Matrix Completion - Recommendation System

Quoc Dung Cao
Department of Industrial and Systems Engineering

Abstract

In this project, we will learn about one more application of the singular value decomposition (SVD), matrix completion. We will apply the SVD concept to try to predict a user-joke matrix where the row indices are the users, the column indices are the jokes, and the matrix entries (i, j) is the rating of user i to joke j. The data matrix is large and sparse with a lot of missing values. Besides SVD, we also explore two other methods the compare their performance. One is the user-based method, where we asign average rating of all users available to the joke. Another one is the minimization of the training loss using stochastic gradient descent (SGD). SGD method appears to yield the best performance in term of test error, followed by SVD method, and then user-based method.

1 Introduction and Overview

Matrix completion problems arise in many real-world applications such as online ads display, Netflix movie recommendation, Spotify music recommendation. We will examine various methods to impute missing values from a user-joke matrix with a lot of missing values. The dataset is obtained from (*http://eigentaste.berkeley.edu/dataset/*), in which there are $24,983$ users and 100 jokes. Each user will rate a joke and give the rating in the range between -10 and 10. Some of the sample jokes are:

a. A man visits the doctor. The doctor says "I have bad news for you.You have cancer and Alzheimer's disease". The man replies ""Well,thank God I don't have cancer!"

b. Q. What's the difference between a man and a toilet?
A. A toilet doesn't follow you around after you use it.

The data given will be in the format of tuples (i, j, r) which represents user, joke, rating, accordingly. There are one training dataset and one test dataset. We will build the method from the training set and evaluate the error metrics on the test set. Virually, the data matrix $X \in \mathbb{R}^{24984 \times 100}$ will be the data we try to impute the missing values. However, in practice, storing the whole matrix of millions entries in memory is not very economical. We use two types of data structures to store the data, namely sparse matrix and dictionaries to reduce memory usage. The sparse matrix will facilitate SVD through `sparse.linalg.svds()` method, and the dictionaries will assist in the SGD method.

2 Theoretical Background

In general, we assume that although the data is large and has multiple features, they have a low-rank representation. In this case, we would like to study the low-rank representation of our users and jokes. For example, although not explicitly given, users can be characterized by three features, such as age, sense of humor, and cultural background.

Within the SVD context, this is exactly how SVD can provide to us, a factorization that is the best in the $l-2$ sense to reconstruct the matrix X using lower rank. In other words, we get the solution to the problem:

$$\arg\min_{U,S,V} ||X - USV^T||_F^2$$

where the Frobenious norm is just sum of square of all element in the matrix, and U, S, V are from SVD.

This is really nice if we want to reduce the dimension of the data, or compress the data to save space while reserving most of its information, depending on the magnitude of the singular values. However, our original matrix is not complete and we are more interested in predicting the future ratings for all the users. This brings up another optimization problem where we try to learn the low rank matrix factorization explicitly. If we can represent this properties for user i by a vector $u_i \in \mathbb{R}^3$, and similarly for joke j by $v_j \in \mathbb{R}^3$, we can assign the rating of user i to joke j as $r_{ij} = u_i^T v_j$. The optimization has a general form:

$$\arg\min_{U,V} ||X - UV^T||_F^2$$

where $X \in \mathbb{R}^{n\times m}$, $U \in \mathbb{R}^{n\times k}$ and $V^T \in \mathbb{R}^{k\times m}$, k is the low rank number that is $k \ll n, m$.

It may seem that the optimization is not much different from the SVD optimization and we still do not have full information in X. We will need to break the loss function down to the entries that we have in the training dataset T.

$$\arg\min_{u_i,v_j} \sum_{(i,j)\in T} (u_i^T v_j - X_{i,j})^2$$

In order to minimize the above loss, we can use an iterative updating scheme, gradient descent (GD).

