first and then splitting it. Instead of the 50-50 split, I have chosen an approximately **70-30** split to speed up the MRR calculation step. My code for splitting the dataset is in splitData.py. Also, I have taken **three folds while splitting the dataset** and generated in total 6 CSV files. 3 for training and 3 for testing. Each training and test file pair correspond to one fold, the others corresponding to other folds. This is to obtain an average RMSE and MRR at the end and obtain more robust results.

The movies.csv is used to create a movie_id_dict which maps the movie id given in movies.csv to integers starting from 0. This dictionary is then used to create another dictionary called training_data_dict which has the following format- key: (user_id, movie_id), value: rating. I decided to choose a dictionary because of the sheer size of our dataset. Overall, we have around 20M datapoints. The user-movie-ratings dataset is usually perceived as a matrix with users in one dimension, movies in another dimension and ratings being the actual values stored. The size of this matrix would be no_users $\times$ no_movies i.e. 138493 $\times$ 27278. This big a matrix can't fit into memory. Therefore, instead of storing the ratings in a sparse matrix format, I decided to use **python dictionary data structure instead of matrix**. Overall, I had to store 20M ratings in the dictionary which is considerably smaller than 138493 $\times$ 27278.

3. Implementation Details

To split the data, I wrote a separate python program that reads in the ratings.csv file and creates a list for each user, splits this list into required proportion and extends the global lists: train_data and test_data. The code then writes these lists back into 2 csv files: train_data.csv and test_data.csv which essentially have ratings from all the users but split as suggested in the homework guidelines. This code is in file splitData.py

I have split the training and testing data into 70 : 30 ratio, i.e. 70% of overall data comprises of training data and 30% (rest of it) comprises of test data, as suggested in one of the piazza posts. I ran this programme of splitting the data set 3 times with random splits but keeping the overall proportion as 70:30. This was to satisfy the guideline of running code over multiple random folds. So in short, for every rank and lambda pair, I ran my training process 3 times (different training and test data), calculated the rmse and mrr and recorded their average.

Fold-1
Number of train_data ratings: 13287329
Number of test_data ratings: 6712561

Fold-2
Number of train_data ratings: 13380393
Number of test_data ratings: 6619497

Fold-3
Number of train_data ratings: 13332058
Number of test_data ratings: 6667832

To be able store these many movie ratings and make use of them, I decided to

use the python dictionary data structure. I made use of the functions build_movies_dict() and read_data(). The read_data() function takes the name of csv file as argument, reads it row by row and creates a dictionary with the following format- key: (user_id, movie_id), value: (rating). This format is chosen because we want our dictionary key to be unique and our dataset ensures that we have only one rating corresponding to the user-movie pair. In simpler words, each user would have rated each movie with a single rating. Dictionary is also used to store only those user-movie pairs that exist in the data set as opposed to storing a sparse matrix which wouldn't have been able to fir in memory.

```
# build movie dictionary with line no as numpy movie id,
# its actual movie id as the key.
def build_movies_dict(movies_file):
    i = 0
    movie_id_dict = {}
    with io.open(movies_file, 'r', encoding="utf8") as f:
        for line in f:
            if i == 0:
                i = i+1
            else:
                movieId = line.split(',')[0]
                movie_id_dict[int(movieId)] = i-1
                i = i+1
    return movie_id_dict
```

Figure 1. Code for build_movies_dict()

```
def read_data(input_file, movies_dict):
    # creating a dictionary, key: (user_id, movie_id), value: rating
    # these user_ids and movie_ids correspond to indices
    # of the hypothetical big ratings matrix
    X = {}

    # because we don't have a header row now
    i = 1
    with open(input_file,'r') as f:
        for line in f:
            if i == 0: # to escape the header row
                i += 1
            else:
                user, movie_id, rating, timestamp = line.split(',')
                m_id = movies_dict[int(movie_id)]
                X[(int(user)-1, m_id)] = float(rating)
                i += 1
    return X
```

Figure 2. Code for read_data()

Then comes the function matrix_factorization() and calc_error(). To reach an optimized implementation for these, I performed several experiments which are summarized in the below section.

