Matrix Completion

Group 8

Group Members: Yefan Li, Jue Li, Yijin Wang, Xiao Ma, Wenxuan Gu, Pengru Lyu, Ziyi Xue, Yanqing Li

1 Introduction

In our final project, we aim to address a matrix completion problem inspired by the Netflix Challenge, known as collaborative filtering. Collaborative filtering involves predicting missing values in a matrix by leveraging user patterns and preferences. Our specific goal is to develop and evaluate five different methods for completing a matrix that represents user preferences for restaurants near our university. By anonymously submitting our favorite restaurant names, we have constructed a matrix with missing values that require accurate predictions.

To aid in our project, we have incorporated two datasets that assist in computing movie ratings. The first dataset was collected from the MovieLens website (movielens.umn.edu) over a seven-month period, ranging from September 19th, 1997, to April 22nd, 1998. This dataset comprises approximately 100,000 ratings on a scale of 1 to 5, provided by 943 users for 1664 movies. Additionally, we utilize a second dataset consisting of a sample of 5000 users from the anonymous rating data collected by the Jester Online Joke Recommender System between April 1999 and May 2003. By employing various machine learning techniques and algorithms, we aim to infer the missing entries in the matrix by leveraging the existing data patterns and similarities between users and restaurants. This project not only enhances our understanding of matrix completion but also contributes to the broader field of recommender systems and personalized recommendations.

2 Methodology

We mainly use 5000 user rating data from Jester Online Joke. When we inspected the data, we noticed that some customers did not provide ratings for certain jokes. To ensure the accuracy of the final testing, we made the decision to exclude those customers from the analysis. The resulting dataset was 1473 × 100. From this dataset, we randomly selected 300 users for use in the testing phase.

Since the rating ranges from -10 to 10 which is pretty large, so that we normalize the rating value and make it range from 0 to 5. We randomly selected 1/3 of the entire data in the total dataset as missing values and set this data as train_data. We then used the remaining complete data table as test_data. We performed analysis and imputation for the missing data using only the train_data.

Finally, we compared the imputed data with the complete data and calculated the mean squared error (MSE) between them.

2.1 Singular Value Decomposition (SVD)

Our first model SVD, which stands for the Singular Value Decomposition, is a widely used matrix factorization technique in linear algebra and numerical analysis. The mechanism is not complicated to understand. Given a rectangular matrix M with dimensions m x n, the SVD factorizes M into three matrices: A m x k matrix U; A k × k diagonal matrix with non-negative entries, which is also known as the singular values; And a k × n matrix V^T. Here T stands for matrix transpose. (See *Figure 2.1.1*)

$$M_{m\times n} = U_{m\times k}\,\Sigma_{k\times k}\,V^{T}_{k\times n}$$

Figure 2.1.1

To apply SVD to our matrix completion problem, we first found the total number of missing values and stored the specific column and row index of missing value. From there, we saw that there are 307 missing values in the original dataset. Then, we calculated the column means of the matrix, and after that did the normalization process: scaled the matrix by subtracting the column means, which we called the "Scaled matrix". We will use it for further prediction. Next, we filled the missing values with the column means to our original dataset to remove all missing values, which we called the "Naive matrix" here.

After that, we put the scaled matrix into the softImpute function with parameter tuning and type = "svd" to get a reconstructed matrix, and then update it by adding back the column means. Finally, we replace the null values with our new predicted values into the original dataset, and at the same time remain the original not-NA values, which is our final predicted SVD matrix. In our final predicted matrix, the original values remain the same, and the null values are replaced by our predicted results.

2.2 Parallel Matrix Factorization (PMF)

The shortened name of Parallel Matrix Factorization is called "PMF". This method was firstly proposed in the paper called "LIBMF: A Library for Parallel Matrix Factorization in Shared-memory Systems" by Chin, Zhuang, et al., 2015.

The authors firstly developed a C++ package called LIBMF. This is a package designed for approximating a matrix with missing value by using the product of two matrices in a latent space. The main idea here is using matrix factorization to decompose the original matrix into the product of two matrices with low dimension which is the similar statement called latent space.

Actually, the method called Parallel Matrix Factorization is mainly for the improvement of accelerated vector operations. It has excellent performance when dealing with the large dataset.