Algorithm 1: Gradient Descent

```
$\arg\min_{\mathbf{x}} l(\mathbf{x}) = \sum_{i=1}^{n} f(\mathbf{x})$, where $\mathbf{x} \in \mathbb{R}^d$
while not converged do
    $\mathbf{x} \leftarrow \mathbf{x} - \eta \nabla l(\mathbf{x})$, where $\eta$ is a learning rate
end
```

Algorithm 2: Stochastic Gradient Descent

```
$\arg\min_{\mathbf{x}} l(\mathbf{x}) = \sum_{i=1}^{n} f(\mathbf{x})$, where $x \in \mathbb{R}^d$
while not converged do
    randomize the $i$ indices
    for $j \in \{1, 2, \cdots n\}$ do
        $\mathbf{x} \leftarrow \mathbf{x} - \eta \frac{\partial f(\mathbf{x_j})}{\partial \mathbf{x_j}}$, where $\eta$ is a learning rate
    end
end
```

Sometimes, it is costly to calculate the full gradient $\nabla l(\mathbf{x})$, we calculate an unbiased estimator of the gradient. This is called stochastic gradient descent (SGD).
Since iterative method like SGD can easily overfit the training data and not generalize well to unseen data, we need to impose some over-fitting counter measures. We put some regularization to the $l-2$ norm of the loss above, which becomes

$$\arg\min_{u_i, v_j} \sum_{(i,j)\in T} (u_i^T v_j - X_{i,j})^2 + \lambda \sum_{i=1}^{n} \|u_i\|_2^2 + \lambda \sum_{j=1}^{m} \|v_j\|_2^2,$$

where λ is the regularization hyper-parameter that requires tuning.
The metric to gauge performance is the mean squared error (MSE) and mean absolute error (MAE) on the test dataset ($Test$)which are defined as below:

- Mean squared error: $\frac{1}{|Test|} \sum_{(i,j)\in Test} (u_i^T v_j - R_{ij})^2$
- Mean absolute error: $\frac{1}{|Test|} \sum_{(i,j)\in Test} |u_i^T v_j - R_{ij}|$

3 Algorithm Implementation and Development

We will first need to process the data. To parse data to the sparse matrix, we need to have a list of (i, j, r) tuples. We also need to keep 2 dictionaries for users and jokes. The user dictionary will have keys as the user index and values as the list of the joke indices that he/she has rated and similarly for the joke dictionary. There will be another dictionary to store the ratings for faster lookup (recall that we do not store the whole matrix in memory).
As mentioned in the introduction, will will implement three different methods. We note that there are off-the-shelf libraries for matrix completion such as `surprise`, we will implement the algorithms here from scratch.

3.1 User-based recommendation

This method is simple to implement. Given a joke j, we gather all the users i's that have read joke j, take all the ratings, and average over the number of those people (Not over everyone in the dataset).

3.2 SVD

We first need to complete the original matrix. There are many ways to initialize the missing values. We can use the matrix from the user-based method, or give them the mean over rows or columns, or initilize to zeros. The current implementation gives zeros to all the missing value in the matrix formed by training data. We then perform (sparse) SVD on the training matrix X and predict the missing values based on the reconstructed matrix USV^T. We will tuned the number of components we retain from SVD to reconstruct the matrix based on the test error. A grid search with $1, 2, 5, 10, 20, 50$ components is performed to get the best low-rank reconstruction in terms of test error.

3.3 SGD

From the SGD algorithm presented above, we need to calculate the gradient of the loss function. However, as the function depends on both U and V at the same time, we will alternating calculate the partial derivatives w.r.t U and V and optimize them in turn. There are 2 ways to implement alternating SGD, as shown below

Algorithm 3: Alternating Stochastic Gradient Descent Scheme 1

```
$\arg\min_{u_i,v_j} \sum_{(i,j)\in T}(u_i^T v_j - X_{i,j})^2 + \lambda \sum_{i=1}^{n} \|u_i\|_2^2 + \lambda \sum_{j=1}^{m} \|v_j\|_2^2$
Initialize $\eta_u, \eta_v, \lambda, d$
while not converged do
    for all $m$ jokes do
        for each user in joke $j$ do
            $v_j = v_j - \eta_v((u_i^T v_j - R_{ij})u_i + 2\lambda v_j)$
        end
    end
    for all $n$ users do
        for each joke in users $i$ do
            $u_i = u_i - \eta_u((u_i^T v_j - R_{ij})v_j + 2\lambda u_i)$
        end
    end
end
```

The first scheme is to update the u, v vectors immediately for each (i, j) pair. This has the advantage that the matrix gets update more frequently, and each operation is cheap. It is also easier to compute the gradient and we have flexibility to model $l - 1$ regularization and also extra bias terms.
The second scheme is to update u_i once we go through all the jokes j rated by user i. The update is solved through least square, either through "back-slash" in `MATLAB`, or `np.linalg.solve`, or `cvxpy`. This method has less frequent updates, more computationally expensive, but can leverage some stable $Ax = b$ solvers.

