Recommender System using Collaborative Filtering
Final Report

Ibtehal Mahmud, Mauricio Soroco, Yimeng Liu

November 2022

Contents

1 Background

Collaborative Filtering is a type of recommendation system/algorithm that gives users recommendations based on past behaviors and it comes in two types; model based and memory based. In this project we will be focused on the model-based approach. The model-based approach is simply a method where we use past data of the user and other users that are similar to recommend a product/service.

Collaborative filtering has many applications in systems that automatically predict a user's preferences for items such as movie, book, and venue recommender systems. Collaborative filtering was famously used to solve the Netflix problem, where a prize of 1 million dollars was offered for improving on Netflix's recommendation system.

2 Problem Statement

This project aims to compare two algorithms that can suggest items to a user based on given preferences (both the user's and others similar to them). We will be using matrix factorization via gradient descent to solve our problem. We will also explore a method involving minimizing the nuclear norm of the prediction matrix.

2.1 Example

Consider the matrix—call it X—shown in Figure 1 of user ratings for items.

	Item 1	Item 2	Item 3	Item 4	Item 5
User 1	?	?	2	2	?
User 2	3	4	?	1	?
User 3	3	4	2	?	?
User 4	?	?	4	4	1
User 5	?	1	?	?	3

Figure 1: Example data matrix X. Entries with "?" are the values we want to predict. We can make use of entries of other users or other items that are similar.

Assume that we are trying to predict the highlighted entry to determine whether user 3 would like item 4. Firstly we remark that user 2 and user 3 tend to have similar ratings for the same items. Therefore it would be logical to assume that user 3 would give the same rating for item 4, namely a rating of 1. Alternatively we also remark that item 3 and item 4 tend to get similar ratings from different users. Therefore we could expect that item 4 would receive give the same rating as item 3, namely a rating of 2. Notice that in a sense we are trying to make the columns and rows linearly dependent when possible for similar ratings.

This way we are trying to minimize the rank of our matrix of predictions, R, to approximate X.

3 Mathematical Theory

In the Netflix problem, users submit their ratings for a subset of items. The goal is to infer missing ratings using ratings for similar items and from similar users. Let X be the matrix that holds the user/item data (with some empty entries we want to predict). Because of this, we consider X to be composed of a few latent factors per user, and similarly for items. We would like to use these latent factors to predict the remaining entries. Therefore in both methods explored, the performance is increased when every row and every column of X contain at least one entry. If a column j of X is missing, then there is no information for how to set $(W^T)_j$, since we cannot compute any error for any approximation. Likewise if a row i of X is missing, then there is no information for how to set Z_i.
We begin with an explanation of the Matrix Factorization method.

3.1 Matrix Factorization

Matrix factorization involves finding a representation for a matrix as a product of two other matrices.

Let X be the matrix that holds the user/item data (with some empty entries we want to predict). Let Z be the matrix that holds the latent features corresponding to users Let W be the matrix that holds the latent features corresponding to the items. Define R to be our prediction matrix. We wish to find Z and W such that:

$$R := ZW \approx X$$

. Thus we can construct a loss function over the available ratings Ω:

$$f(Z, W) = \sum_{(i,j)\in\Omega} \left(z_i w_j^T - x_{ij}\right)^2 \tag{1}$$

Minimizing this objective function should give us a matrix whose entries match X as closely as possible. The remaining entries are thus the predictions.
Objective function in the form above are commonly known as minimizing squared errors or a "least squares" problem.
One possible issue with the objective presented in 1 is that the resulting Z and W may have very large or complicated entries. While the objective function encourages the results of ZW to match existing x_{ij}s, there is no other guidance for the rest of ZW. Thus large or complicated entries may lead to an increase in poor predictions. In mathematical modeling, this is known as overfitting. This means that the predictions are trying to match the data so closely, that they also fit to minor details that do not reflect the general trend. To avoid this, we can add regularization to penalize really large W and Z, and therefore favouring a more simple model.