It converts the problem into a Non-Convex Optimization Problem with the following:

$$\min_{P,Q} \sum_{(u,v)\in R} \left[f\left(\boldsymbol{p}_u, \boldsymbol{q}_v; r_{u,v}\right) + \mu_p \left\|\boldsymbol{p}_u\right\|_1 + \mu_q \left\|\boldsymbol{q}_v\right\|_1 + \frac{\lambda_p}{2} \left\|\boldsymbol{p}_u\right\|_2^2 + \frac{\lambda_q}{2} \left\|\boldsymbol{q}_v\right\|_2^2 \right],$$

Figure 2.2.1

2.3 Alternating Least Squares (ALS)

In the context of recommendation systems, the ALS algorithm is used to identify latent factors that describe the preferences of users and the attributes of items. These latent factors are represented as vectors in a low-dimensional space and can capture the underlying characteristics of users and items that are relevant for making recommendations. And the function used in ALS algorithms is:

$$\text{argmin}_{U,V} \textstyle\sum_{m,n|r_{m,n}\neq 0}(r_{m,n} - u_m^T v_n)^2 + \lambda(\sum_m n_{u_m}||u_m||^2 + \sum_n n_{v_m}||v_m||^2)$$

Figure 2.3.1

The ALS algorithm works by alternately fixing one of the two-factor matrices and solving for the other. The algorithm uses least-squares regression to optimize the factor matrices in order to minimize the difference between the original matrix and its factored representation.

There are two main variations of the ALS algorithm: one that minimizes the squared error between the observed ratings and the predicted ratings, and another that minimizes a regularized version of the squared error to avoid overfitting. This process is repeated until convergence is reached. The two functions are:

$$\text{U}_M = (\textstyle\sum_{r_{m,n}\in r_{m*}} V_n V_n^T + \lambda I_k)^{-1} + \sum_{r_{m,n}\in r_{m*}} r_{m,n} V_n$$

$$\text{V}_n = (\textstyle\sum_{r_{m,n}\in r_{*n}} U_m U_m^T + \lambda I_k)^{-1} + \sum_{r_{m,n}\in r_{*n}} r_{m,n} U_m$$

Figure 2.3.2

For our report, the package we used in Python was implicit.

2.4 Non-Negative Matrix Factorization (NMF)

Non-Negative Matrix Factorization(NMF) is a linear algebraic method used for dimensionality reduction and feature extraction. It is widely applied to solve the recommender system problem.

Non-negative Matrix Factorization works by breaking down a single non-negative matrix into the product of two non-negative matrices(Figure 2.4.1). It aims to find two lower-rank factor matrices that, when multiplied together, approximate the original matrix. The method we used here is lee method which want to find the lowest Frobenius Norm, ||V-WH||. It uses iteration to update the matrix W and matrix H(Figure 2.4.2).

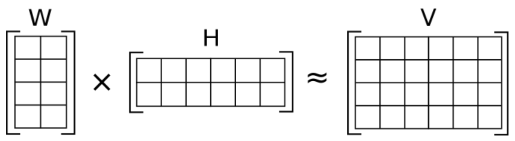


Figure 2.4.1

$$H_{kj} \leftarrow H_{kj}\frac{(W^TV)_{kj}}{(W^TWH)_{kj}}$$

$$W_{ik} \leftarrow W_{ik}\frac{(VH^T)_{ik}}{(WHH^T)_{ik}}$$

Figure 2.4.2

First, we replace all missing values with either the column mean or zero. Then, we iteratively run the NMF algorithm several times to refine the prediction result.

2.5 Stochastic Gradient Descent (SGD)

Stochastic Gradient Descent (SGD) is an iterative optimization algorithm commonly used in machine learning and matrix completion tasks. It is particularly suited for large-scale problems and can speed up convergence by estimating the gradient using a randomly selected subset of observed entries.

