Matrix Completion Using Stochastic Gradient Descent

Completed for the Certificate in Scientific Computation
Spring 2019

Hai Lan
Bachelor of Science in Mathematics
Department of Mathematics
College of Natural Sciences

[illegible]

George Biros
Professor
Oden Institute for Computational Engineering and Sciences

Matrix Completion Using Stochastic Gradient Descent

Hai Lan
The University of Texas at Austin

Table of Contents

Abstract

Data sizes have grown exponentially during the last decade, making statistical machine learning methods limited by computing time and underlying algorithms computationally complex in non-trivial ways. As a prevalent task in machine learning, matrix approximation is a common tool in recommendation systems, text mining, and computer vision. It is realistic to assume, in many real datasets, that the partially observed matrix is low-rank. This research implements the stochastic gradient descent (SGD) algorithm on matrix completion problems and tests it on a real-life dataset MovieLens, aiming to predict the unobserved ratings of movies in order to make good recommendations to users on what to watch. The SGD algorithm shows outstanding performance for large-scale problems. It selects an observation uniformly at random and updates the parameters with only $O(d)$ computation. It tries to lower the computation per iteration, at the cost of an increased number of iterations necessary for convergence.

Keywords: machine learning, recommendation systems, matrix completion, stochastic gradient descent.

Matrix Completion Using Stochastic Gradient Descent

Matrix approximation is used primarily for recommendation systems. Our essential goal is to predict the preference of a user for an item and give recommendations accordingly. Gradient descent is a very classical technique for finding the global, more often local, minimum of a function and there are many variations. Stochastic gradient descent is an iterative method for optimizing a differentiable objective function. It is called "stochastic" because samples are selected randomly (or shuffled) instead of as a single group (as in standard gradient descent) or in the order they appear in the training set. A batch is the total number of examples used to calculate the gradient in a single iteration, and thus the one example comprising each batch is chosen at random. The algorithm can be made practical for a variety of scientific computing problems. The research is therefore focused on stochastic gradient descent for the matrix completion problem.

Materials and Methods

Given few entries on the observed matrix, our goal is to complete the matrix by approximating the unobserved entries. In recommendation systems, the matrix corresponds to ratings of various items by different users, and hence we try to predict all the ratings based on the few observed ones. See Figure-1 below showing an example of movie recommendation.

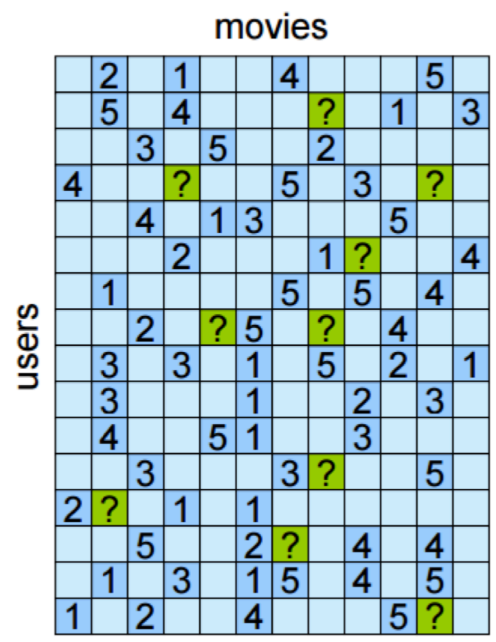


Figure-1. Movie Recommendation Systems

The data collected from given users and items can be constructed to a matrix $M \in \mathbb{R}^{m \times n}$. We want to predict unobserved entries of M given training data $M_{a_1,b_1}, \dots, M_{a_m,b_m}$ (observed ratings of users on items), where M_{ij} is the rating of user i for item j.

A popular assumption is that the observed matrix is low-rank. This allows us to decompose M using singular value decomposition (SVD), or $M \approx \widehat{M} = UV^T$, where $U \in \mathbb{R}^{m \times r}, V \in \mathbb{R}^{n \times r}, r \ll \min(m, n)$. See Figure-2. In many real datasets, such an assumption is reasonable. For instance, in our MovieLens dataset, the ratings matrix is low-rank since user preferences can often be described by just a few factors, such as the movie genre, released time, and main actors.

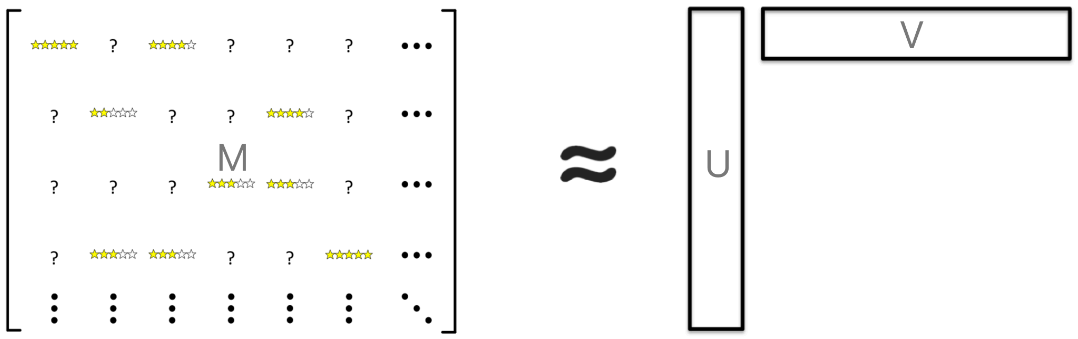


Figure-2. Singular Value Decomposition

Alternating Least Squares Minimization

Alternating minimization represents a widely applicable and empirically successful approach for finding low-rank matrices that best fit the observed data. We wish to solve the following minimization problem (λ_U, λ_V are regularization parameters of U, V, respectively):

$$\sum_i \sum_j K_{ij}\left(U_i V_j^T - M_{ij}\right)^2 + \lambda_U \sum_i \|u_i\|^2 + \lambda_V \sum_j \left\|v_j\right\|^2$$
$$= \sum_i \sum_j K_{ij}\left(U_i V_j^T - M_{ij}\right)^2 + \lambda_U \sum_i u_i^T u_i + \lambda_V \sum_j v_j^T v_j.$$

This can be solved by setting the partial derivatives $\frac{\partial}{\partial u_i}$ and $\frac{\partial}{\partial v_j}$ to be zeros.

$$\frac{\partial}{\partial u_i} = \sum_j 2K_{ij}\left(U_i V_j^T - M_{ij}\right)V_j + 2\lambda_U u_i = 0;$$
$$\frac{\partial}{\partial v_j} = \sum_i 2K_{ij}\left(U_i V_j^T - M_{ij}\right)U_i + 2\lambda_V v_j = 0.$$

We solve in an alternating minimization scheme. Given a subset of entries drawn from the unknown matrix, alternating minimization starts from a poor approximation to the target matrix and gradually improves the approximation quality by fixing one of the factors and minimizing a certain objective function over the other. The update step is usually combined with an initialization procedure. See the pseudocode below.

```
Start with $U^{(0)}, V^{(0)}$.
For $i = 1$ to $n$: $u_i^{(i)} \leftarrow u_i^{(i-1)} - \alpha K_{ij} \frac{\partial}{\partial u_i}$.
For $i = 1$ to $n$: $v_j^{(i)} \leftarrow v_j^{(i-1)} - \alpha K_{ij} \frac{\partial}{\partial v_j}$.
```

A major advantage of alternating minimization over alternatives is that each update is computationally cheap and has a small memory footprint.

Stochastic Gradient Descent

Again, our task is to find two matrices U and V such that the product UV^T approximates M. In this way, each row of U would represent the strength of the associations between a user and the features. Similarly, each row of V would represent the strength of the associations between an item and the features. To get the prediction of a rating of an item V_j by U_i, we can calculate the dot product of their vectors $M_{ij} \approx U_i^T V_j = \sum_{k=1}^{n} U_{ik} V_{kj}$. To approach this problem, we first initialize the two matrices with some values, calculate the difference of the product to M, and then try to minimize the difference iteratively. This is called gradient descent, which finds a local minimum of the difference. See Figure-3.

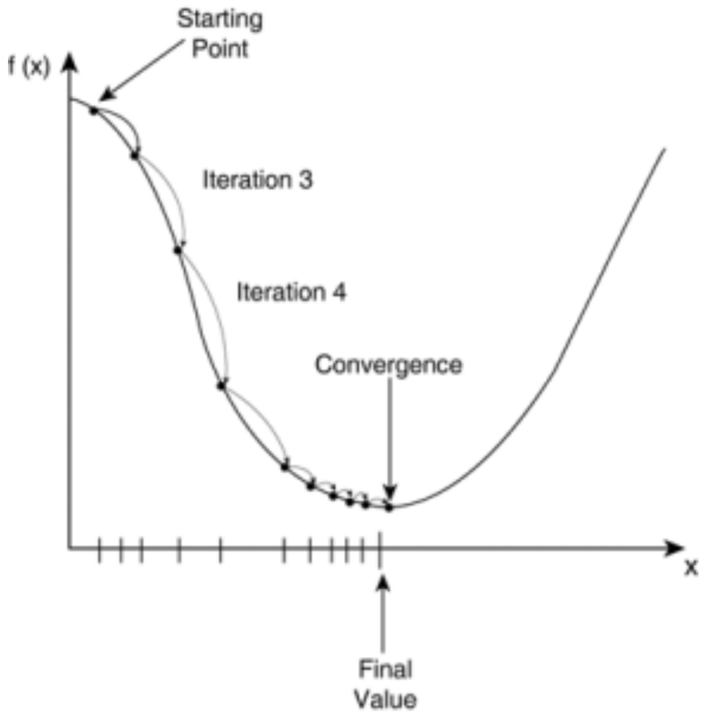


Figure-3. Stochastic Gradient Descent

The difference is also called the error between the estimated rating and the true rating, which can be calculated by $e_{ij}^2 = \left(M_{ij} - \sum_{k=1}^{n} U_{ik} V_{kj}\right)^2$. Here we consider the squared error because the estimated rating can be either higher or lower than the true rating.