3.1. Experiments

I used stochastic gradient descent, by this I mean that I am making an update on every data point by calculating the gradient at that point, and drew learning curves to confirm that the code is converging in the right direction.

For optimization, I tried doing it two ways so as to achieve a faster strategy. I have explained these two strategies below and my observations with each of these:

1. Stochastic gradient descent and then recording error after each update.

This error is calculated using an optimized function called optimal_calc_error which instead of going through all the data points, simply updates the previous error

with the difference corresponding to the user 'i' and movie 'j'. The code snippet for this is as follows:

```
def optim_calc_error(V_old, W_old, V_new, W_new, X, u_id, m_id, error):
    m_list = train_user_to_movie_dict[u_id]
    u_list = train_movie_to_user_dict[m_id]

    for m in m_list:
        error = error - (np.power(X[u_id, m] - np.dot(V_old[u_id],W_old[m]), 2))
        error = error + np.power(X[u_id, m] - np.dot(V_new[u_id],W_new[m]), 2)

    for u in u_list:
        error = error - (np.power(X[u, m_id] - np.dot(V_old[u],W_old[m_id]), 2))
        error = error + np.power(X[u, m_id] - np.dot(V_new[u],W_new[m_id]), 2)

    error = error - (lambd/2) * np.sum(np.power(V_old[u_id],2) + pow(W_old[m_id],2))
    error = error + (lambd/2) * np.sum(np.power(V_new[u_id],2) + pow(W_new[m_id],2))
    return error
```

Figure 3. Code for optim_calc_error()

The learning curve obtained after plotting the cost function/error vs iteration_number, is as shown below. This graph is plotted after 2046 iterations of stochastic gradient descent update and calculating this error using the optim_calc_error() function.

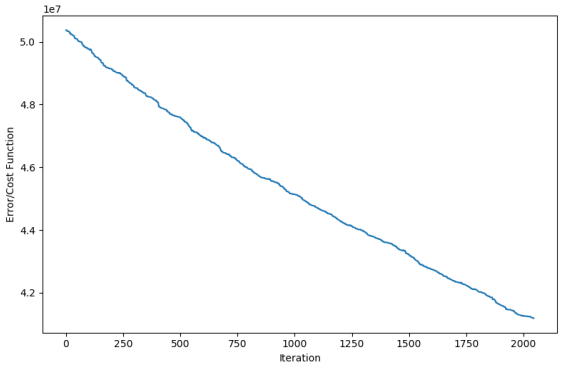

Figure 4. Error vs iterations

Table 1. 1st Approach- Values of Error/Cost Function

Iteration	Error/Cost
1	50371337.6018
2	50357169.2566
3	50358980.1965
..	..
2045	41192266.2838
2046	41190033.7557

Because the number of points plotted is so high, the irregularity of increase isn't as straight forward to observe, but observing the numeric values, it was clear that the error is decreasing but the decrease isn't totally uniform. The error went up at times but in overall pattern, it reduced. To prevent making note of so many values, I adopted batching along with stochastic gradient descent as outlined in the following point.

2. SGD with a 'batch' size of 1000 and recording error after the batch update.

By this I mean, that after 1000 iterations/updates, I am calculating the error and not after every single update. The function to calculate this error after each batch of 1000 randomly picked elements are updated, is calculated using the calc_error() function which does so by going over all data points. This is in contrast to the previous optim_calc_error() function as this function calculates error from the scratch for all data points (in training) rather than simply updating it. Below is the screenshot of my code that calculates error.

```
def calc_error(V, W, X):
    error = 0
    V_norm = LA.norm(V)
    W_norm = LA.norm(W)
    for key, value in X.items():
        i = key[0]
        j = key[1]
        rating = value
        error += np.power(rating - np.dot(V[i], W[j]), 2)

    error += (lambd)*(np.power(V_norm, 2) + np.power(W_norm, 2))
    return error
```

Figure 5. Error vs iterations

I have also included the graph that shows the gradual decrease in the error values and the corresponding table of values. This graph is plotted for 46 iterations, with each iteration making update on 1000 i, j pairs. This plot is more smooth because error is calculated over all the data points but the function takes more time as compared to before.