To apply SGD to our matrix completion problem, we start with an input matrix R of size 41 × 15 that contains missing values (NAs). Our objective is to minimize the loss function J by iteratively updating the model's parameters. The model makes predictions by factorizing the matrix into two lower-rank matrices: U of size 41 × k and V of size k × 15, where k is the chosen rank. The prediction matrix R of size 41 × 15 is obtained as the dot product of U and V. In each iteration, we randomly select an entry (i, j) from the locations in R that are not NAs. This entry represents a known rating or preference of a user for a particular restaurant. Using this selected entry, we estimate the gradient of the loss function with respect to the model's parameters. The gradient provides the direction and magnitude of the update required to minimize the loss.

$$J = \left(r_{ij} - \vec{U}_i \cdot \vec{V}_j\right)^2 + \lambda\left(\left\|\vec{U}_i\right\|_2^2 + \left\|\vec{V}_j\right\|_2^2\right)$$

Figure 2.5.1 Loss Function J

$$U_{ik} = U_{ik} + \alpha(e_{ij} \cdot V_{kj} - \lambda U_{ik})$$

$$V_{kj} = V_{kj} + \alpha\left(e_{ij} \cdot U_{ik} - \lambda V_{kj}\right)$$

Figure 2.5.2 Updated Position

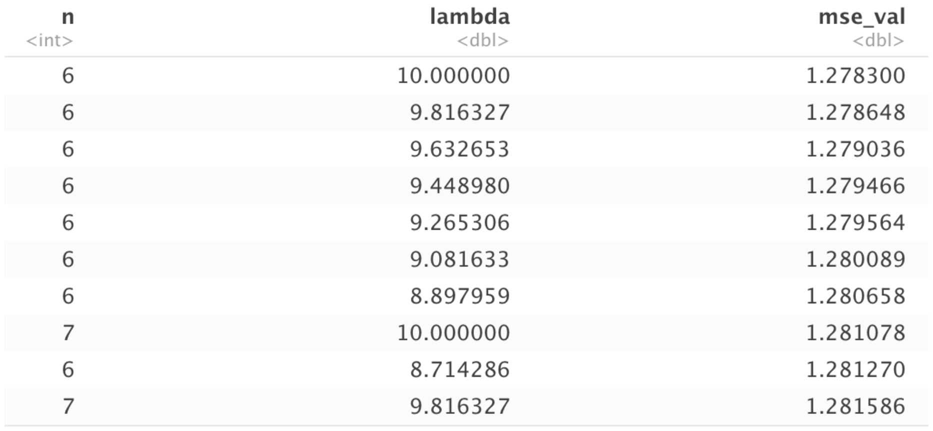
The model's parameters (U and V) are then updated by taking a small step in the direction of the negative gradient, multiplied by a learning rate. This update process is repeated for each iteration, gradually refining the model's parameters and improving the accuracy of predicted ratings. By iteratively applying SGD, the algorithm converges towards a solution that minimizes the loss function and provides accurate predictions for the missing entries in the matrix. SGD's stochastic nature allows it to handle missing data effectively and converge faster than traditional gradient descent methods.

3 Result

3.1 Singular Value Decomposition (SVD)

For the result part, we calculated the MSE using the sum of the squared differences between the predicted matrix and naïve matrix - which gives exactly the error of all missing values, and then divide by the number of null values.

We used the 300 x 100 processed sample dataset from Jester Online Joke to find the most appropriate parameters. We run the model with this randomly chosen sample dataset repeatedly with different rank.max and lambda in the softImpute function to accomplish the parameter tuning process. Each time, we stored the corresponding MSE result. (See *Figure 3.1.1* for detailed results)



n	lambda	mse_val
<int>	<dbl>	<dbl>
6	10.000000	1.278300
6	9.816327	1.278648
6	9.632653	1.279036
6	9.448980	1.279466
6	9.265306	1.279564
6	9.081633	1.280089
6	8.897959	1.280658
7	10.000000	1.281078
6	8.714286	1.281270
7	9.816327	1.281586

Figure 3.1.1

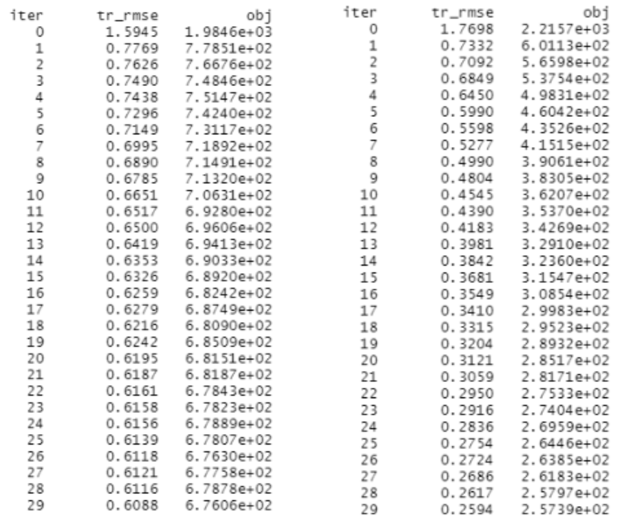
During the parameter tuning, we learned that increasing one or both rank and lambda could lead to lower MSE results. The rank.max parameter in the softImpute function (which is also the k in *Figure 2.1.1*) controls the maximum rank of the low-rank approximation. Increasing the rank.max may improve the accuracy of the approximation but may also increase the computational time required. On the other hand, the lambda parameter controls the amount of regularization applied to the low-rank approximation. Regularization helps to prevent overfitting and can improve the accuracy of the method.