We use stochastic gradient descent (SGD) to minimize the partial derivates (towards 0). Gradient descent updates the parameters in the opposite direction of the gradient of the functions $\nabla \frac{\partial}{\partial u_i}, \nabla \frac{\partial}{\partial v_j}$ with respect to the parameters λ_U, λ_V. SGD performs the parameter update for each training set, one at a time. It is therefore usually much faster than the standard gradient descent.

In order to apply SGD, we wish to find the value of the derivative of

$$\frac{\partial}{\partial U_i}\left(M_{ij} - U_i V_j\right)^2 = -2V_j\left(M_{ij} - U_i V_j\right);$$

$$\frac{\partial}{\partial V_j}\left(M_{ij} - U_i V_j\right)^2 = -2U_i\left(M_{ij} - U_i V_j\right).$$

The SGD procedure then becomes:

```
1. Randomly initialize all vectors $U_i$ and $V_j$.
2. For a given number of times:
      For all known ratings $M_{ij}$:
         a. Compute $\frac{\partial}{\partial U_i}$ and $\frac{\partial}{\partial V_j}$.
         b. Update $U_i$ and $V_j$:
            $U_i \leftarrow U_i - \alpha \cdot V_j\big(M_{ij} - U_i V_j\big)$;
            $V_j \leftarrow V_j - \alpha \cdot U_i\big(M_{ij} - U_i V_j\big)$.
            Note that $\alpha$ is called the learning rate.
```

SGD has been successfully applied to large-scale and sparse machine learning problems. On large datasets, SGD can converge faster than batch training because it performs updates more frequently. It stands out due to its efficiency.

Results

Here, I analyze the performance of the SGD algorithm in different test cases. For each test, I show the respective relative Frobenius error, defined as the square root of the sum of the absolute squares of the elements of the matrix, or $\|M\|_F = \sqrt{\sum_{i=1}^{m}\sum_{j=1}^{n}\left|m_{ij}\right|^2}$. I also display the accuracy of the approximation, the difference between the expected value and the true value. I show these for different number of observations (different percentages of training and test sets) and batch sizes.

Test 1: MovieLens

The first test deals with the MovieLens dataset, which describes 5-star rating and free-text tagging activity from MovieLens, a movie recommendation service. It contains 100,836 ratings and 3,683 tag applications across 9,742 movies. These data were created by 610 users. For this test, I also record the running time for the algorithm which stops when the difference between consecutive relative errors is less than 0.001 (with a maximum of 50 iterations).

When the training-to-test ratio is 70%:30%, the loop breaks after 16 iterations, giving a running time of `81.34656095504761` seconds; the respective errors are as follows.

```
iter 1 : MSE = 617.5445561492028 , RelErr = 0.9708185155550686
iter 2 : MSE = 407.095036396454 , RelErr = 0.6399787594091518
iter 3 : MSE = 328.07135323940935 , RelErr = 0.5157486062772041
iter 4 : MSE = 297.9009430924541 , RelErr = 0.4683188418968052
iter 5 : MSE = 276.573940796015 , RelErr = 0.43479146560548837
iter 6 : MSE = 265.5307787036773 , RelErr = 0.41743092680263794
iter 7 : MSE = 257.02293485995136 , RelErr = 0.4040560662380105
iter 8 : MSE = 251.27975288525045 , RelErr = 0.3950274263711008
iter 9 : MSE = 248.08474839921885 , RelErr = 0.3900046802689206
iter 10 : MSE = 244.33763025217095 , RelErr = 0.3841139770947069
iter 11 : MSE = 241.30308160482014 , RelErr = 0.37934347756739994
iter 12 : MSE = 240.61462834493662 , RelErr = 0.3782611861519321
iter 13 : MSE = 237.98202763358276 , RelErr = 0.37412257382153974
iter 14 : MSE = 235.4921953770808 , RelErr = 0.37020840239671005
iter 15 : MSE = 233.60780896728622 , RelErr = 0.3672460295624381
iter 16 : MSE = 232.64858911094015 , RelErr = 0.36573807618845705
```

When I change the training-to-test ratio while holding the batch size constant at `10%`, the result can be summarized by Table-1 below.

Table-1. Performance Analysis of SGD in Test 1 with Varying Training-to-Test Ratio

Training-to-test ratio	Number of iterations	Final Relative Error	Running time
70%:30%	16	0.36573807618845705	81.34656095504761
60%:40%	17	0.3723791913651768	98.40568494796753
50%:50%	18	0.3860155879489815	110.762216091156

When the size of training set increases, the number of iterations decreases, meaning that the matrix completion converges faster, the error becomes smaller and it takes shorter to run.

When I change the batch size while holding the training-to-test ratio constant at `70%:30%`, the result can be summarized by Table-2 below.

Table-2. Performance Analysis of SGD in Test 1 with Varying Batch Size

Batch size	Number of iterations	Final Relative Error	Running time
70%	4	0.35230901016196353	35.25954723358154
60%	5	0.35206724018416885	39.045366048812866
50%	6	0.3525845983008475	42.40799427032471
40%	6	0.356113766665196	40.46257710456848
30%	9	0.35436462158130927	54.95181107521057
20%	8	0.36467036156495847	42.93065690994263
10%	18	0.3625631227163355	81.97705507278442

When the batch size increases, number of iterations, final relative error, and running time all decrease.

Instead of breaking the loop if the error is to some threshold, if I allow the iterations to run for 50 times while holding training-to-test ratio = `70%:30%` and batch size = `10%`, the error versus number of iterations can be plotted, as shown below in Figure-4. The final relative error is `0.35196906710451076`.

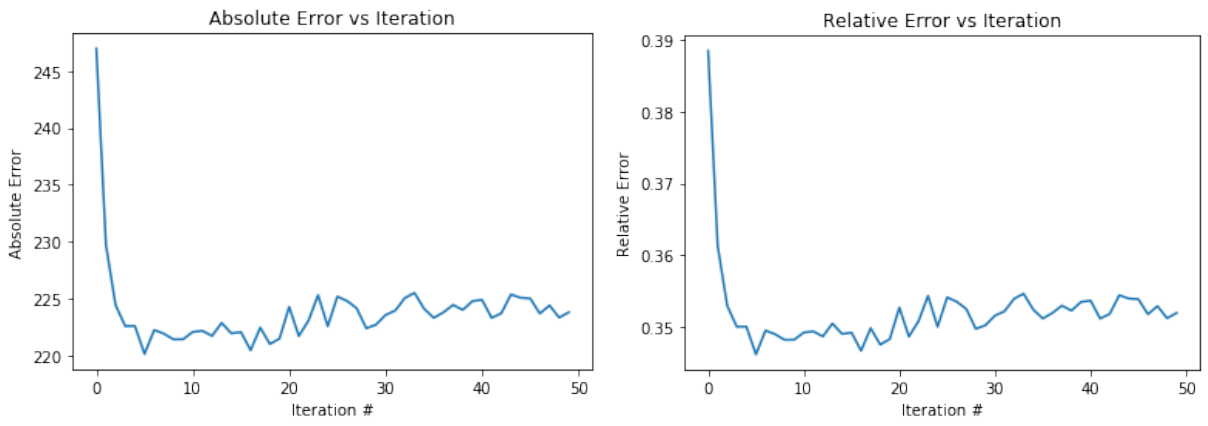


Figure-4. Error versus Number of Iterations

Hence, the convergence behavior proves the algorithm to be successful.

Test 2: Matrix from Well-Separated Kernel

The second test deals with a low-rank matrix from well-separated kernel. I set the distance between the point sets (two samples of 1,000 points from the standard normal distribution) to be 5. A simple visualization can be shown below in Figure-5.

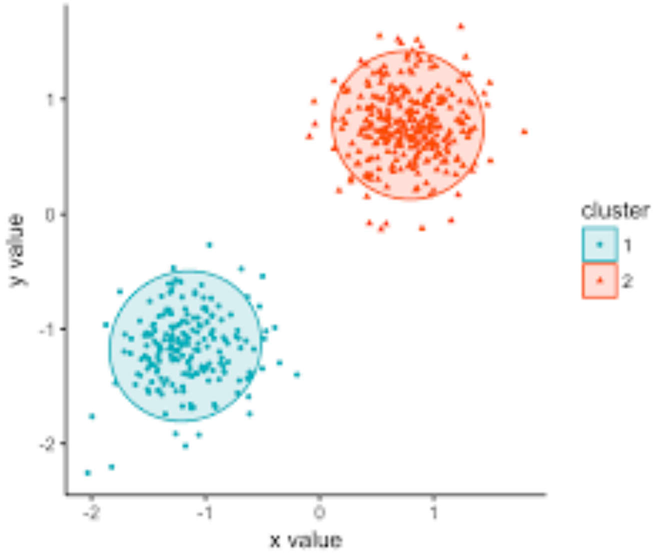


Figure-5. Matrix Visualization for Test 2

The 1000×1000 matrix is low-rank (rank = 12). The relative errors show a convergent trend with a faster rate. The estimates are considered good.

Test 3: Rank-1 Matrix

The third test deals with a rank-1 matrix of random normalized vectors. The relative errors show a convergent trend with a faster rate. The estimates are considered good.

Test 4: Low-Rank Matrix with High Condition Number

The fourth test deals with a low-rank matrix of high condition number. The 1000×1000 matrix is low-rank (rank = 20) and has a high condition number. The relative errors again show a convergent trend with a faster rate. The estimates are considered good as well.

Discussion

Unlike the traditional algorithm which computes true gradient, in SGD's iteration, the gradient is computed against a single randomly chosen training example. In each iteration, a random index is chosen and the gradient is computed only with respect to the training example at the chosen index. Several passes can be made over the training set until the algorithm converges. If this is done, the data can be shuffled for each pass to prevent cycles. The "stochastic" aspect of the algorithm therefore involves taking the derivative and updating one individual sample at a time.