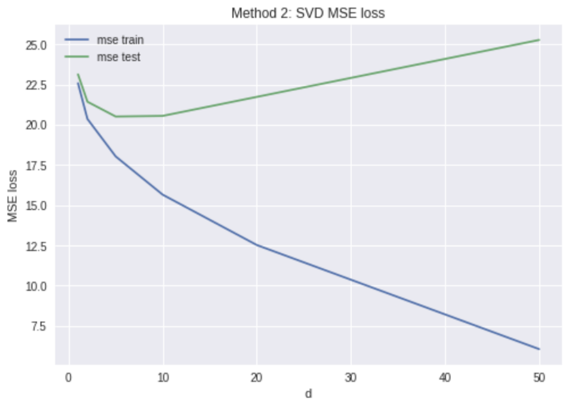
Algorithm 4: Alternating Stochastic Gradient Descent Scheme 2

```
$\arg\min_{u_i,v_j} \sum_{(i,j)\in T}(u_i^T v_j - X_{i,j})^2 + \lambda \sum_{i=1}^n \|u_i\|_2^2 + \lambda \sum_{j=1}^m \|v_j\|_2^2$
Initialize $\eta_u, \eta_v, \lambda, d$
while not converged do
    for all m jokes do
        Solve the following least square problem, with regularization:
        $U_{\text{all users i rated j}} v_j = R_{\text{all users i rated j}}$, in the form $Ax = b$
        The solution is:
        $v_j = (U^T U + \lambda I)^{-1} U^T R_{ij}$
    end
    for all n users do
        Solve the following least square problem, with regularization:
        $V_{\text{all jokes j rated by i}} u_i = R_{alljokesjratedbyi}$
        The solution is:
        $u_i = (V^T V + \lambda I)^{-1} V^T R_{ij}$
    end
end
```

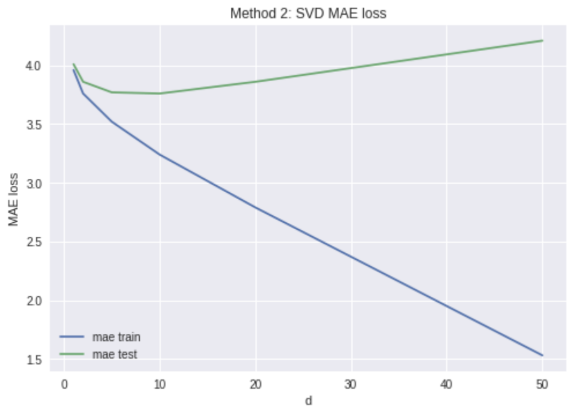
4 Computational Results

We present below the results from the prediction to the test data. Various tuning parameters are optimized over the test data such as $d = 5$, $\lambda = 0.001$, $\eta = 0.001$. SGD method appears to yield the best performance in term of test error, followed by SVD method, and then user-based method. The 2 SGD schemes yields similar results. There are also plots showing the process of tuning the hyper-parameters. The pattern is expected, in which the training error decreases as we have more overfitting parameters(e.g higher rank matrix), but the test error has a U shape and the best parameter set can yield the best test results.

Method	MSE	MAE
User-based method	24.66	4.10
SVD	20.52	3.77
SGD	**18.6**	**3.37**

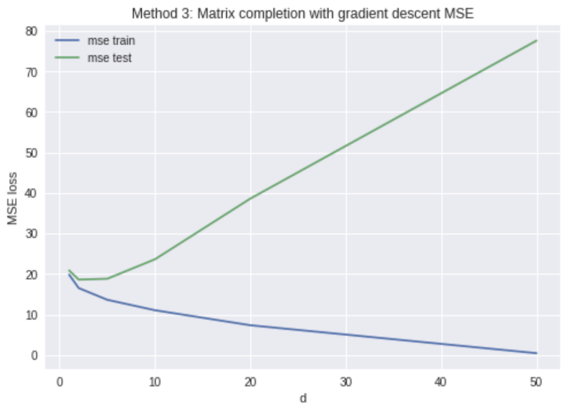


(a) MSE

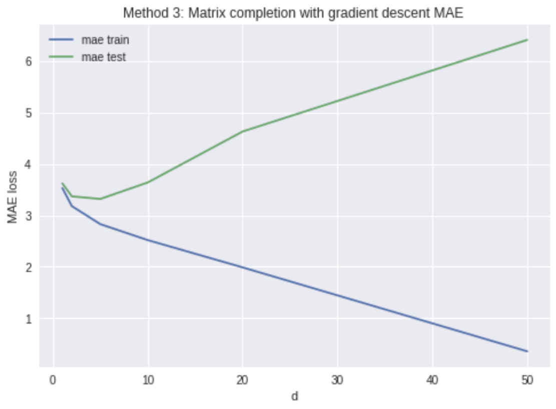


(b) MAE

Figure 1: Performance of SVD method in the test set. Best performance is at rank-5 approximation.