Regularization, generally, is the process of changing the result into one that it simpler, particularly to avoid overfitting the data. In this case, we are adding the values of matrix Z and W to the objective function in the hopes of biasing them towards smaller values.

$$f(Z,W) = \sum_{(i,j)\in\Omega} \left(z_i w_j^T - x_{ij}\right)^2 + \lambda(\|Z\|_F^2 + \|W\|_F^2) \tag{2}$$

In 2, $\|W\|_F$ denotes the Frobenius norm of W defined $\|W\|_F = \sum\sum w_{ij}^2$ (the sum over all element-wise squares of W). The λs are chosen to increase or decrease the penalty for W and Z. The larger the λ, the more that Z and W need to be smaller to decrease the objective function.

As it stands, our objective function assumes that the mean of the ratings is 0. In other words, if the data matrix X is particularly sparse such that there is not enough information to construct a row in Z or a column in W, then these will be minimized towards 0 by the objective function. Instead, it would be more reasonable to predict the average rating when there is not enough information to predict otherwise. Additionally, by adding a bias term, we are increasing the expressiveness of our model, and so we expect it to fit better to the data. Using this logic, we update our objective function to include the overall average prediction.

$$f(Z,W) = \sum_{(i,j)\in\Omega} \left(z_i w_j^T + \beta - x_{ij}\right)^2 + \lambda(\|Z\|_F^2 + \|W\|_F^2) \tag{3}$$

This also means that our model becomes:

$$ZW + \beta \approx X \tag{4}$$

We can obtain β by taking the average of all ratings in X.

To minimize the objective function in 3 we will be using an iterative algorithm known as Gradient Descent (GD).

3.1.1 Gradient Descent Algorithm

We fist begin by explaining the basic concepts of gradient descent. Suppose we have an objective function $f(w)$ for which we wish to find w that minimizes the function. The algorithm is as follows:

```
1 function Gradient_Descent(f, w, ϵ, maxIterations, α):
2     ▶ w is an initial guess
3     ▶ f is the objective function
4     ▶ ϵ stop if the gradient gets below this
5     ▶ maxIterations is the max number of iterations
6     ▶ α is the starting step size
7
8
```

```
9       for i in 1:maxIterations
10          w_{t+1} = w_t − α ∗ ∇f(w)
11
12          while f(w_{t+1}) > f(w_t)      ▶ Decrease step-size if we increased the function
13              α = α/2
14              w_{t+1} = w_t − α ∗ ∇f(w)
15          end while
16          w = w_{t+1}
17
18          if ‖∇f(w)‖ < ϵ                 ▶ stop if the gradient is really small
19              return w
20          end if
21      end for
22      return w                           ▶ Reached maximum number of iterations
23  end function
```

We start at a given w. We know that the gradient of a function points in the direction of steepest ascent, and so the negative of the gradient will point in the direction that minimizes the objective function the fastest. Thus we use the negative gradient and our current location w to pick a w_{t+1} that is now a step closer to the minimum argument of the objective function. The size of the step we take is given by α. If α is too big, we risk overshooting the minimum. Therefore before we update w to be w_{t+1}, we check that this step decreases the value of the objective function (lines 12). If it doesn't, we pick a closer w_{t+1} until we are sure that we are not overshooting the minimum (lines 12-14). If f is convex, then gradient descent will converge to the global minimum.

In our case, as per equation 3 we are trying to minimize both W and Z. Therefore, for each update of w to be w_{t+1} in the code in the we will instead update the matrices to be W_{t+1} and Z_{t+1}.