I conduct tests of convergence for the algorithm for different low-rank matrices (with different condition numbers) and investigate the effect of different number of observations as well as batch size. Results have shown that the algorithm is highly efficient (almost linear in the size of training set). It tries to lower the computation per iteration, at the cost of an increased number of iterations necessary for convergence. However, a weakness of this algorithm is that it requires a number of hyperparameters such as the regularization parameter and the number of iterations. It is also sensitive to feature scaling, so using scaled data is crucial.

Matrix completion is the state-of-the-art solution for sparse data problem. I apply the SGD algorithm to a real-life dataset MovieLens which can be made practical for recommendation systems in general. The model learns to factorize rating matrix into user and movie representations, which allows it to predict better personalized movie ratings for users. With matrix completion, less-known movies can have rich latent representations as much as popular movies have, which improves recommender's ability to recommend less-known movies.

If I had more time, I would construct local matrix approximations as a weighted sum of low-rank matrices, known as the LLORMA (local low-rank matrix approximation) algorithm. In LLORMA, we randomly choose among training data points as anchor points. Our goal is to estimate the local models at the anchor points. The weight is therefore proportional to the distance between the anchor point and the test point. I am also interested in understanding and testing LLORMA in the context of dense matrix approximation and its connections to Fast Multipole Methods (FMM). FMM finds application in potential theory, and function approximation can be used for solution of partial differential approximation and data analysis.

Acknowledgements

This work would not have been possible without the support of Dr. George Biros for coming up with this topic and supervising this research.

I would also express my deepest appreciation to Dr. William Ruys who has invested his full effort in guiding me all along till the completion of the research by providing all the necessary information.

Literature Cited

Candès, Emmanuel and Benjamin Recht. "Exact Matrix Completion via Convex Optimization."

Hardt, Moritz. "Understanding Alternating Minimization for Matrix Completion."

Harper, Maxwell and Joseph Konstan. 2015. The MovieLens Datasets: History and Context. ACM Transactions on Interactive Intelligent Systems (TiiS) 5, 4: 19:1–19:19. https://doi.org/10.1145/2827872.

Lee, Joonseok et al. "LLORMA: Local Low-Rank Matrix Approximation."

Computer Code

See Appendix A for the complete computer code run via Jupyter Notebook.

Reflection

By doing this research, I have a better understanding of scientific computation and data sciences. It applies the tools and techniques of computation learned in class to a scientific discipline in the pursuit of new knowledge. As a student of math major, I often encounter abstract materials which lack practicality, but this research combines linear algebra with machine learning—I clearly see the significance of math in the current world of tremendous amount of data. Every day, when we are shopping online, browsing Netflix, streaming music, and even typing a note on a smartphone, whether we realize it or not, we are contributing to and taking advantage of recommendation systems algorithms. Recommendation systems with strong algorithms are at the core of today's most successful online companies such as Amazon and Spotify.

As an undergraduate and since this is the first time that I conduct an independent research, I was at first struggling to find the right information, but I am given this opportunity to start to really think as a scientist who does something novice. Due to the lack of mastery of machine learning algorithms and technical capabilities, I spent a long time on understanding the materials and consulting with peers. These experiences develop significant transferable skills such as self-efficacy and critical thinking.

Appendix A

Computer Code

In [1]:
```
# import necessary libraries
import pandas as pd
import numpy as np
import scipy as sp
from scipy import sparse
from scipy.sparse.linalg import norm
import random
import matplotlib.pyplot as plt
import time
```

In [2]:
```
# compute mean squared error: NORM(UV-M, 'fro')
def MSE(U, V, M, obs):
    s = 0
    st = 0
    nSamp = len(obs)
    for l in range(nSamp):
        i, j = obs[l]
        s += (np.dot(U[i, :], V[j, :].T) - M[i, j])**2
    return np.sqrt(s)

# compute relative Frobenius error: NORM(UV-M, 'fro')/NORM(M, 'fro')
def RelError(U, V, M, obs):
    s = 0
    st = 0
    nSamp = len(obs)
    for l in range(nSamp):
        i, j = obs[l]
        s += (np.dot(U[i, :], V[j, :].T) - M[i, j])**2
        st += (M[i, j])**2
    return np.sqrt(s)/np.sqrt(st)
```

In [3]:
```
### TEST 1: MOVIELENS ###

# read in the movielens dataset https://www.kaggle.com/grouplens/movie
lens-latest-small
dataset = pd.read_csv('ratings.csv').drop('timestamp', axis=1)
print(dataset.shape)

# convert to numpy matrix
M = dataset.pivot(index='userId', columns='movieId', values='rating').
values
print(M.shape)
M
```

```
(100836, 3)
(610, 9724)
```

Out[3]:
```
array([[4. , nan, 4. , ..., nan, nan, nan],
       [nan, nan, nan, ..., nan, nan, nan],
       [nan, nan, nan, ..., nan, nan, nan],
       ...,
       [2.5, 2. , 2. , ..., nan, nan, nan],
       [3. , nan, nan, ..., nan, nan, nan],
       [5. , nan, nan, ..., nan, nan, nan]])
```

In [4]:
```
# produce vectors of users, movies, ratings (given values)
users = dataset['userId'].values
movies = dataset['movieId'].values
ratings = dataset['rating'].values
obs = [(users[i]-1, movies[i]-1) for i in range(len(users))] # form tu
ples

# form sparse matrix
M = sparse.coo_matrix((ratings, (users-1,movies-1))).tocsr()
print(M)
```

```
  (0, 0)	4.0
  (0, 2)	4.0
  (0, 5)	4.0
  (0, 46)	5.0
  (0, 49)	5.0
  (0, 69)	3.0
  (0, 100)	5.0
  (0, 109)	4.0
  (0, 150)	5.0
  (0, 156)	5.0
  (0, 162)	5.0
  (0, 215)	5.0
  (0, 222)	3.0
  (0, 230)	5.0
  (0, 234)	4.0
  (0, 259)	5.0
  (0, 295)	3.0
  (0, 315)	3.0
  (0, 332)	5.0
  (0, 348)	4.0
  (0, 355)	4.0
  (0, 361)	5.0
  (0, 366)	4.0
  (0, 422)	3.0
  (0, 440)	4.0
  :	:
  (609, 156370)	5.0
  (609, 156725)	4.5
  (609, 157295)	4.0
  (609, 158237)	5.0
  (609, 158720)	3.5
  (609, 158871)	3.5
  (609, 158955)	3.0
  (609, 159092)	3.0
  (609, 160079)	3.0
  (609, 160340)	2.5
  (609, 160526)	4.5
  (609, 160570)	3.0
  (609, 160835)	3.0
  (609, 161581)	4.0
  (609, 161633)	4.0
  (609, 162349)	3.5
  (609, 163936)	3.5
  (609, 163980)	3.5
  (609, 164178)	5.0
  (609, 166527)	4.0
  (609, 166533)	4.0
  (609, 168247)	5.0
  (609, 168249)	5.0
  (609, 168251)	5.0
  (609, 170874)	3.0
```

In [5]:
```
# set up parameters for SGD
```

```
maxiter = 50

alpha = 0.1 # step size (learning rate)
beta = 0.01 # regularization

nSamp = len(obs) # size of the observed set

np.random.shuffle(obs)

# form training and test sets (hyperparameter)
train = obs[1:(int)(np.floor(0.7*nSamp))] # training set [70%]
test = obs[(int)(np.floor(0.7*nSamp)):len(obs)+1] # test set [30%]

start = time.time()

# start SGD

k = 10 # guess of the rank of matrix

# initialize matrices
U = np.random.normal(size=(np.max(users), k), scale=1/k)
V = np.random.normal(size=(np.max(movies), k), scale=1/k)

errorVec = np.zeros((2, maxiter), dtype='float')

RelErr_prev = 1
for itr in range(maxiter):
    np.random.shuffle(train)
    selected_updates = train[1:(int)(np.floor(0.1*len(train)))] # batc
h size (hyperparameter) [10%]

    for l in selected_updates:
        i, j = l
        error = ( np.dot(U[i, :], V[j, :].T) - M[i, j] )
        U_temp = np.copy(U[i, :])
        # update U
        U[i, :] -= alpha * (error*V[j, :] + beta*U[i, :])
        # update V
        V[j, :] -= alpha * (error*U_temp + beta*V[j, :])

    errorVec[0, itr] = MSE(U, V, M, test)
    errorVec[1, itr] = RelError(U, V, M, test)

    # print errors during iteration
    if RelErr_prev - errorVec[1, itr] < 0.001:
        break
    else:
        print("iter", itr+1,
              ": MSE =", errorVec[0, itr],
              ", RelErr =", errorVec[1, itr])
        RelErr_prev = errorVec[1, itr]

end = time.time()
```

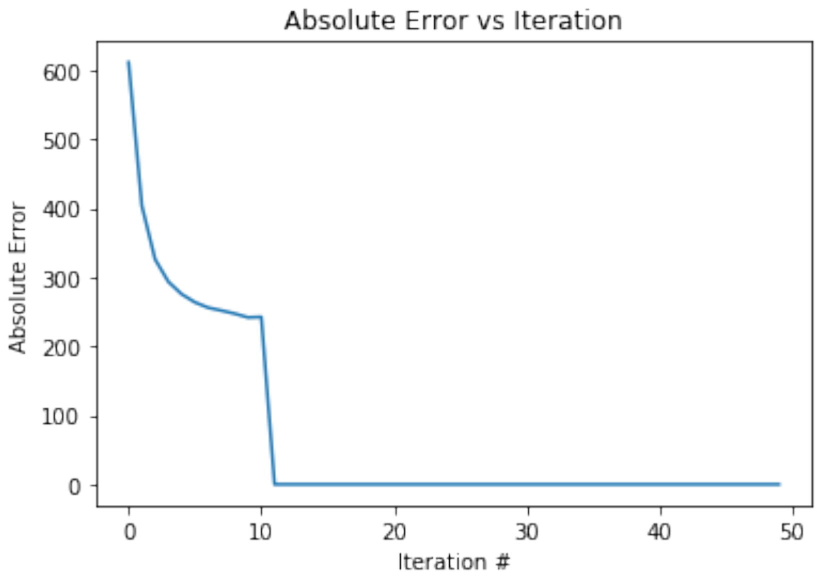
```
print("\nRunning time:", end-start)
```