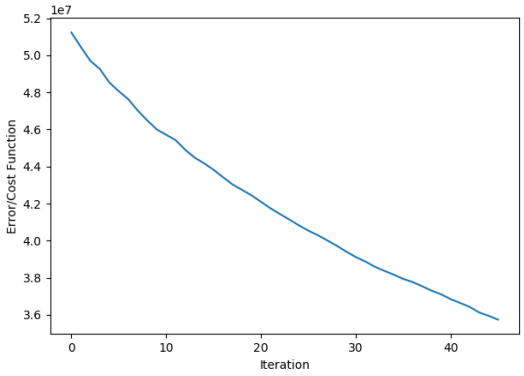

Figure 6. Error vs iterations

Table 2. 2nd Approach- Values of Error/Cost Function

Iteration	Error/Cost
1	51228490.1752
2	50447414.6348
3	49702246.6462
..	..
44	36132157.1923
45	35952827.7325
46	35743210.4142

3.2. Adopted approach

After confirming that my stochastic gradient descent is working as expected by monitoring the error values, I moved on to actually start training my model and learn the values in V and W matrices. This part is covered via matrix_factor_sgd() function. Below is the code snippet for the same.

```
def matrix_factor_sgd(X, rank, lambd):
    V = np.random.rand(u, rank)
    W = np.random.rand(m, rank)
    for epoch in range(epochs):
        print(epoch)
        for key, value in X.items():
            i = key[0]
            j = key[1]
            rating = value
            eij = rating - np.dot(V[i], W[j])

            V_update = V[i] + alpha * (2.0 * eij * W[j] - (lambd*2.0 * V[i]))
            W_update = W[j] + alpha * (2.0 * eij * V[i] - (lambd*2.0 * W[j]))

            V[i] = V_update
            W[j] = W_update
    return V, W
```

Figure 7. Code for matrix_factor_sgd()

Overall training and metric calculation code in the kg2719-hw1.ipynb. Now moving on to the various parameters of the training process.

Hyperparamters:
rank = [1, 8, 16, 32, 64, 128, 256]
lambda = [0.01, 0.02, 0.05, 0.1, 0.2, 0.5]
epochs = 2
alpha = 0.02

'rank' is the number of latent features that we are trying to learn of the interaction between users and movies. 'lambd' is the regularization parameter which controls the degree of regularization we perform during the training process by preventing the values in V and W matrices to obtain really high values, as the objective of the optimization process is to reduce the overall error and lambda multiplied with the sum of L2-norm of the V and W matrices is a positive term that we add to the squared error. We are searching over the aforementioned values of 'rank' and 'lambda' to find the pair of these values which gives the most optimal performance. To assess this, I drew surface plots of the mean RMSE and MRR values vs rank and lambda which are detailed in the next section: Results.

'epochs' is the number of epochs for which I am running my whole training process. Because of the extremely large value of the cost function, letting the optimization run until the error stops decreasing (reached the minima plateau) was not a practical solution, and hence I chose the iteration size to be 2 times the number of points in training set, which comes out to be roughly 27 million. Overall, each of the i, j pair has been updated twice by the gradient descent update rule. The function that performs this task and returns the trained/estimated V and W matrices is matrix_factor_sgd() as explained above.