3.1.2 Objective Function Gradient

The gradients of our objective function in 3 are given by:

$$\frac{\partial}{\partial z_{ab}} f(Z,W) = \sum_{(i,j)\in\Omega} \left[2 \cdot (z_a w_j^T + \beta - x_{aj}) w_{bj}\right] + 2\lambda z_{ab} \tag{5}$$

$$\frac{\partial}{\partial w_{ab}} f(Z,W) = \sum_{(i,j)\in\Omega} \left[2 \cdot (z_i w_b^T + \beta - x_{ib}) z_{ia}\right] + 2\lambda w_{ab} \tag{6}$$

3.2 Matrix Completion by Convex Optimization

Given that we want to recover a matrix X of size $n \times d$ but only observe m number of entries in the matrix where m is much smaller than nd. As stated in section 2.1, the matrix X that we are recovering is said to be a low rank matrix.
If we have a enough measurements (at least one entry per row and one per column), then we can solve the following optimization problem

$$\begin{aligned} &\text{minimize rank(R)} \\ &\text{subject to: } R_{ij} = X_{ij} \quad \forall (i,j) \in \Omega \end{aligned} \tag{7}$$

From the results of the work by Candes and Recht in their paper "Exact Matrix Completion via Convex Optimization", this is equivalent to minimizing the sum of singular values over a constrained set.

$$\begin{gathered} \text{minimize } \|R\|_* \\ \text{subject to } R_{ij} = X_{ij}, (i,j) \in \Omega \\ \text{where } \|R\|_* = \textstyle\sum_{k=1}^{\in} \sigma_k \text{ is the convex relaxation of the Rank} \\ \text{and } \sigma_k \text{ are the singular values of } \|R\|_* \end{gathered}$$

This sum is known as the Nuclear Norm. Directly minimizing the rank problem is an np-hard problem. Instead, we can minimize the convex envelope, which turns out to be the Nuclear Norm. The Nuclear Norm is a convex relaxation of the rank function and as such minimizing it gives us a result that will closely match the result from minimizing the rank function. We use the Nuclear Norm as the rank function is not convex, where the Nuclear norm is (it is simply the sum of the singular values). The rank represents the number of non-zero singular values, whereas the Nuclear norm is the sum of the singular values.

The Nuclear norm method can be seen as extension of the trace heuristic that is often used by the control community. We can minimize the trace function when we have a positive definite symmetric matrix, then the eigen values are the same as the of the SVDs, therefore the trace is the sum of these eigen values, which is the same as the sum of SVDs. We can now formulate the nuclear norm in terms of semi definite programming (where our optimization variables are Y, W_1, W_2

$$\begin{gathered} \text{minimize } \{\text{Trace}(Y) = \text{Trace}(W_1) + \text{Trace}(W_2)\} \\ \text{subject to: } R_{ij} = X_{ij} \quad \forall (i,j) \in \Omega \\ Y = \begin{bmatrix} W_1 & R \\ R^T & W_2 \end{bmatrix} \succeq 0 \end{gathered}$$

as minimizing the trace is similar to minimizing the nuclear norm. The sum of all non-negative eigenvalues (of a particular matrix Y) is the nuclear norm of matrix Y. Even for the general matrices which may or may not be positive definite or symmetric, we can still

create a similar equation as written above.

The rank function counts the total number of nonvanishing singular values whereas the nuclear norm method sums up the amplitudes of these singular values. The nuclear norm is a convex function and can easily be optimized using semidefinite programming. Minimizing the nuclear norm also leads to low rank matrices. The nuclear norm is always equal to the L1 norm of the singular values as they are always positive. From compressed sensing, we know that if we minimize the L1 norm of a vector leads to sparse solutions. A sparse set of singular values means that most of them are 0, which means that the desired matrix is low rank.

3.2.1 The Use of CVXPY

To implement this method into Python to predict the matrix, we chose to use CVXPY, a domain-specific language for convex optimization embedded in Python. It allows people to express the problem in a natural way and automatically transforms it into standard form, calls a solver, and hence gives the results.