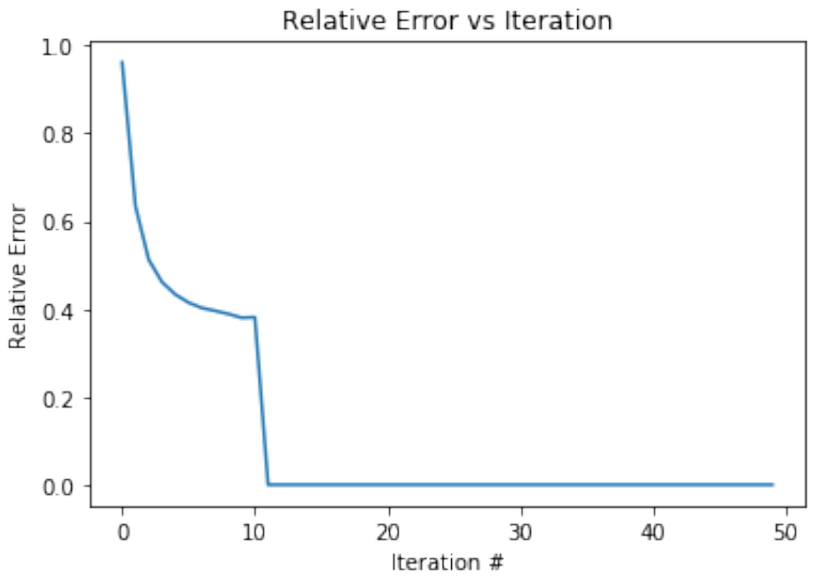
```
iter 1 : MSE = 611.1950647241977 , RelErr = 0.9613515290180383
iter 2 : MSE = 403.547888914588 , RelErr = 0.6347423309366939
iter 3 : MSE = 325.84540103033703 , RelErr = 0.5125237302846443
iter 4 : MSE = 293.36336760592764 , RelErr = 0.461432590482555
iter 5 : MSE = 275.4578085038903 , RelErr = 0.4332688541990592
iter 6 : MSE = 263.6702196109559 , RelErr = 0.414728101402287
iter 7 : MSE = 255.86212088406293 , RelErr = 0.40244670699474283
iter 8 : MSE = 251.70578997879747 , RelErr = 0.39590919499325944
iter 9 : MSE = 247.23193494378745 , RelErr = 0.38887224782737756
iter 10 : MSE = 241.71540571643246 , RelErr = 0.3801952736276864

Running time: 49.79625988006592
```

In [6]:
```
# print matrices
print("True Answer")
print(M)
print("\nEstimated Answer")
print(np.dot(U, V.T))
print("\nAccuracy (estimated - true)")
print(np.dot(U, V.T)-M)
```

```
True Answer
  (0, 0)	4.0
  (0, 2)	4.0
  (0, 5)	4.0
  (0, 46)	5.0
  (0, 49)	5.0
  (0, 69)	3.0
  (0, 100)	5.0
  (0, 109)	4.0
  (0, 150)	5.0
  (0, 156)	5.0
  (0, 162)	5.0
  (0, 215)	5.0
  (0, 222)	3.0
  (0, 230)	5.0
  (0, 234)	4.0
  (0, 259)	5.0
  (0, 295)	3.0
  (0, 315)	3.0
  (0, 332)	5.0
  (0, 348)	4.0
  (0, 355)	4.0
  (0, 361)	5.0
  (0, 366)	4.0
  (0, 422)	3.0
  (0, 440)	4.0
  :	:
  (609, 156370)	5.0
  (609, 156725)	4.5
  (609, 157295)	4.0
  (609, 158237) 5.0
  (609, 158720) 3.5
  (609, 158871) 3.5
  (609, 158955) 3.0
  (609, 159092) 3.0
  (609, 160079) 3.0
  (609, 160340) 2.5
  (609, 160526) 4.5
  (609, 160570) 3.0
  (609, 160835) 3.0
  (609, 161581) 4.0
  (609, 161633) 4.0
  (609, 162349) 3.5
  (609, 163936) 3.5
  (609, 163980) 3.5
  (609, 164178) 5.0
  (609, 166527) 4.0
  (609, 166533) 4.0
  (609, 168247) 5.0
  (609, 168249) 5.0
  (609, 168251) 5.0
  (609, 170874) 3.0

Estimated Answer
[[ 4.27020683  4.90147728  5.21642394 ...  0.06357361 -0.34890203
   0.62410587]
 [ 2.95664264  3.26890648  3.2413917  ...  0.02221734 -0.12954662
   0.44331462]
 [ 2.8768171   3.18806988  3.27222652 ...  0.06994537 -0.25445925
   0.3869408 ]
 ...
 [ 2.32408863  2.84287845  2.81155619 ...  0.19506648 -0.16784314
   0.43128932]
 [ 3.24103594  3.25026083  3.50530818 ... -0.01348508 -0.24585766
   0.44636067]
 [ 3.68876344  3.69933534  3.80061377 ... -0.14740372 -0.30785077
   0.35316538]]

Accuracy (estimated - true)
[[ 0.27020683  4.90147728  1.21642394 ...  0.06357361 -0.34890203
   0.62410587]
 [ 2.95664264  3.26890648  3.2413917  ...  0.02221734 -0.12954662
   0.44331462]
 [ 2.8768171   3.18806988  3.27222652 ...  0.06994537 -0.25445925
   0.3869408 ]
 ...
 [-0.17591137  0.84287845  0.81155619 ...  0.19506648 -0.16784314
   0.43128932]
 [ 0.24103594  3.25026083  3.50530818 ... -0.01348508 -0.24585766
   0.44636067]
 [-1.31123656  3.69933534  3.80061377 ... -0.14740372 -0.30785077
   0.35316538]]
```

```
In [7]: # plot error curves for maxiter
        plt.plot(errorVec[0, :])
        plt.title('Absolute Error vs Iteration')
        plt.xlabel('Iteration #')
        plt.ylabel('Absolute Error')
        plt.show()

        plt.plot(errorVec[1, :])
        plt.title('Relative Error vs Iteration')
        plt.xlabel('Iteration #')
        plt.ylabel('Relative Error')
        plt.show()
```





```
In [8]: ### TEST 2: MATRIX FROM WELL-SEPARATED KERNEL ###

        # construct a low-rank matrix
        N = 1000  # matrix size

        d = 1 # dimension of point set
        # generate well-separated point set
        X = np.random.randn(d, N) + 5 # [the larger the distance, the smaller
        the rank]
        Y = np.random.randn(d, N)

        bandwidth = 10 # bandwidth of Gaussian kernel [the larger the bandwidt
        h, the smaller the rank]
        M = np.zeros([N, N]) # fill in full true matrix M
        for i in range(N):
            for j in range(N):
                M[i, j] = np.exp(-np.linalg.norm(X[:, i]-Y[:, j])**2/bandwidth
        )

        print(M)
        print("\nRank of M:", np.linalg.matrix_rank(M))
```

```
[[0.00513907 0.22012294 0.01979851 ... 0.15135983 0.01846026 0.02577
823]
 [0.03506236 0.55706278 0.10071643 ... 0.43790368 0.09545886 0.12311
703]
 [0.00885863 0.29268294 0.03160395 ... 0.20842955 0.02959429 0.04047
517]
 ...
 [0.00442005 0.20294327 0.01738219 ... 0.13824478 0.01618869 0.02273
235]
 [0.00514363 0.22022759 0.01981367 ... 0.1514402  0.01847451 0.02579
729]
 [0.01771638 0.41117296 0.05693121 ... 0.30710481 0.05362276 0.07129
178]]

Rank of M: 12
```