Constants:
u (number of users)= 138493
m (number of movies) = 27278

Parameters:
V (size: u by rank) and W (size: m by rank) matrices. V and W are randomly initialized using numpy's random.rand(size) function. I have carefully taken into consideration the sizes of these matrices and formulated the update rule, error calculation, RMSE and MRR calculation accordingly. The complete Jupyter iPython notebook file is submitted as part of the zip file submission on courseworks2. Based on the surface plot analysis, only the best of parameters has been written in the notebook. I didn't include the code used to build plots and the section of the code that trained and evaluated RMSE and MRR for various rank and lambda values. Only the final pieces of code which were tested and values that proved to give the best performance have been submitted in the courseworks.

4. Results

This section provides details on all the RMSE and MRR values that I calculated. Ideally, the training process should have been run longer for higher values of rank because that means more number of features to learn, but for the sake of consistency, all other parameters have been fixed and only rank and lambda were varied.

I then plotted surface plots with the main dimension being avg. RMSE (and standard dev. obtained over multiple random folds) and similar plots for MRR with rank varying amongst [1, 8, 16, 32, 64, 128] and lambda varying amongst [0.01, 0.02, 0.05, 0.1, 0.2, 0.5]. Going over the surface plot, below are the findings:

Minimum average RMSE error- 0.947332356
This occurred at rank- 16 and lambda- 0.05

Minimum std. deviaton in average RMSE error- 0.012643843
This occurred at rank- 16 and lambda- 0.05

Maximum average MRR - 0.189902148
This occurred at rank- 8 and lambda- 0.01

Minimum std. deviation in average MRR- 0.002216969
This occurred at rank- 32 and lambda- 0.2

A few observations from the surface plots were that I was able to observe significant variation in the values of average RMSE but not so much for MRR. This could be because we aren't exactly trying to optimize MRR (we optimized the average RMSE) and also that it is a a possibility for the V and W to generate good predicted rankings for the movies which users rated greater than or equal to 3 in test set.

Corresponding surface plots are as below:

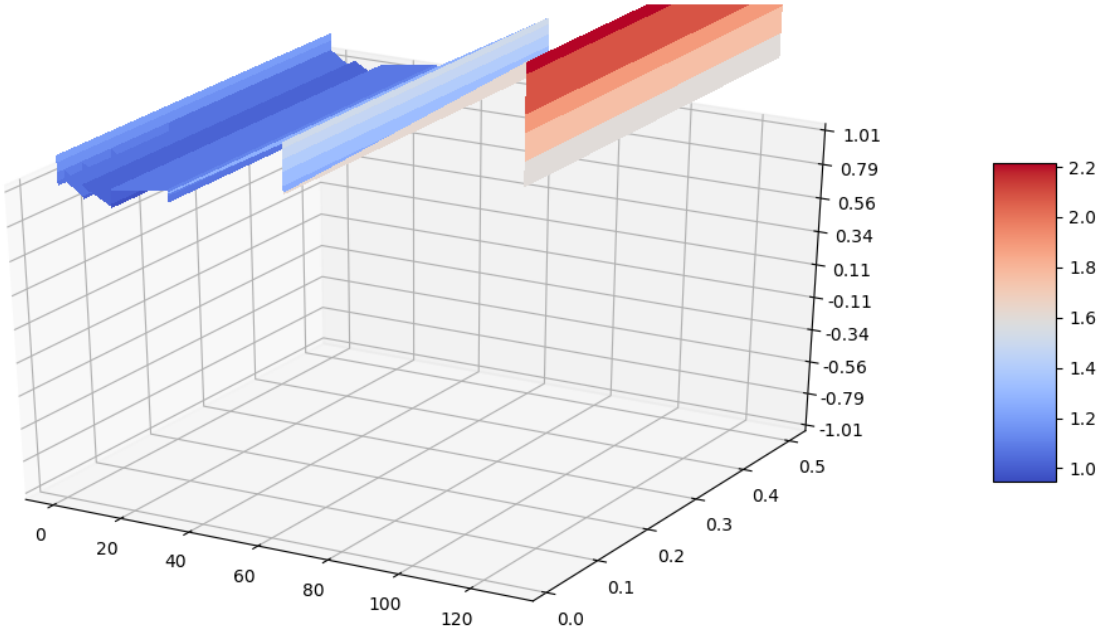

Figure 8. Surface plot of RMSE vs rank, lambda (Min. occurs at 16, 0.05)