(a) MSE



(b) MAE

Figure 2: Performance of SGD method in the test set. Best performance is at d = 5.

5 Summary and Conclusions

We explore various methods in matrix completion application of different complexity levels. Each method has its own tradeoffs. User-based method is easy to implement but performance is not very good. SGD method gives good performance but the implementation is much more time consuming and the amount of hyper-parameter tunning is massive. Also, the iterative optimization may not be stable, depending on the learning rates, and the solution may be stuck at local optimum. SVD is also simple to implement and reasonable tunning is required but performance is just a little worse than SGD method. There have been much larger interest in the literature to study non-negative matrix factorization, which is a much harder problem. We would love to study further into this area after this project.

References

Appendix A Python functions used and brief implementation explanation

Appendix B Python code

```
# -*- coding: utf-8 -*-
"""joke_recommendation.ipynb

Automatically generated by Colaboratory.

Original file is located at
    https://colab.research.google.com/drive/190kgipcJjwRQCzhLa-2A3MPIu6_JlU8s
"""

from google.colab import drive
drive.mount('/gdrive')
!ls /gdrive

import os
import numpy as np
import matplotlib.pyplot as plt

os.chdir('/gdrive/My Drive/')

!ls

n = 24983 #users
m = 100 # jokes

#mean square error
def mse(n,m,U,V,S_mat, S_dict, average = False):
    loss = 0
    num_elements = S_mat.nnz
    if not average:
        for (k,v) in S_dict.items():
            loss += (U[k[0],:].dot(V[k[1],:]) - v)**2
    else:
        for (k,v) in S_dict.items():
            loss += (U[k[0]]*V[k[1]] - v)**2
    return (loss/num_elements)

def mae(n,m,U,V,S_mat, S_dict, average = False):
    loss = 0
    num_elements = S_mat.nnz
    if not average:
        for (k,v) in S_dict.items():
            loss += np.abs(U[k[0],:].dot(V[k[1],:]) - v)
    else:
        for (k,v) in S_dict.items():
            loss += np.abs(U[k[0]]*V[k[1]] - v)
    return (loss/num_elements)

#mean abs error
def mae1(n,m,U,V,S_mat,S_dict, average = False):
    loss = 0
    for i in range(n): #n users
        user_i = S_mat.getrow(i)
        N = user_i.size
        sum_user_i = 0
        if not average:
            for j in range(m): #j ratings
                if (i,j) in S_dict:
                    sum_user_i += np.abs(U[i,:].dot(V[j,:]) - S_dict[(i,j)]) #S_mat.
        else:
            for j in range(m): # j ratings
                if (i,j) in S_dict:
                    sum_user_i += np.abs(U[i]*V[j] - S_dict[(i,j)]) #- S_mat.todok(
        loss += sum_user_i

# import sparse module from SciPy package
from scipy import sparse
# import uniform module to create random numbers
from scipy.stats import uniform

#Parse to sparse matrix
i_index = []
j_index = []
s_rating = []

user_dict = {}
joke_dict = {}
with open('jester/train.txt', 'r') as f:
    lines = f.readlines()
    for line in lines:
        token = line.strip().split(',')
        i_index.append(int(token[0])-1)
        j_index.append(int(token[1])-1)
        s_rating.append(float(token[2]))
        if int(token[0])-1 not in user_dict:
            user_dict[int(token[0])-1] = [int(token[1])-1]
        else:
            user_dict[int(token[0])-1].append(int(token[1])-1)

        if int(token[1])-1 not in joke_dict:
            joke_dict[int(token[1])-1] = [int(token[0])-1]
        else:
            joke_dict[int(token[1])-1].append(int(token[0])-1)

S_mat = sparse.coo_matrix((s_rating, (i_index, j_index)))

#Parse to sparse matrix (TEST)
i_index = []
j_index = []
s_rating = []