In [9]:
```
col_idx = [np.random.randint(1, N) for i in range(100000)]
row_idx = [np.random.randint(1, N) for i in range(100000)]
obs = [(i, j) for i, j in zip(row_idx, col_idx)]

# set up parameters for SGD
maxiter = 100

alpha = 0.1 # step size (learning rate)
beta = 0.01 # regularization

nSamp = len(obs) # size of the observed set

np.random.shuffle(obs)

# form training and test sets (hyperparameter)
train = obs[1:(int)(np.floor(0.7*nSamp))] # training set [70%]
test = obs[(int)(np.floor(0.7*nSamp)):len(obs)+1] # test set [30%]

# start SGD

k = 10 # guess of the rank of matrix

# initialize matrices
U = np.random.normal(size=(N, k), scale=1/k)
V = np.random.normal(size=(N, k), scale=1/k)

errorVec = np.zeros((2, maxiter), dtype='float')

for itr in range(maxiter):
    np.random.shuffle(train)
    selected_updates = train[1:(int)(np.floor(0.1*len(train)))] # batc
h size (hyperparameter) [10%]

    for l in selected_updates:
        i, j = l
        error = ( np.dot(U[i, :], V[j, :].T) - M[i, j] )
        U_temp = np.copy(U[i, :])
        # update U
        U[i, :] -= alpha * (error*V[j, :] + beta*U[i, :])
        # update V
        V[j, :] -= alpha * (error*U_temp + beta*V[j, :])

    errorVec[0, itr] = MSE(U, V, M, test)
    errorVec[1, itr] = RelError(U, V, M, test)

    # print errors during iteration
    print("iter", itr+1,
          ": MSE =", errorVec[0, itr],
          ", RelErr =", errorVec[1, itr])
```

```
iter 1 : MSE = 37.85447802254936 , RelErr = 1.0104046754648228
iter 2 : MSE = 37.82370083405363 , RelErr = 1.0095831764829792
iter 3 : MSE = 37.80564359178891 , RelErr = 1.0091011959363339
iter 4 : MSE = 37.78661736804037 , RelErr = 1.0085933515164343
iter 5 : MSE = 37.76372378055331 , RelErr = 1.0079822801440776
iter 6 : MSE = 37.7449215286283 , RelErr = 1.007480413938368
iter 7 : MSE = 37.72454236620334 5 , RelErr = 1.0069364571313564
iter 8 : MSE = 37.703730954034576 , RelErr = 1.006380962794707
iter 9 : MSE = 37.676434936142094 , RelErr = 1.0056523825700976
iter 10 : MSE = 37.650927236902966 , RelErr = 1.0049715358138378
iter 11 : MSE = 37.61397312777905 , RelErr = 1.0039851636172854
iter 12 : MSE = 37.57112190415671 , RelErr = 1.002841386739115
iter 13 : MSE = 37.52332077485157 , RelErr = 1.0015654879005775
iter 14 : MSE = 37.445621372538014 , RelErr = 0.9994915499286507
iter 15 : MSE = 37.343153350603686 , RelErr = 0.9967564925759538
iter 16 : MSE = 37.22385034127217 , RelErr = 0.993572078875842
iter 17 : MSE = 37.06926265860341 , RelErr = 0.9894458532481963
iter 18 : MSE = 36.87070176763217 , RelErr = 0.9841459029363129
iter 19 : MSE = 36.61503184760664 , RelErr = 0.9773216090597566
iter 20 : MSE = 36.280640000566294 , RelErr = 0.9683960841724337
iter 21 : MSE = 35.87923325426267 , RelErr = 0.9576818100781852
iter 22 : MSE = 35.372255525423085 , RelErr = 0.9441496549849117
iter 23 : MSE = 34.721152457239356 , RelErr = 0.9267705331829825
iter 24 : MSE = 33.92202098277593 , RelErr = 0.9054402647368569
iter 25 : MSE = 33.02198498336658 , RelErr = 0.8814166715083848
iter 26 : MSE = 31.87957274759806 , RelErr = 0.8509236169312979
iter 27 : MSE = 30.58741149140139 , RelErr = 0.8164334893976971
iter 28 : MSE = 29.07497770321732 , RelErr = 0.7760638884748724
iter 29 : MSE = 27.385516745133277 , RelErr = 0.73096911131148
iter 30 : MSE = 25.721454849971266 , RelErr = 0.6865522812040096
iter 31 : MSE = 24.003267835194254 , RelErr = 0.6406907534867474
iter 32 : MSE = 22.31230223036167 , RelErr = 0.5955558145726437
iter 33 : MSE = 20.590820271652436 , RelErr = 0.549606338825758
iter 34 : MSE = 18.97550982814372 , RelErr = 0.5064907733839064
iter 35 : MSE = 17.48328761046946 , RelErr = 0.4666606559359147
iter 36 : MSE = 16.16248500872148 , RelErr = 0.4314060389424564
iter 37 : MSE = 15.061553382774507 , RelErr = 0.4020201770753009
iter 38 : MSE = 14.031423442393423 , RelErr = 0.37452414060961964
iter 39 : MSE = 13.182248619465842 , RelErr = 0.3518581244288672
iter 40 : MSE = 12.454546232372772 , RelErr = 0.33243442787630195
iter 41 : MSE = 11.907154520891144 , RelErr = 0.31782355028707027
iter 42 : MSE = 11.32335649128339 , RelErr = 0.3022409220365513 4
iter 43 : MSE = 10.831389840838579 , RelErr = 0.2891094398513764
iter 44 : MSE = 10.380974071487998 , RelErr = 0.2770870260438523
iter 45 : MSE = 10.079162370102685 , RelErr = 0.2690311243349962
iter 46 : MSE = 9.768914404483771 , RelErr = 0.26075004343280894
iter 47 : MSE = 9.568785754186708 , RelErr = 0.25540824678105845
iter 48 : MSE = 9.357090388824302 , RelErr = 0.24975771352554788
iter 49 : MSE = 9.193214155309246 , RelErr = 0.24538355962907774
iter 50 : MSE = 9.06599164360533 , RelErr = 0.2419877600469664 3
iter 51 : MSE = 8.891421452717822 , RelErr = 0.23732816503248877
iter 52 : MSE = 8.770081882522975 , RelErr = 0.234089391829204
iter 53 : MSE = 8.682893558733776 , RelErr = 0.2317621773294097
iter 54 : MSE = 8.558678242876143 , RelErr = 0.2284466452621479 5
iter 55 : MSE = 8.47883372592968 , RelErr = 0.2263154408588552 7
iter 56 : MSE = 8.42919168668197 , RelErr = 0.22499041422627167
iter 57 : MSE = 8.352547564856287 , RelErr = 0.22294464360453878
iter 58 : MSE = 8.283093405831014 , RelErr = 0.2210907861304558
iter 59 : MSE = 8.188481831529732 , RelErr = 0.218565431614402
iter 60 : MSE = 8.125971435850433 , RelErr = 0.21689691577799566
iter 61 : MSE = 8.077476085283065 , RelErr = 0.21560248691485337
iter 62 : MSE = 7.991016194616249 , RelErr = 0.2132947156197931
iter 63 : MSE = 7.886574468416388 , RelErr = 0.21050697652052208
iter 64 : MSE = 7.908145184727047 , RelErr = 0.21108273806187614
iter 65 : MSE = 7.857692647370355 , RelErr = 0.20973606833355762
iter 66 : MSE = 7.8138514734960705 , RelErr = 0.20856586788768022
iter 67 : MSE = 7.7491376571454715 , RelErr = 0.2068385387571909
iter 68 : MSE = 7.691706233848007 , RelErr = 0.2053055899054384
iter 69 : MSE = 7.695043618386431 , RelErr = 0.20539467075181644
iter 70 : MSE = 7.654452689391465 , RelErr = 0.2043112252367695
iter 71 : MSE = 7.640303214643176 , RelErr = 0.2039335500926947
iter 72 : MSE = 7.5557190211119245 , RelErr = 0.2016758445561532
iter 73 : MSE = 7.585442527254736 , RelErr = 0.20246921884492333
iter 74 : MSE = 7.560660579262995 , RelErr = 0.2018077437057584
iter 75 : MSE = 7.514206997874625 , RelErr = 0.20056781336518062
iter 76 : MSE = 7.53795981557819 , RelErr = 0.20120181915041027
iter 77 : MSE = 7.506361237919813 , RelErr = 0.20035839580205386
iter 78 : MSE = 7.453003536662651 , RelErr = 0.19893418198010718
iter 79 : MSE = 7.402114965946253 , RelErr = 0.19757587373058008
iter 80 : MSE = 7.407203029907192 , RelErr = 0.19771168338596973
iter 81 : MSE = 7.374190274012495 , RelErr = 0.19683051305557683
iter 82 : MSE = 7.350216155336532 , RelErr = 0.19619060034601807
iter 83 : MSE = 7.363448825988039 , RelErr = 0.19654380432593305
iter 84 : MSE = 7.366869056751843 , RelErr = 0.1966350964883255
iter 85 : MSE = 7.311057762399023 , RelErr = 0.19514539181654955
iter 86 : MSE = 7.277649883310041 , RelErr = 0.1942536749314623
iter 87 : MSE = 7.258231795507884 , RelErr = 0.19373537094924392
iter 88 : MSE = 7.2010325741138095 , RelErr = 0.1922086199874398
iter 89 : MSE = 7.181515966837375 , RelErr = 0.19168768634176717
iter 90 : MSE = 7.143829498086369 , RelErr = 0.19068176613847992
iter 91 : MSE = 7.1371133978690064 , RelErr = 0.19050250124262083
iter 92 : MSE = 7.115175707798954 , RelErr = 0.18991694450604463
iter 93 : MSE = 7.108892075756976 , RelErr = 0.18974922297015837
iter 94 : MSE = 7.069100536024617 , RelErr = 0.18868711460439064
iter 95 : MSE = 7.109802346963343 , RelErr = 0.18977351975962423
iter 96 : MSE = 7.069373388628742 , RelErr = 0.18869439753527023
iter 97 : MSE = 7.055289748225271 , RelErr = 0.188318479629263
iter 98 : MSE = 7.0293542306995445 , RelErr = 0.1876262136269955
iter 99 : MSE = 7.014545187725381 , RelErr = 0.18723093341070665
iter 100 : MSE = 6.9926084410536715 , RelErr = 0.1866454018551433
```