Finally, we put back the parameters that performed the best in sample data, rank.max = 6 and lambda = 10, to our feedback data. The MSE is around 0.0037 (0.003699374), which is also the best result for the feedback dataset with SVD model.

3.2 Parallel Matrix Factorization (PMF)

We used the following way to test the performance of the method of Parallel Matrix Factorization. For the package selection, the LIBMF package has a R version called Recosystem. We used the Recosystem package in R to test the performance. For the dataset we used, we selected the original dataset called Feedback.csv.

We first trained the model and then we used the tuned model to make the prediction. So, we had two attempts here. First of all, as the Naive Mean Guess, we filled all missing values with the colmeans as a basic dataset. For the first attempts, we trained the model with the basic dataset. For the second attempt, we used the predicted values that we gained from first attempts to train the model again to get the final results. We set the iteration to 30 for both attempts. For these two attempts, we get the following results:



```
iter      tr_rmse          obj
   0       1.5945   1.9846e+03
   1       0.7769   7.7851e+02
   2       0.7626   7.6676e+02
   3       0.7490   7.4846e+02
   4       0.7438   7.5147e+02
   5       0.7296   7.4240e+02
   6       0.7149   7.3117e+02
   7       0.6995   7.1892e+02
   8       0.6890   7.1491e+02
   9       0.6785   7.1320e+02
  10       0.6651   7.0631e+02
  11       0.6517   6.9280e+02
  12       0.6500   6.9606e+02
  13       0.6419   6.9413e+02
  14       0.6353   6.9033e+02
  15       0.6326   6.8920e+02
  16       0.6259   6.8242e+02
  17       0.6279   6.8749e+02
  18       0.6216   6.8090e+02
  19       0.6242   6.8509e+02
  20       0.6195   6.8151e+02
  21       0.6187   6.8187e+02
  22       0.6161   6.7843e+02
  23       0.6158   6.7823e+02
  24       0.6156   6.7889e+02
  25       0.6139   6.7807e+02
  26       0.6118   6.7630e+02
  27       0.6121   6.7758e+02
  28       0.6116   6.7878e+02
  29       0.6088   6.7606e+02
```

```
iter      tr_rmse          obj
   0       1.7698   2.2157e+03
   1       0.7332   6.0113e+02
   2       0.7092   5.6598e+02
   3       0.6849   5.3754e+02
   4       0.6450   4.9831e+02
   5       0.5990   4.6042e+02
   6       0.5598   4.3526e+02
   7       0.5277   4.1515e+02
   8       0.4990   3.9061e+02
   9       0.4804   3.8305e+02
  10       0.4545   3.6207e+02
  11       0.4390   3.5370e+02
  12       0.4183   3.4269e+02
  13       0.3981   3.2910e+02
  14       0.3842   3.2360e+02
  15       0.3681   3.1547e+02
  16       0.3549   3.0854e+02
  17       0.3410   2.9983e+02
  18       0.3315   2.9523e+02
  19       0.3204   2.8932e+02
  20       0.3121   2.8517e+02
  21       0.3059   2.8171e+02
  22       0.2950   2.7533e+02
  23       0.2916   2.7404e+02
  24       0.2836   2.6959e+02
  25       0.2754   2.6446e+02
  26       0.2724   2.6385e+02
  27       0.2686   2.6183e+02
  28       0.2617   2.5797e+02
  29       0.2594   2.5739e+02
```

Figure 3.2.1 First and Second Attempts

We could see that the RMSE kept decreasing and there was no obvious overfitting problem. These are pretty satisfactory results.