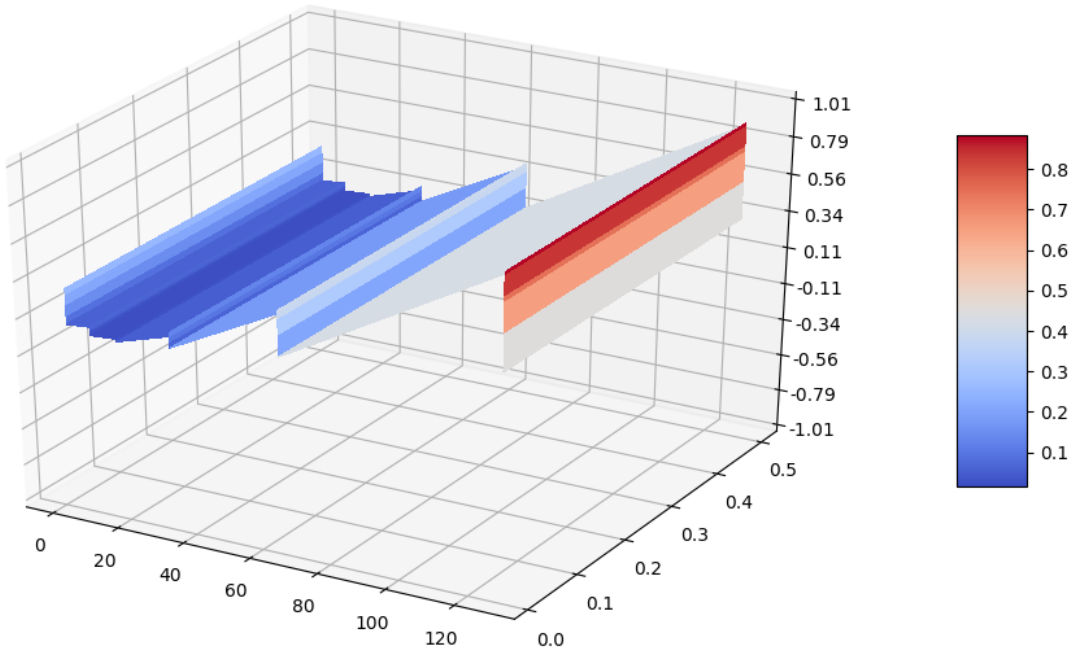

Figure 9. Surface plot of std. dev. in RMSE vs rank, lambda (Min. occurs at 16, 0.05)