In [10]:

```
# print matrices
print("True Answer")
print(M)
print("\nEstimated Answer")
print(np.dot(U, V.T))
print("\nAccuracy (estimated - true)")
print(np.dot(U, V.T)-M)
```

```
True Answer
[[0.00513907 0.22012294 0.01979851 ... 0.15135983 0.01846026 0.02577
823]
 [0.03506236 0.55706278 0.10071643 ... 0.43790368 0.09545886 0.12311
703]
 [0.00885863 0.29268294 0.03160395 ... 0.20842955 0.02959429 0.04047
517]
 ...
 [0.00442005 0.20294327 0.01738219 ... 0.13824478 0.01618869 0.02273
235]
 [0.00514363 0.22022759 0.01981367 ... 0.1514402  0.01847451 0.02579
729]
 [0.01771638 0.41117296 0.05693121 ... 0.30710481 0.05362276 0.07129
178]]

Estimated Answer
[[-0.01766299  0.01814391  0.00814028 ... -0.01593543  0.01316297
  -0.029026  ]
 [ 0.0390221   0.54133806  0.09593972 ...  0.41298131  0.10677811
   0.14979037]
 [ 0.03438604  0.25181451  0.04483363 ...  0.20857155  0.04083663
   0.08644096]
 ...
 [ 0.03228397  0.16958962  0.02848478 ...  0.13665726  0.03225806
   0.05243882]
 [ 0.05345308  0.2210505   0.04081142 ...  0.16693716  0.03953436
   0.06662432]
 [ 0.06941458  0.3732521   0.06065793 ...  0.28840572  0.06071899
   0.10916358]]

Accuracy (estimated - true)
[[-2.28020658e-02 -2.01979028e-01 -1.16582290e-02 ... -1.67295256e-0
1
  -5.29728513e-03 -5.48042393e-02]
 [ 3.95974596e-03 -1.57247151e-02 -4.77670968e-03 ... -2.49223691e-0
2
   1.13192553e-02  2.66733330e-02]
 [ 2.55274154e-02 -4.08684305e-02  1.32296793e-02 ...  1.42009278e-0
4
   1.12423349e-02  4.59657912e-02]
 ...
 [ 2.78639119e-02 -3.33536518e-02  1.11025968e-02 ... -1.58752050e-0
3
   1.60693775e-02  2.97064748e-02]
 [ 4.83094513e-02  8.22904723e-04  2.09977556e-02 ...  1.54969518e-0
2
   2.10598416e-02  4.08270320e-02]
 [ 5.16981920e-02 -3.79208574e-02  3.72671196e-03 ... -1.86990892e-0
2
   7.09622600e-03  3.78717984e-02]]
```

```
In [11]: ### TEST 3: RANK-1 MATRIX ###

         # construct a low-rank matrix
         N = 1000  # matrix size

         # random normalized vectors
         vec1 = np.random.randn(1, N)
         vec2 = np.random.rand(N)
         vec1 = vec1/np.linalg.norm(vec1)
         vec2 = vec2/np.linalg.norm(vec2)

         M = np.outer(vec1, vec2)

         print(M)
         print("\nRank of M:", np.linalg.matrix_rank(M))
```

```
[[ 9.59351214e-04  4.78202927e-05  9.19048499e-04 ...  5.87133876e-0
4
   6.12429240e-05  5.16328613e-04]
 [-3.36505627e-04 -1.67736251e-05 -3.22368895e-04 ... -2.05945278e-0
4
  -2.14817975e-05 -1.81109359e-04]
 [ 1.26658385e-03  6.31347617e-05  1.21337417e-03 ...  7.75163748e-0
4
   8.08559964e-05  6.81683070e-04]
 ...
 [-5.35926663e-04 -2.67140642e-05 -5.13412177e-04 ... -3.27993225e-0
4
  -3.42124088e-05 -2.88438964e-04]
 [-2.11358572e-03 -1.05354834e-04 -2.02479316e-03 ... -1.29353854e-0
3
  -1.34926779e-04 -1.13754459e-03]
 [-1.61216092e-03 -8.03605667e-05 -1.54443342e-03 ... -9.86660857e-0
4
  -1.02916896e-04 -8.67674738e-04]]

Rank of M: 1
```

In [12]:
```python
col_idx = [np.random.randint(1, N) for i in range(100000)]
row_idx = [np.random.randint(1, N) for i in range(100000)]
obs = [(i, j) for i, j in zip(row_idx, col_idx)]

# set up parameters for SGD
maxiter = 100

alpha = 0.1 # step size (learning rate)
beta = 0.01 # regularization

nSamp = len(obs) # size of the observed set

np.random.shuffle(obs)

# form training and test sets (hyperparameter)
train = obs[1:(int)(np.floor(0.7*nSamp))] # training set [70%]
test = obs[(int)(np.floor(0.7*nSamp)):len(obs)+1] # test set [30%]

# start SGD

k = 10 # guess of the rank of matrix

# initialize matrices
U = np.random.normal(size=(N, k), scale=1/k)
V = np.random.normal(size=(N, k), scale=1/k)

errorVec = np.zeros((2, maxiter), dtype='float')

for itr in range(maxiter):
    np.random.shuffle(train)
    selected_updates = train[1:(int)(np.floor(0.1*len(train)))] # batc
h size (hyperparameter) [10%]

    for l in selected_updates:
        i, j = l
        error = ( np.dot(U[i, :], V[j, :].T) - M[i, j] )
        U_temp = np.copy(U[i, :])
        # update U
        U[i, :] -= alpha * (error*V[j, :] + beta*U[i, :])
        # update V
        V[j, :] -= alpha * (error*U_temp + beta*V[j, :])

    errorVec[0, itr] = MSE(U, V, M, test)
    errorVec[1, itr] = RelError(U, V, M, test)

    # print errors during iteration
    print("iter", itr+1,
          ": MSE =", errorVec[0, itr],
          ", RelErr =", errorVec[1, itr])
```

```
iter 1 : MSE = 5.486272079874367 , RelErr = 31.605341847625883
iter 2 : MSE = 5.336379978491398 , RelErr = 30.74184272919058
iter 3 : MSE = 5.192359127803759 , RelErr = 29.912166739210704
iter 4 : MSE = 5.054472104435736 , RelErr = 29.11782653033106
iter 5 : MSE = 4.924124876305148 , RelErr = 28.366921609105933
iter 6 : MSE = 4.798815722530692 , RelErr = 27.645040050186036
iter 7 : MSE = 4.675590221599263 , RelErr = 26.935162008305923
iter 8 : MSE = 4.559137131060492 , RelErr = 26.26429849987878
iter 9 : MSE = 4.446165130065232 , RelErr = 25.613488868368982
iter 10 : MSE = 4.338606873757441 , RelErr = 24.993866762565556
iter 11 : MSE = 4.234653506645783 , RelErr = 24.3950117192971
iter 12 : MSE = 4.133401580782098 , RelErr = 23.81171915139997
iter 13 : MSE = 4.036123397864225 , RelErr = 23.251318540443457
iter 14 : MSE = 3.9414919217973767 , RelErr = 22.706165090688437
iter 15 : MSE = 3.850381542332053 , RelErr = 22.18129599069761
iter 16 : MSE = 3.763115368600016 , RelErr = 21.678572609067853
iter 17 : MSE = 3.678012544552942 , RelErr = 21.188311862417514
iter 18 : MSE = 3.595273059883058 , RelErr = 20.711665852300573
iter 19 : MSE = 3.515787881866731 , RelErr = 20.25376726716825 7
iter 20 : MSE = 3.4381206050158943 , RelErr = 19.8063412555693
iter 21 : MSE = 3.3628403316420097 , RelErr = 19.37266630476029
iter 22 : MSE = 3.2910397682871344 , RelErr = 18.959037283697427
iter 23 : MSE = 3.2204292183921175 , RelErr = 18.552263697737768
iter 24 : MSE = 3.152095718584238 , RelErr = 18.15860775256595
iter 25 : MSE = 3.0859155144381734 , RelErr = 17.777356523109983
iter 26 : MSE = 3.0212642761961894 , RelErr = 17.404913367582598
iter 27 : MSE = 2.959495821022556 , RelErr = 17.0490773622332
iter 28 : MSE = 2.899534713838308 , RelErr = 16.703653135630915
iter 29 : MSE = 2.8411905556787422 , RelErr = 16.36754383652972
iter 30 : MSE = 2.785116560173094 , RelErr = 16.044512501058872
iter 31 : MSE = 2.7303658474658445 , RelErr = 15.729104339319763
iter 32 : MSE = 2.6758083485849444 , RelErr = 15.414809244694816
iter 33 : MSE = 2.6234036027760648 , RelErr = 15.1129157400319
iter 34 : MSE = 2.571870374110488 , RelErr = 14.816042875402713
iter 35 : MSE = 2.5216743137140365 , RelErr = 14.526873175989946
iter 36 : MSE = 2.473473983890701 , RelErr = 14.249200490593397
iter 37 : MSE = 2.4262177286727376 , RelErr = 13.97696643459729
iter 38 : MSE = 2.380082892485007 , RelErr = 13.711192654593503
iter 39 : MSE = 2.3347191898224238 , RelErr = 13.449861224206655
iter 40 : MSE = 2.291274749686291 , RelErr = 13.199586290355281
iter 41 : MSE = 2.2482604741687813 , RelErr = 12.951789450850013
iter 42 : MSE = 2.2069125889995918 , RelErr = 12.713592360654198
iter 43 : MSE = 2.166155690863937 , RelErr = 12.478799831324205
iter 44 : MSE = 2.1266187222475312 , RelErr = 12.251035077672217
iter 45 : MSE = 2.088560573272517 , RelErr = 12.031789515124263
iter 46 : MSE = 2.051057666815243 , RelErr = 11.815742596268421
iter 47 : MSE = 2.014162684569346 , RelErr = 11.6031978100517
iter 48 : MSE = 1.9781991089653292 , RelErr = 11.396018675572114
iter 49 : MSE = 1.9432260416544318 , RelErr = 11.194545665898438
iter 50 : MSE = 1.9090273611457231 , RelErr = 10.997533747335291
iter 51 : MSE = 1.8754856931898414 , RelErr = 10.80430674975819
iter 52 : MSE = 1.8424007992204467 , RelErr = 10.613711137897994
iter 53 : MSE = 1.8101504091957457 , RelErr = 10.42792294026393
iter 54 : MSE = 1.7787096885379874 , RelErr = 10.246799089704385
iter 55 : MSE = 1.748289147636222 8 , RelErr = 10.07155229545280 2
iter 56 : MSE = 1.7181411350091622 , RelErr = 9.897875483360279
iter 57 : MSE = 1.6889066766151273 , RelErr = 9.729461478764497
iter 58 : MSE = 1.6603097661658317 , RelErr = 9.56472026334949
iter 59 : MSE = 1.631845494571191 , RelErr = 9.4007431544685
iter 60 : MSE = 1.6040194090218247 , RelErr = 9.24044251074083
iter 61 : MSE = 1.57702658126934 , RelErr = 9.084942102425154
iter 62 : MSE = 1.550327826008949 , RelErr = 8.931135788296803
iter 63 : MSE = 1.5244721054195305 , RelErr = 8.782186032242418
iter 64 : MSE = 1.4992197659173256 , RelErr = 8.636712236776232
iter 65 : MSE = 1.4744273318069323 , RelErr = 8.493887866441382
iter 66 : MSE = 1.4501216038963218 , RelErr = 8.353867315457732
iter 67 : MSE = 1.426271886257779 , RelErr = 8.216473750581386
iter 68 : MSE = 1.4028697248891198 , RelErr = 8.08165846995704
iter 69 : MSE = 1.3800890684773288 , RelErr = 7.9504235579939335
iter 70 : MSE = 1.357621662985626 , RelErr = 7.820993223395807
iter 71 : MSE = 1.3355999909417957 , RelErr = 7.694130672128117
iter 72 : MSE = 1.313948889059091 , RelErr = 7.569402903177159
iter 73 : MSE = 1.29314157107974 9 , RelErr = 7.449536008481624
iter 74 : MSE = 1.2722810445483463 , RelErr = 7.3293625897106125
iter 75 : MSE = 1.2520340899907532 , RelErr = 7.212723839234947
iter 76 : MSE = 1.232312004482152 , RelErr = 7.099108756830627
iter 77 : MSE = 1.2128673956737441 , RelErr = 6.987092163498062
iter 78 : MSE = 1.1937432796753886 , RelErr = 6.876921866644043
iter 79 : MSE = 1.1749979236176784 , RelErr = 6.768933531826904
iter 80 : MSE = 1.156728820071205 , RelErr = 6.663688794703114
iter 81 : MSE = 1.13868951550571 , RelErr = 6.559767884623319
iter 82 : MSE = 1.1211642093106085 , RelErr = 6.458808018758777
iter 83 : MSE = 1.1037192777171387 , RelErr = 6.35831117527509
iter 84 : MSE = 1.0868485202805165 , RelErr = 6.261122036958597
iter 85 : MSE = 1.0700044928973216 , RelErr = 6.164086885258859
iter 86 : MSE = 1.053479884808046 , RelErr = 6.068891845720907
iter 87 : MSE = 1.0373238465425836 , RelErr = 5.975820064947148
iter 88 : MSE = 1.0215294040709233 , RelErr = 5.884831366912888
iter 89 : MSE = 1.0060946016234251 , RelErr = 5.79591448481124
iter 90 : MSE = 0.99095266883821 , RelErr = 5.708684767629321
iter 91 : MSE = 0.9760927378982254 , RelErr = 5.6230796079958445
iter 92 : MSE = 0.961436013600716 , RelErr = 5.538645082138389
iter 93 : MSE = 0.9469549965055725 , RelErr = 5.455222771153805
iter 94 : MSE = 0.9327747036746541 , RelErr = 5.373532873916539
iter 95 : MSE = 0.9188035303891451 , RelErr = 5.293047673534162
iter 96 : MSE = 0.9052591659902196 , RelErr = 5.215021235781069
iter 97 : MSE = 0.8919024773709389 , RelErr = 5.1380759615367895
iter 98 : MSE = 0.8787819783585451 , RelErr = 5.062491329484119
iter 99 : MSE = 0.8658386186162096 , RelErr = 4.987927162166577
iter 100 : MSE = 0.8530474672485987 , RelErr = 4.914239837565765
```