Then, we calculated the MSE between the Naive Mean Guess and the first attempt. Also, we calculated the MSE between the first attempt and the second attempt. The results showed that MSE between First Attempt and Naive Mean Guess is: 0.06829268. Then, the MSE between

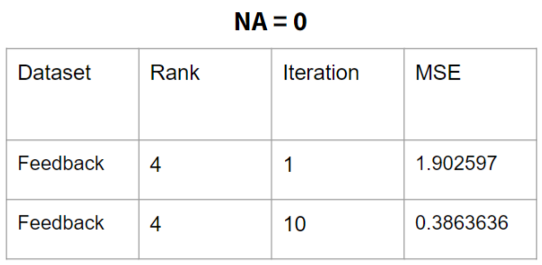
First Attempt and Second Attempt is: 0.04878049. In this way of comparison, we roughly get the results that the MSE decreases 0.02276422.

3.3 Alternating Least Squares (ALS)

To test our ALS method, we applied the sample dataset and randomly selected 1/3 of the total number of values as missing values. After using the ALS method, we compared the imputed data of the missing value and the original data by estimating the MSE. We could observe that the MSE ranged from 1.78 to 1.86, by turning the rank and lambda. And the min MSE is 1.77 with rank equal to 4, and lambda equal to 1.

Then, we applied the ALS method to impute the missing value for the feedback dataset. We firstly compared the MSE which is calculated by the reconstructed values and the existing values. We observed the MSE is from 0.86 to 1.45, by turning the rank from 2 to 14 and lambda from 0 to 10. And also, we compared the imputed value and the naive guess, which is the overall mean. The MSE is from 0 to 0.06, by turning the same parameters. In this situation, as the MSE is close to 0, the imputed values with ALS methods are close to 0. To avoid the problems of overfitting, we prefer to choose low rank and high lambda.

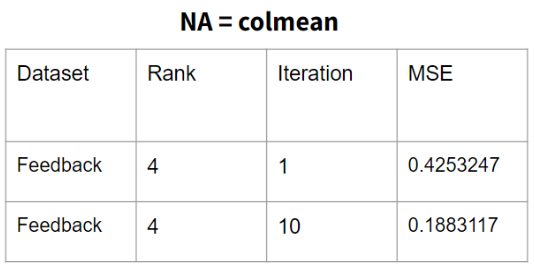
3.4 Non-Negative Matrix Factorization (NMF)



NA = 0

Dataset	Rank	Iteration	MSE
Feedback	4	1	1.902597
Feedback	4	10	0.3863636

Figure 3.4.1 NA=0



NA = colmean

Dataset	Rank	Iteration	MSE
Feedback	4	1	0.4253247
Feedback	4	10	0.1883117

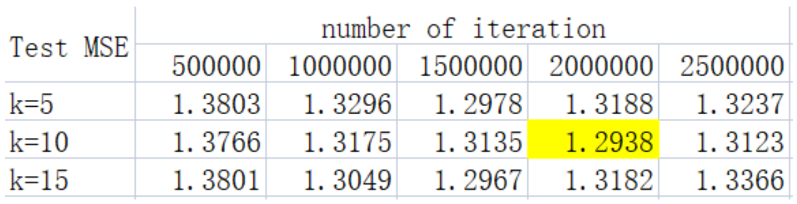
Figure 3.4.2 NA=colmean

The testing method we use in the NMF approach involves calculating the Mean Squared Error (MSE) between the original matrix (excluding NA parts) and the prediction matrix. Our findings indicate that replacing NA values with the column mean is generally more effective than replacing them with 0.

However, in several datasets we tested, we found that replacing NA values with 0 works better in boolean datasets (0 or 1). For other datasets, including Movielens and Jester, replacing NA values with the column mean yields better results. After parameter tuning in the Jester dataset, we achieved the best result: an MSE of 1.3939 with a rank of 4.

3.5 Stochastic Gradient Descent (SGD)

Based on the test dataset we chose, we also applied the SGD algorithm to get the test MSE with different parameters (rank “k”) and numbers of iterations. A brief summary of the results are shown in the following figure.