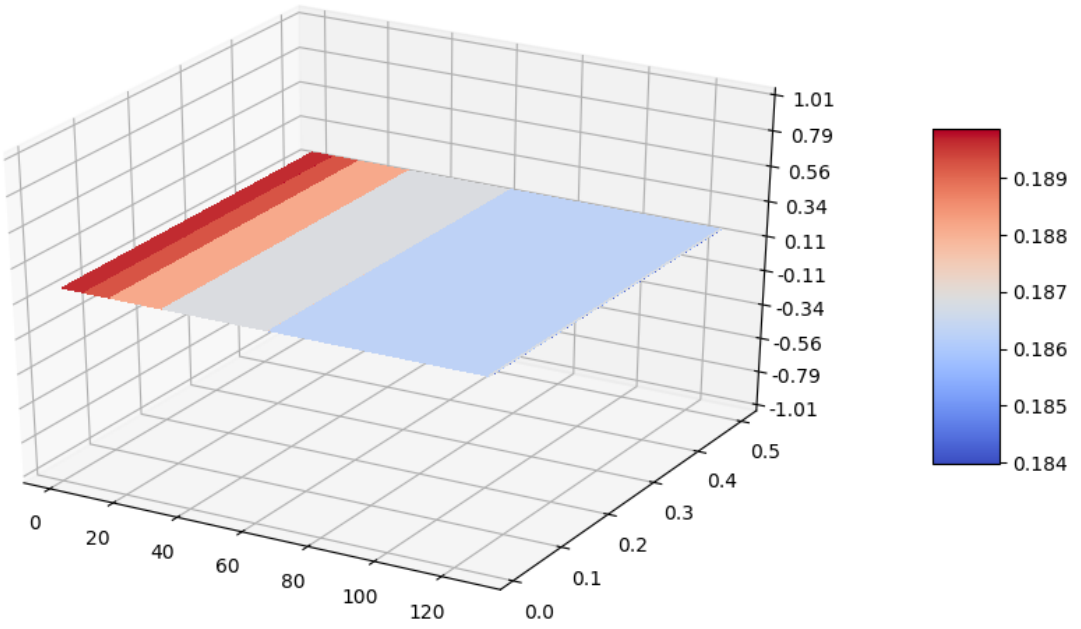

Figure 10. Surface plot of MRR vs rank, lambda (Max. occurs at 8, 0.01)

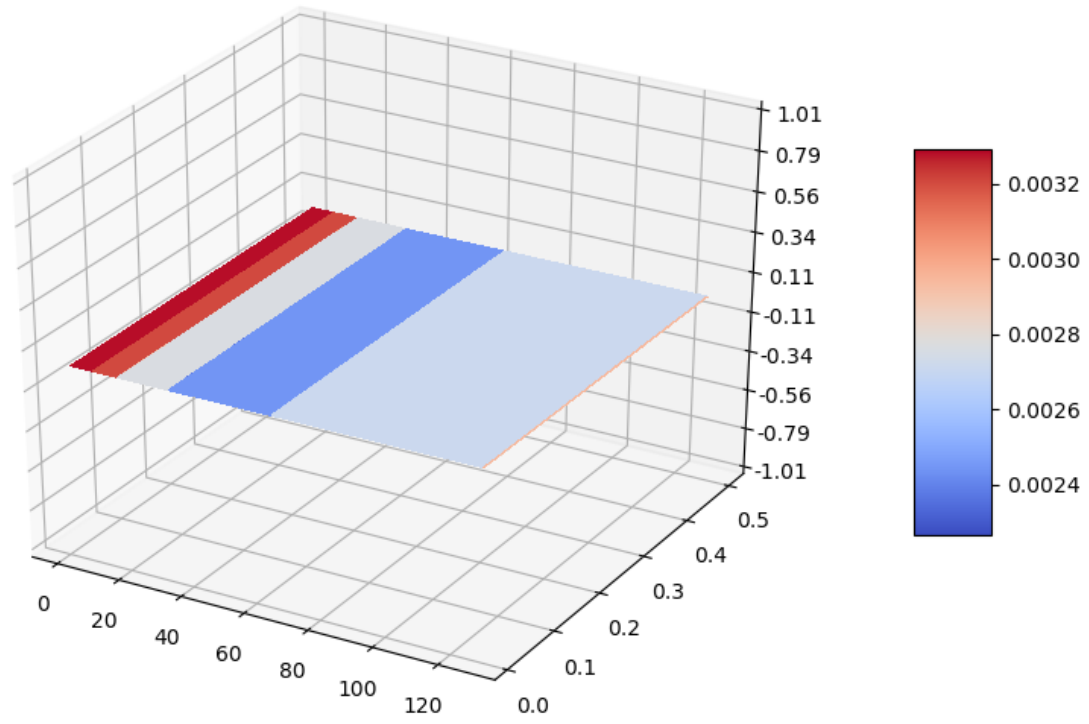

Figure 11. Surface plot of std. dev in MRR vs rank, lambda (Min. occurs at 32, 0.2)

The complete recording of RMSE and MRR for these 36 combinations of rank and lambda values are available in the excel sheet which I have submitted as part of the zip file. A snapshot of the same is below to give an idea of how it looks.