In [13]:
```
# print matrices
print("True Answer")
print(M)
print("\nEstimated Answer")
print(np.dot(U, V.T))
print("\nAccuracy (estimated - true)")
print(np.dot(U, V.T)-M)
```

```
True Answer
[[ 9.59351214e-04  4.78202927e-05  9.19048499e-04 ...  5.87133876e-0
4
   6.12429240e-05  5.16328613e-04]
 [-3.36505627e-04 -1.67736251e-05 -3.22368895e-04 ... -2.05945278e-0
4
  -2.14817975e-05 -1.81109359e-04]
 [ 1.26658385e-03  6.31347617e-05  1.21337417e-03 ...  7.75163748e-0
4
   8.08559964e-05  6.81683070e-04]
 ...
 [-5.35926663e-04 -2.67140642e-05 -5.13412177e-04 ... -3.27993225e-0
4
  -3.42124088e-05 -2.88438964e-04]
 [-2.11358572e-03 -1.05354834e-04 -2.02479316e-03 ... -1.29353854e-0
3
  -1.34926779e-04 -1.13754459e-03]
 [-1.61216092e-03 -8.03605667e-05 -1.54443342e-03 ... -9.86660857e-0
4
  -1.02916896e-04 -8.67674738e-04]]

Estimated Answer
[[ 0.06175911 -0.0018164  -0.00283631 ...  0.01472882  0.00721257
   0.01367728]
 [ 0.01646642  0.00473873 -0.0003908  ...  0.00592001 -0.00020123
   0.00401437]
 [-0.0170306  -0.0003385  -0.00150738 ...  0.00319896 -0.00196007
  -0.00041008]
 ...
 [-0.00494431 -0.00736253 -0.00344018 ... -0.00269096 -0.00492024
   0.0005428 ]
 [-0.00796658 -0.00082093  0.00212506 ...  0.00207843 -0.00096444
  -0.00526269]
 [-0.00503321  0.0116015   0.00191469 ... -0.01005529  0.00539808
   0.00145297]]

Accuracy (estimated - true)
[[ 6.07997625e-02 -1.86422360e-03 -3.75535386e-03 ...  1.41416871e-0
2
   7.15132303e-03  1.31609553e-02]
 [ 1.68029288e-02  4.75550094e-03 -6.84312203e-05 ...  6.12595850e-0
3
  -1.79747887e-04  4.19548300e-03]
 [-1.82971865e-02 -4.01630691e-04 -2.72075059e-03 ...  2.42379177e-0
3
  -2.04092266e-03 -1.09176488e-03]
 ...
 [-4.40838623e-03 -7.33581421e-03 -2.92676677e-03 ... -2.36296789e-0
3
  -4.88602367e-03  8.31235766e-04]
 [-5.85299226e-03 -7.15576252e-04  4.14985520e-03 ...  3.37196811e-0
3
  -8.29510816e-04 -4.12514751e-03]
 [-3.42105061e-03  1.16818624e-02  3.45912694e-03 ... -9.06863062e-0
3
   5.50099513e-03  2.32063991e-03]]
```