Test MSE	number of iteration				
	500000	1000000	1500000	2000000	2500000
k=5	1. 3803	1. 3296	1. 2978	1. 3188	1. 3237
k=10	1. 3766	1. 3175	1. 3135	1. 2938	1. 3123
k=15	1. 3801	1. 3049	1. 2967	1. 3182	1. 3366

Figure 3.5.1

We set rank k = 5, 10, 15 seperately to check the effect that the value of the parameter has on the result with α = 0.01 and λ = 0.1. As the result in the table shows, the minimum test MSE of different k are all very close to 1.29, however, the number of iterations which cause the overfit situation is different. When k = 10, the test MSE becomes higher between 2,000,000 and 2,500,000 iterations, but when k = 5 or 15, the test MSE becomes higher between 1,500,000 and 2,000,000 iterations. Besides, we think the randomness of the SGD implement process may also cause this difference.

4 Difficulties & Conclusion

4.1 Conclusion

After using four algorithm tests, this is the minimum mean square error value we get, and their range is from 1.27 to 1.78, the minimum mean square error value is from the SVD algorithm test, whose parameters are rank.max = 6, and lambda = 10.

Method	Min MSE for the sample dataset
SVD	1.2783
ALS	1.7767
NMF	1.3939
SGD	1.2938

4.2 Difficulties

When we used the SVD algorithm test, we noticed that the SoftImpute() function requires a fixed range of parameter values for "rankmax," with the largest value being one less than the number of columns. Additionally, other parameter values must be positive. Since our goal was to impute missing values of feedback, our group examined the values of this file and set "rankmax" to 14. We then applied this range to Jester's dataset to measure the results. Next, we expanded the "rankmax" parameter range for calculation and found a smaller MSE value. However, since the parameter corresponding to the smaller MSE could not be used for auxiliary calculation in feedback, our team ultimately concluded that "rankmax" should be set to 6 and "lambda" should be set to 10.

Also, for SVD algorithm test and ALS algorithm test, we observe that, when we use a larger parameter, the mean square error value would be much closer to their column mean, it means that sometimes it might bring an overfit problem.

During our project, we encountered challenges related to debugging, parameter tuning, and incorporating the mathematical aspects of stochasticity into the SGD algorithm. To address debugging issues, we systematically reviewed our code, verified mathematical formulas, and tested with small-scale datasets. This helped us identify and fix any syntax errors, logic problems, or inconsistencies in our implementation. Parameter tuning proved challenging initially. To overcome this, we conducted multiple experiments with different parameter combinations and evaluated their impact on performance. Through trial and error, we refined the parameters based on observed results, aiming for optimal prediction accuracy and convergence speed. Incorporating the mathematical concept of stochasticity into SGD required a deeper understanding. We studied the mathematical formulation and its application in matrix completion. We grasped the stochastic gradient estimation process and random entry selection. Initially, we did not incorporate stochasticity into the process, but after realizing our oversight, we made the necessary adjustments by randomly sampling a pair of i and j in each iteration to ensure the algorithm followed the intended stochastic gradient descent approach.

Since SVD has the overfit problem mentioned above, even though it performed the best for the Jester's dataset, we finally decided to use SGD method to complete the “Feedback” dataset. Considering the size of the dataset is smaller than the test dataset we previously used, so we set k = 5 and implement 200,000 iterations. Besides, for the data which we predicted slightly larger than 5, we scale it’s value to 5, and for the data which we predicted slightly smaller than 1, we scale it’s value to 1.

Citation

Chin, W.-S., Yuan, B.-W., Yang, M.-Y., Zhuang, Y., Juan, Y.-C., & Lin, C.-J. (2016). LIBMF: A Library for Parallel Matrix Factorization in Shared-memory Systems. Journal of Machine Learning Research.

Figure 2.4.1: Wikimedia Foundation. (2023, May 4). *Non-negative matrix factorization*. Wikipedia. https://en.wikipedia.org/wiki/Non-negative_matrix_factorization

Figure 2.4.2: An introduction to NMF package - NMF.R-forge.r-project.org. (n.d.). https://nmf.r-forge.r-project.org/vignettes/NMF-vignette.pdf

Appendix

Code:
https://drive.google.com/drive/folders/1yqAUtrXZDGkuUMFGM9sB4YfpbBv78vZa?usp=sharing

Contribution

SVD	Yijin Wang, Yefan Li, Ziyi Xue, Jue Li
PMF	Xiao Ma
ALS	Ziyi Xue, Jue Li
NMF	Wenxuan Gu
SGD	Yanqing Li, Pengru Lyu