In our matrix completion problem, since the nuclear norm can be thought of as a convex relaxation of the number of non-zero eigenvalues, we try to minimize the nuclear norm of the matrix instead of minimizing the rank, as explained above. If the matrix variable is symmetric and positive semidefinite, the nuclear norm of the matrix is the sum of the (non-negative) eigenvalues, which equals the trace of the matrix, hence we transformed the problem into a semidefinite program, which is a subfield of convex optimization. A semidefinite program (SDP) is an optimization problem of the form:

$$\textbf{minimize } tr(CX)$$

$$\textbf{subject to } tr(A_iX) = b_i, i = 1, \ldots, p$$

$$X \succeq 0,$$

where tr is the trace function, $X \in \mathbf{S^n}$ is the optimization variable, $C, A_1, \ldots, A_p \in \mathbf{S^n}$, and $b_1, \ldots, b_p \in \mathbf{R}$ are problem data, and $X \succeq 0$ is a matrix inequality. Here $\mathbf{S^n}$ denotes the set of n-by-n symmetric matrices.
However, in our problem, the general matrix $\mathbf{X}$ may not be positive definite or even symmetric, thus we formulated a new semidefinite program to ensure it's symmetry as well as positive definite.

$$\text{minimize } \{\text{Trace}(Y) = \text{Trace}(W_1) + \text{Trace}(W_2)\}$$

$$\text{subject to: } R_{ij} = X_{ij} \quad \forall (i,j) \in \Omega$$

$$Y = \begin{bmatrix} W_1 & R \\ R^T & W_2 \end{bmatrix} \succeq 0$$

Therefore, we can finally implement this problem with CVXPY and solve the prediction matrix.

4 Data and Computations

Please see attached Python code.

4.1 Data

For this project, we decided to create the data so that we could obtain the true answers for the predictions. Obviously, we created the data to follow the expected behaviour of ratings collected from users for real-world items.

Firstly, we know that similar users should give similar ratings for the same items. Likewise similar items are expected to receive the similar ratings from the same users. Thus to create our fully filled matrix of ratings, Y, we generate two random matrices $\tilde{Z}$ and $\tilde{W}$ of size $n \times \tilde{k}$ and $\tilde{k} \times d$ (just like our expected models) such that $Y = \tilde{Z}\tilde{W}$. $\tilde{k}$ was set to 5 (this information would not be present in real-world datasets). Because ratings can vary slightly (due to humans seldomly having identical preferences) we introduced noise to approximately 10 % of the data, following a normal distribution, $\mathcal{N}(0, \frac{\sigma_Y}{10})$, where σ_Y is the standard deviation of Y. We did this step ensuring that the maximum rating was never exceeded (and similarly for the minimum rating).

The matrix of ratings X used to construct predictions was created by randomly sampling 60% of the ratings of Y. In the attached code we referred to this as the training set. Another 20% of the data was used to construct the validation set which was used to determine the best k and the best λ to use in the matrix factorization models. Finally, the last 20% of Y was the test set which was used to evaluate the final performance of each method. In other words, the test set is what we expect the predictions to be. If the model overfits on the training set (for example by modelling closely the noise we introduced), we expect the test set to have large errors, and similarly if the model doesn't well represent the training data, we expect it to do poorly on the test set.

Each run of the models was generated with new data. For this reason performance varied and therefore we compared multiple runs.

5 Results and Discussion

We ran each method for 40 different generated datasets of size 50 users $\times$ 30 items (see Figure 2). We see that the most basic Least Squares model performs the worst. This is expected for