In [14]:

```
### TEST 4: LOW-RANK MATRIX WITH HIGH CONDITION NUMBER ###

# construct a low-rank matrix
N = 1000  # matrix size

tempM = np.random.randn(N,N)

U,S,V = np.linalg.svd(tempM)

# manipulate S to ensure low-rank
S[0] = 100
S[19] = 0.01
for i in range(20,N):
    S[i] = 0

M = U*S*V.T

print(M)
print("\nRank of M:", np.linalg.matrix_rank(M))
print("Condition number of M:", np.linalg.cond(M))
```

```
[[ 0.17208238  0.01705938 -0.04426555 ... -0.         -0.
  -0.        ]
 [ 0.2588888   0.01771982 -0.01113161 ...  0.          0.
   0.        ]
 [-0.04090502  0.0235689  -0.0022449  ... -0.          0.
   0.        ]
 ...
 [ 0.10131762 -0.00092461 -0.00976139 ... -0.          0.
   0.        ]
 [ 0.0939843  -0.00060227  0.00052677 ...  0.          0.
   0.        ]
 [-0.0183171   0.00171524 -0.00604721 ... -0.          0.
   0.        ]]

Rank of M: 20
Condition number of M: inf
```

```
In [15]: col_idx = [np.random.randint(1, N) for i in range(100000)]
         row_idx = [np.random.randint(1, N) for i in range(100000)]
         obs = [(i, j) for i, j in zip(row_idx, col_idx)]

         # set up parameters for SGD
         maxiter = 100

         alpha = 0.1 # step size (learning rate)
         beta = 0.01 # regularization

         nSamp = len(obs) # size of the observed set

         np.random.shuffle(obs)

         # form training and test sets (hyperparameter)
         train = obs[1:(int)(np.floor(0.7*nSamp))] # training set 70%
         test = obs[(int)(np.floor(0.7*nSamp)):len(obs)+1] # test set 30%

         # start SGD

         k = 20 # guess of the rank of matrix

         # initialize matrices
         U = np.random.normal(size=(N, k), scale=1/k)
         V = np.random.normal(size=(N, k), scale=1/k)

         errorVec = np.zeros((2, maxiter), dtype='float')

         for itr in range(maxiter):
             np.random.shuffle(train)
             selected_updates = train[1:(int)(np.floor(0.1*len(train)))] # batc
         h size (hyperparameter)

             for l in selected_updates:
                 i, j = l
                 error = ( np.dot(U[i, :], V[j, :].T) - M[i, j] )
                 U_temp = np.copy(U[i, :])
                 # update U
                 U[i, :] -= alpha * (error*V[j, :] + beta*U[i, :])
                 # update V
                 V[j, :] -= alpha * (error*U_temp + beta*V[j, :])

             errorVec[0, itr] = MSE(U, V, M, test)
             errorVec[1, itr] = RelError(U, V, M, test)

             # print errors during iteration
             print("iter", itr+1,
                   ": MSE =", errorVec[0, itr],
                   ", RelErr =", errorVec[1, itr])
```

```
iter 1 : MSE = 2.3565136449916206 , RelErr = 1.7020667060125096
iter 2 : MSE = 2.3300653522250556 , RelErr = 1.6829635878767342
iter 3 : MSE = 2.304057863686501 , RelErr = 1.6641788545727039
iter 4 : MSE = 2.2790870329398145 , RelErr = 1.6461428802316516
iter 5 : MSE = 2.254287970588023 , RelErr = 1.628230971060651 3
iter 6 : MSE = 2.23050098485324 , RelErr = 1.6110500663196086
iter 7 : MSE = 2.207378607126165 , RelErr = 1.594349195787098
iter 8 : MSE = 2.1848273119372896 , RelErr = 1.5780608077270402
iter 9 : MSE = 2.162898346108352 , RelErr = 1.562221916781491
iter 10 : MSE = 2.141364870436578 , RelErr = 1.5466686811430923
iter 11 : MSE = 2.120214642702452 , RelErr = 1.531392258480593
iter 12 : MSE = 2.09980923121665 , RelErr = 1.516653802971844
iter 13 : MSE = 2.0801782111959692 , RelErr = 1.502474676255018
iter 14 : MSE = 2.061025181202508 , RelErr = 1.4886407929925916
iter 15 : MSE = 2.042593439911437 , RelErr = 1.4753278833677974
iter 16 : MSE = 2.02443777059175 , RelErr = 1.4622143754786792
iter 17 : MSE = 2.006960484801698 , RelErr = 1.4495908516056322
iter 18 : MSE = 1.9896269735949779 , RelErr = 1.4370711734845414
iter 19 : MSE = 1.9729414776430663 , RelErr = 1.4250195449300391
iter 20 : MSE = 1.9566764401425303 , RelErr = 1.4132716058249357
iter 21 : MSE = 1.9411836664085111 , RelErr = 1.4020814587139678
iter 22 : MSE = 1.9258573652356983 , RelErr = 1.3910115516892376
iter 23 : MSE = 1.9109276938823228 , RelErr = 1.380228123128874
iter 24 : MSE = 1.896307878357878 , RelErr = 1.3696684977665983
iter 25 : MSE = 1.8824057670284773 , RelErr = 1.3596272570178582
iter 26 : MSE = 1.868519423399295 , RelErr = 1.3495974049906003
iter 27 : MSE = 1.8550337300283088 , RelErr = 1.3398569353170922
iter 28 : MSE = 1.8420550068425838 , RelErr = 1.3304826409361004
iter 29 : MSE = 1.8293496124296165 , RelErr = 1.3213057669285893
iter 30 : MSE = 1.8170393554847464 , RelErr = 1.3124142934873662
iter 31 : MSE = 1.8050324116641658 , RelErr = 1.3037418975683324
iter 32 : MSE = 1.7933665502620655 , RelErr = 1.2953158592418952
iter 33 : MSE = 1.782024970466432 , RelErr = 1.2871240435888183
iter 34 : MSE = 1.7708492951052812 , RelErr = 1.2790520576744373
iter 35 : MSE = 1.7601264494557702 , RelErr = 1.2713071423786853
iter 36 : MSE = 1.7498273377670914 , RelErr = 1.263868282372904 6
iter 37 : MSE = 1.7396333345434565 , RelErr = 1.2565053402890303
iter 38 : MSE = 1.7297594106875498 , RelErr = 1.2493735856783323
iter 39 : MSE = 1.720395076748479 , RelErr = 1.2426098985443534
iter 40 : MSE = 1.7110858554998167 , RelErr = 1.235886018298666
iter 41 : MSE = 1.7021592206428309 , RelErr = 1.2294384732063224
iter 42 : MSE = 1.693476758144714 , RelErr = 1.2231672893429677
iter 43 : MSE = 1.6848593358269344 , RelErr = 1.2169430828123093
iter 44 : MSE = 1.6766447851043471 , RelErr = 1.2110098630664858
iter 45 : MSE = 1.6686096677660256 , RelErr = 1.2052062447723473
iter 46 : MSE = 1.660721849522361 , RelErr = 1.199509017920233
iter 47 : MSE = 1.6530827201103448 , RelErr = 1.1939914144627917
iter 48 : MSE = 1.645610243762886 8 , RelErr = 1.1885941814658576
iter 49 : MSE = 1.6384354432685058 , RelErr = 1.1834119542932195
iter 50 : MSE = 1.631526486254309 , RelErr = 1.1784217410041393
iter 51 : MSE = 1.6248546129052641 , RelErr = 1.1736027689102255
iter 52 : MSE = 1.618202330354573 , RelErr = 1.1687979468916543
iter 53 : MSE = 1.6117851095662308 , RelErr = 1.1641629056848348
iter 54 : MSE = 1.6054830102142088 , RelErr = 1.1596110145859408
iter 55 : MSE = 1.599345213137557 , RelErr = 1.155177796015518 7
iter 56 : MSE = 1.5933448987616046 , RelErr = 1.150843878685304
iter 57 : MSE = 1.5875929568983043 , RelErr = 1.146689356278336
iter 58 : MSE = 1.5819591894113296 , RelErr = 1.1426201890620227
iter 59 : MSE = 1.5764787212299134 , RelErr = 1.1386617471303262
iter 60 : MSE = 1.5712678163139933 , RelErr = 1.134898005815085
iter 61 : MSE = 1.5661466738611343 , RelErr = 1.1311990982851905
iter 62 : MSE = 1.561117200682534 , RelErr = 1.127566401795496
iter 63 : MSE = 1.5561202965182366 , RelErr = 1.123957229309157
iter 64 : MSE = 1.5514118586143455 , RelErr = 1.1205564107267683
iter 65 : MSE = 1.5468646096997591 , RelErr = 1.1172720160032672
iter 66 : MSE = 1.5424441149153816 , RelErr = 1.1140791734697295
iter 67 : MSE = 1.5380678583971747 , RelErr = 1.1109182834267455
iter 68 : MSE = 1.5337615590155056 , RelErr = 1.1078079221439925
iter 69 : MSE = 1.5296441790705837 , RelErr = 1.1048340139151351
iter 70 : MSE = 1.525688136941705 , RelErr = 1.1019766370399966
iter 71 : MSE = 1.521872093630736 , RelErr = 1.0992203787505075
iter 72 : MSE = 1.5181658690237279 , RelErr = 1.096543440502346
iter 73 : MSE = 1.5145245496597637 , RelErr = 1.093913382124142
iter 74 : MSE = 1.5109216436864237 , RelErr = 1.0913110690354193
iter 75 : MSE = 1.50750945771449 , RelErr = 1.0888465095155138
iter 76 : MSE = 1.5041532860700357 , RelErr = 1.0864224081198652
iter 77 : MSE = 1.500818226195236 , RelErr = 1.0840135553693118
iter 78 : MSE = 1.4976819352231017 , RelErr = 1.0817482697617438
iter 79 : MSE = 1.4945757171444929 , RelErr = 1.079504705255147
iter 80 : MSE = 1.4915989993310446 , RelErr = 1.0773546764215636
iter 81 : MSE = 1.4886802262249128 , RelErr = 1.0752464999902875
iter 82 : MSE = 1.4857770337050966 , RelErr = 1.073149577131544
iter 83 : MSE = 1.4830118357573603 , RelErr = 1.0711523252283441
iter 84 : MSE = 1.4803381650528584 , RelErr = 1.069221181778933
iter 85 : MSE = 1.4777104145461273 , RelErr = 1.067323205648508
iter 86 : MSE = 1.4751051709176923 , RelErr = 1.0654414858246342
iter 87 : MSE = 1.4726923450306504 , RelErr = 1.0636987458161187
iter 88 : MSE = 1.4703351231619988 , RelErr = 1.0619961675731098
iter 89 : MSE = 1.4680657070186762 , RelErr = 1.0603570097995716
iter 90 : MSE = 1.4657605953506865 , RelErr = 1.0586920698014244
iter 91 : MSE = 1.4635561460535413 , RelErr = 1.0570998363926607
iter 92 : MSE = 1.4614017026015902 , RelErr = 1.055543721291292
iter 93 : MSE = 1.4592894138215329 , RelErr = 1.054018053738439
iter 94 : MSE = 1.457235082698029 , RelErr = 1.0525342479408901
iter 95 : MSE = 1.4553140448248107 , RelErr = 1.0511467174201383
iter 96 : MSE = 1.4534103989474314 , RelErr = 1.0497717488198879
iter 97 : MSE = 1.451633070680187 , RelErr = 1.0484880171191353
iter 98 : MSE = 1.4497535293689419 , RelErr = 1.04713045880614
iter 99 : MSE = 1.4480054208345141 , RelErr = 1.045867831983983
iter 100 : MSE = 1.4462939572556714 , RelErr = 1.044631673143022
```

In [16]:
```
# print matrices
print("True Answer")
print(M)
print("\nEstimated Answer")
print(np.dot(U, V.T))
print("\nAccuracy (estimated - true)")
print(np.dot(U, V.T)-M)
```

```
True Answer
```

```
[[ 0.17208238  0.01705938 -0.04426555 ... -0.         -0.
  -0.        ]
 [ 0.2588888   0.01771982 -0.01113161 ...  0.          0.
   0.        ]
 [-0.04090502  0.0235689  -0.0022449  ... -0.          0.
   0.        ]
 ...
 [ 0.10131762 -0.00092461 -0.00976139 ... -0.          0.
   0.        ]
 [ 0.0939843  -0.00060227  0.00052677 ...  0.          0.
   0.        ]
 [-0.0183171   0.00171524 -0.00604721 ... -0.          0.
   0.        ]]

Estimated Answer
[[ 1.20782728e-02 -4.39258110e-04  8.29318428e-04 ...  9.39115821e-0
4
   2.66742305e-03  3.71426322e-03]
 [-6.75267155e-03  2.11121411e-03  2.10848559e-03 ... -3.41161736e-0
3
  -8.65584423e-04 -3.21577916e-05]
 [ 8.30148478e-03  1.99275771e-03  2.82820394e-03 ...  9.70560357e-0
4
  -1.36583621e-03  6.27179544e-04]
 ...
 [-3.68469937e-03  1.53832456e-03  4.25293669e-04 ... -1.91883377e-0
3
  -3.06175342e-03  1.94102416e-03]
 [ 2.30204924e-03  4.93317719e-04 -2.38828710e-03 ...  1.11174927e-0
3
  -2.95399044e-03 -1.59106019e-03]
 [ 8.35679725e-03 -1.51024657e-04 -4.53559329e-04 ...  1.81966748e-0
3
  -3.09701502e-04  4.64844990e-05]]

Accuracy (estimated - true)
[[-1.60004108e-01 -1.74986405e-02  4.50948717e-02 ...  9.39115821e-0
4
   2.66742305e-03  3.71426322e-03]
 [-2.65641467e-01 -1.56086096e-02  1.32400981e-02 ... -3.41161736e-0
3
  -8.65584423e-04 -3.21577916e-05]
 [ 4.92065048e-02 -2.15761423e-02  5.07310673e-03 ...  9.70560357e-0
4
  -1.36583621e-03  6.27179544e-04]
 ...
 [-1.05002314e-01  2.46293119e-03  1.01866880e-02 ... -1.91883377e-0
3
  -3.06175342e-03  1.94102416e-03]
 [-9.16822547e-02  1.09558775e-03 -2.91506005e-03 ...  1.11174927e-0
3
  -2.95399044e-03 -1.59106019e-03]
 [ 2.66738946e-02 -1.86625969e-03  5.59364578e-03 ...  1.81966748e-0
3
  -3.09701502e-04  4.64844990e-05]]
```

```
In [ ]:
```