Rank	Lambda		RMSE-f1	RMSE-f2	RMSE-f3	Avg. RMSE	Standard Deviation		MRR-f1	MRR-f2	MRR-f3	Avg. MRR	Standard Deviation
1	0.01		1.019434219	1.598329406	1.02988179	1.215882	0.270464925		0.185651022	0.193227	0.189808	0.189562	0.003097818
1	0.02		1.012174393	1.513624777	1.009815047	1.178538	0.236944039		0.185636407	0.193204	0.189689	0.18951	0.003092205
1	0.05		1.005273478	1.364295551	0.980800308	1.11679	0.175297964		0.185581857	0.19314	0.189542	0.189421	0.003086716
1	0.1		1.006500525	1.211060949	0.964307245	1.060623	0.107761378		0.185547603	0.193151	0.189438	0.189379	0.003104164
1	0.2		1.022682239	1.112724121	0.960876593	1.032094	0.062347726		0.185515313	0.193349	0.189356	0.189407	0.003198267
1	0.5		1.12729729	1.158679564	1.030591107	1.105523	0.054511554		0.18556939	0.193483	0.189328	0.18946	0.00323221
8	0.01		0.938289804	1.084478793	0.946596384	0.989788	0.067042092		0.185751795	0.193966	0.189989	0.189902	0.003354002
8	0.02		0.935740395	1.085888037	0.939833162	0.987154	0.069835594		0.185751178	0.193577	0.190069	0.189799	0.003200641
8	0.05		0.935287384	1.049235701	0.936104483	0.973543	0.053524199		0.185686086	0.193606	0.189854	0.189715	0.003234745
8	0.1		0.943539038	1.037029054	0.941963016	0.974177	0.044447745		0.185612796	0.193418	0.189533	0.189521	0.003186307
8	0.2		0.968623428	0.991835651	0.957299465	0.972586	0.014375085		0.185510298	0.193535	0.189314	0.189453	0.003277495
8	0.5		1.075791163	1.058334704	1.033553591	1.055893	0.017329628		0.185527054	0.193555	0.189322	0.189468	0.003279025
16	0.01		0.930848334	0.992355237	0.936294379	0.953166	0.027800037		0.185005754	0.19256	0.189308	0.188958	0.003093778
16	0.02		0.924280174	0.992547124	0.93491341	0.95058	0.029990901		0.185368367	0.192546	0.189323	0.189079	0.002935208

Figure 12. Overall results, excel sheet snapshot

5. Conclusion

We achieved the best results (least avg. RMSE) with rank= 16 and lambda= 0.05. The estimated V and W matrices were used to calculate these performance metrics on the held out test data. L2 regularization was used. Stochastic Gradient Descent was used with approx. 26 million iterations (2*training_data size, so that each i, j pair is updated during the process) and learning rate was chosen to be 0.02. Python dictionary was used to store the given data set i.e. roughly 13 million ratings that were used during training and roughly 7 million ratings that were used for testing. Surface plots were drawn to compare the performance between various rank and lambda values. Data was split into training and test at the CSV level rather than loading the data first and then splitting. The split was done in such a way so as to ensure that every user exists in both training and test data. The split proportion chosen was 70:30. To achieve robust results as per the multiple random folds suggestion provided in the homework-guidelines, 3 random folds of the dataset were used to create 3 sets of train-test data, and hence at the end, avg RMSE and avg. MRR values were calculated for the 3 values obtained for each setting of the hyperparameter.

6. References

1.Matrix Completion has No Spurious Local Minimum, Rong Ge, Jason D. Lee, Tengyu Ma, January 31, 2017
2.http://www.quuxlabs.com/blog/2010/09/matrix-factorization-a-simple-tutorial-and-implementation-in-python/
3.Matrix Factorization+ for Movie Recommendation, Lili Zhao, Zhongqi Lu, Sinno Jialin Pan, Qiang Yang
4.https://datajobs.com/data-science-repo/Recommender-Systems-[Netflix].pdf