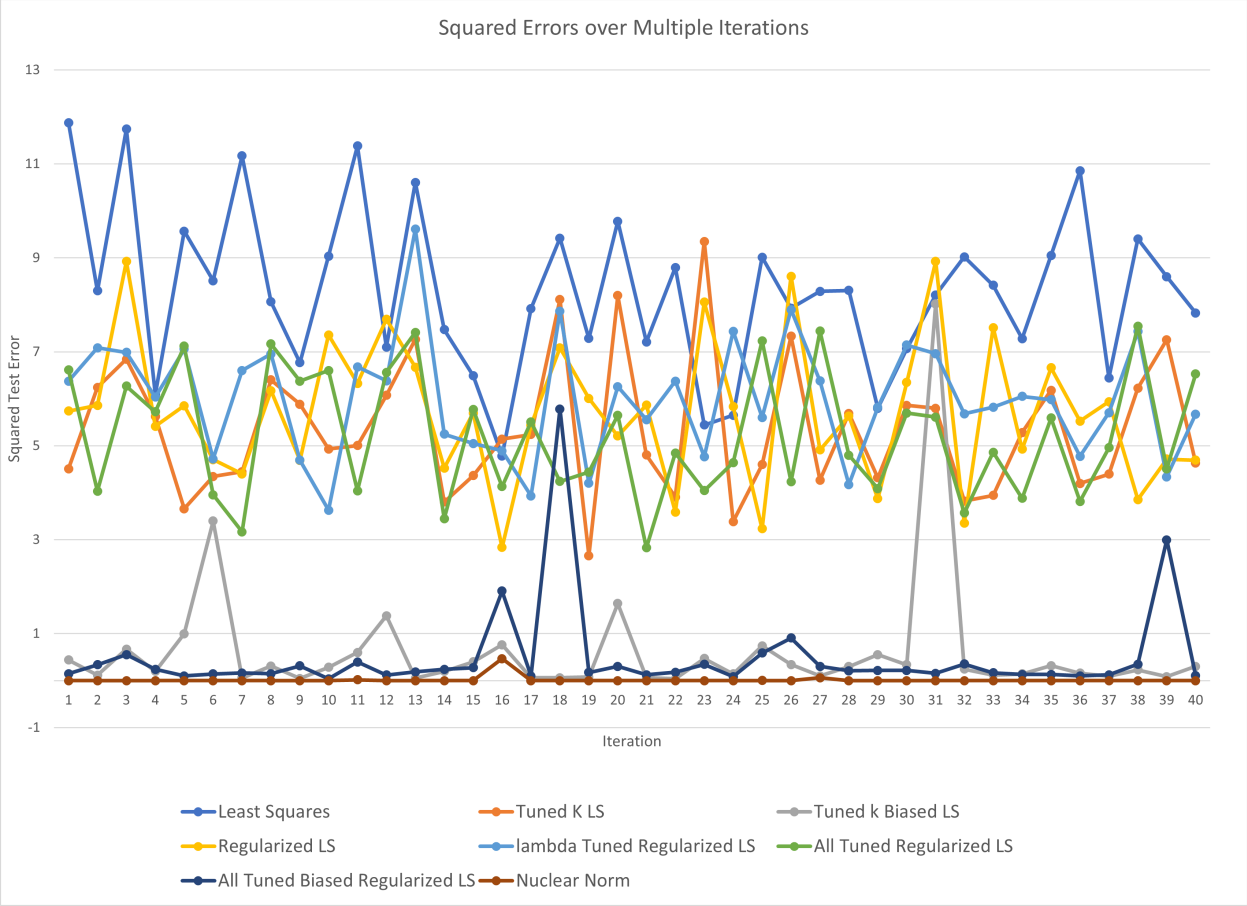

Figure 2: Line plot of the squared test error of each model over different instances of data.

three reasons. Firstly, this model that had no disincentive to overfit on the training dataset, secondly, k was arbitrarily set to 3, so there is no guarantee this is the best k, and finally, it doesn't have the same expressiveness as the models with a bias term. As soon as we tuned k, we started getting some improvements. Recall that our true $\tilde{k} = 5$, and so we expected that as k approaches $\tilde{k}$, the errors would go down. The k that was chosen had a mean of 5.9 and ranged from 4 to 7 among the 40 iterations. In theory with $k = \tilde{k}$, the model should already be able to obtain low errors, so the fact that k values greater than $\tilde{k}$ were chosen could be attributed to the fact that the model overfit on the noise. Nonetheless, tuning k increased performance compared to every model that didn't tune k. When we added a bias term to this model, the errors decreased considerably. Interestingly, the chosen k now had a mean of 6.325 and ranged from 6 to 7. When we compare regularized Least Squares to the most simple Least Squares model, we see that regularization improved the test error. This is because regularization discourages overfitting on the training dataset by penalizing the squared values of the elements in Z and W. When we started to tune the amount of

regularization, λ, we see that the errors went up slightly but are very similar to the former method. The λs ranged from 1 to 9, but these would depend on the values of the data matrix. It appears that tuning both λ and k gave a better model than with an arbitrary k,

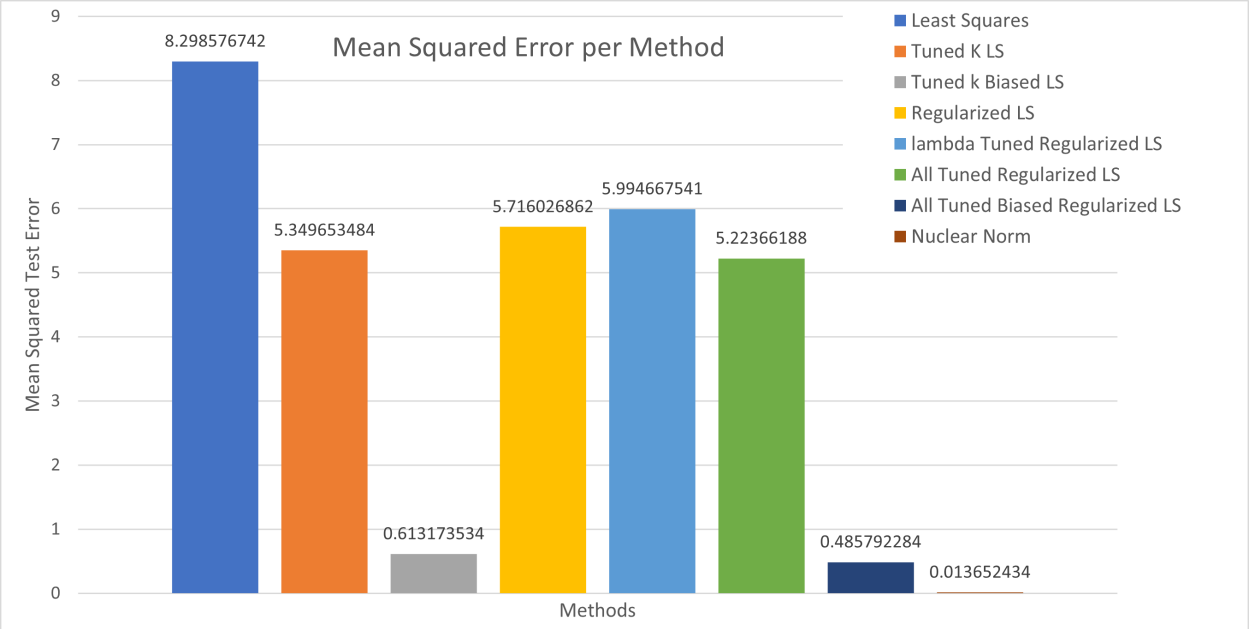

Figure 3: Mean squared test error of each model.

and this is expected. It is difficult to tell by the plots which of tuned least squares, and tuned regularized least squares (orange and green respectively), did better, though by the means in Figure 3, regularized least squares did slightly better. This matches our expectations that regularization reduces overfitting. We are not quite sure why tuning λ and not k seemed to perform worse that regularized least squares with arbitrary values for these. It could be the case that it's because the model was too poor to begin with such that regularizing a bad model will not improve it. Finally, as before we see that adding a bias term increased the performance of our model. As we expected, the regularization to prevent overfitting, and the increased expressiveness given by the bias term gave the best model out of the matrix factorization techniques.

Finally we observed that the nuclear norm optimization performed better than the matrix factorization methods. There are a few possible reasons for this. Firstly, it could be possible that CVXPY is performing some optimizations that lead to better performance than the vanilla gradient descent that we implemented. It is also possible that our matrix factorization models were to simple. One improvement to our objective functions could be:

$$f(Z,W) = \sum_{(i,j)\in\Omega} \left(z_i w_j^T + \beta + \beta_i + \beta_j - x_{ij}\right)^2 + \lambda(\|Z\|_F^2 + \|W\|_F^2) \tag{8}$$

where now we have more biases that capture the specific trends of user or items ratings. For example, a user that tends to rate items more strictly would have a lower rating. Similarly an item that is particularly popular might have a larger bias to represent its ratings. Each

user/item specific bias would be found using gradient descent. Another reason that may explain the performance difference is that while matrix factorization methods need a specific integral ks, the nuclear norm method can optimize the rank of its matrix with the use of singular values which are determined on a continuous scale.

6 Conclusion

The project discussed two methods to build a recommendation system via model-based collaborative filtering. The model is aiming to predict the ratings of users to certain items based on their past behaviors and similar decisions made by other users. The first approach focused on matrix factorization. We created different objective functions as least squares problems with bias terms and regularization to try to get the best prediction matrix, which matches the original matrix the most. And we implemented this approach using gradient descent as well as a bit of knowledge in machine learning. The second method is to predict the rating matrix through nuclear norm minimization. We did several transformations to the problem and finally constructed it as a semidefinite program problem, thus solving it via the convex optimization language CVXPY.

Although the models had different performances, both approaches successfully returned the prediction matrix. The recommendation system is widely being used nowadays, thus our methods can be applied to real-life problems, such as predicting the ratings of movies, books, restaurants, etc. Hence provide suggestions for items that are most pertinent to a particular user.

In the end, we want to discuss some further improvements to our project. Firstly, the matrix factorization method appeared to give a less accurate result than the nuclear norm minimization method. Thus, we want to explore more on it and adjust the objective function to see if the result will be improved. Moreover, in the project, we created the data instead of using real-life data. Although we tried to create data that is close to reality, it is still good to test our methods using real data to see the performance.

References

[1] E. Candes and B. Recht, "Exact matrix completion via convex optimization," *Communications of the ACM*, vol. 55, no. 6, pp. 111–119, 2012.

[2] A. Ramlatchan, M. Yang, Q. Liu, M. Li, J. Wang, and Y. Li, "A survey of matrix completion methods for recommendation systems," *Big Data Mining and Analytics*, vol. 1, no. 4, pp. 308–323, 2018.

[3] "Matrix completion using the nuclear norm for low rank factorization." https://www.youtube.com/watch?v=Ceq5dCc8RjY&t=231s. Accessed: 2022-12-08.

[4] "Cvxpy documentation." https://www.cvxpy.org/index.html. Accessed: 2022-12-08.

[5] "Recommender system — matrix factorization." https://towardsdatascience.com/recommendation-system-matrix-factorization-d61978660b4b. Accessed: 2022-12-08.

[6] "How to build simple recommender systems in python." https://medium.com/swlh/how-to-build-simple-recommender-systems-in-python-647e5bcd78bd. Accessed: 2022-12-08.

[7] "Cpsc340: Machine learning and data mining recommender systems." https://www.cs.ubc.ca/~fwood/CS340/lectures/L30.pdf. Accessed: 2022-12-08.

[8] "Build a recommendation engine with collaborative filtering." https://realpython.com/build-recommendation-engine-collaborative-filtering/. Accessed: 2022-12-08.

[9] "Svd and its application in recommender system." https://analyticsindiamag.com/singular-value-decomposition-svd-application-recommender-system/. Accessed: 2022-12-08.