with open('jester/test.txt', 'r') as f:
    lines = f.readlines()
    for line in lines:
        token = line.strip().split(',')
        i_index.append(int(token[0])-1)
        j_index.append(int(token[1])-1)
        s_rating.append(float(token[2]))

S_mat_test = sparse.coo_matrix((s_rating, (i_index, j_index)))

import scipy
from scipy.sparse.linalg import svds, eigs

d = [1,2,5,10,20,50]
U, S, VT = svds(S_mat, k=d[1])

#Parse to dictionary
S_dict = {}
with open('jester/train.txt', 'r') as f:
    lines = f.readlines()
    for line in lines:
        token = line.strip().split(',')
        S_dict[(int(token[0])-1, int(token[1])-1)] = float(token[2])

S_dict_test = {}
with open('jester/test.txt', 'r') as f:
    lines = f.readlines()
    for line in lines:
        token = line.strip().split(',')
        S_dict_test[(int(token[0])-1, int(token[1])-1)] = float(token[2])

# Method 3: This does not work, as learning rate for jokes and users should be diffe
# Otherwise the loss will blow up
# Recall that we have almost 25000 users and 100 jokes.
# One way to fix it is to have different learning rate for U and V
# Another way to fit it is to update at EVERY point in the dataset so we have A LOT

lambda_ = 0.001
lr = 0.01
```

```
d = 2
U = np.random.rand(n,d)
V = np.random.rand(m,d)
loss = []
for iter in range(5):
    #fix u update v
    for j in range(m): #m jokes m = 100
        gradvj = 0
        for user_idx in joke_dict[j]:
            gradvj += (U[user_idx,:].dot(V[j,:]) - S_dict[(user_idx, j)])*U[user_idx
        gradvj += lambda_ *V[j,:]
        V[j,:] -= lr * gradvj


    #fix v update u
    for i in range(n): #n users n = 24983
        gradui = 0
        for joke_idx in user_dict[i]:
            gradui += (U[i,:].dot(V[joke_idx,:]) - S_dict[(i, joke_idx)])*V[joke_idx
        gradui += lambda_ *U[i,:]
        U[i,:] -= lr * gradui

    loss.append(mse(n,m,U,V,S_mat, S_dict,average = False))
    print("mse_loss:", loss[iter])


  #see that the loss blows up to inf below

# We will try different learning rate here. Hopefully, the training loss will decrea
# It decreases but at a very slow rate
lambda_ = 0.001
lr_v = 0.001
lr_u = lr_v/(25000/100)
d = 2
U = np.random.rand(n,d)
V = np.random.rand(m,d)
loss = []
for iter in range(10):
    #fix u update v
    for j in range(m): #m jokes m = 100
        gradvj = 0
        for user_idx in joke_dict[j]:
            gradvj += (U[user_idx,:].dot(V[j,:]) - S_dict[(user_idx, j)])*U[user_idx
        gradvj += lambda_ *V[j,:]
        V[j,:] -= lr_v * gradvj


    #fix v update u
    for i in range(n): #n users n = 24983
```

```
            gradui = 0
            for joke_idx in user_dict[i]:
                gradui += (U[i,:].dot(V[joke_idx,:]) - S_dict[(i, joke_idx)])*V[joke_idx
            gradui += lambda_ *U[i,:]
            U[i,:] -= lr_u * gradui

        loss.append(mse(n,m,U,V,S_mat, S_dict,average = False))
        print("mse loss:", loss[iter])



#Method 3:
lambda_ = 0.001
lr = 0.001
d = 50
U = np.random.randn(n,d)
V = np.random.randn(m,d)
loss = []
for iter in range(50):
    #fix u update v
    for j in range(m): #m jokes
        U_joke_j = U[joke_dict[j],:]
        R_joke_j = []
        for user_idx in joke_dict[j]:
            R_joke_j.append(S_dict[(user_idx,j)])
        R_joke_j = np.array(R_joke_j)
        A = U_joke_j.T.dot(U_joke_j) + lambda_*np.eye(d)
        b = U_joke_j.T.dot(R_joke_j)
        V[j,:] = np.linalg.solve(A,b)


    #fix v update u
    for i in range(n): #n users
        V_user_i = V[user_dict[i],:]
        R_joke_i = []
        for joke_idx in user_dict[i]:
            R_joke_i.append(S_dict[(i,joke_idx)])
        R_joke_i = np.array(R_joke_i)
        A = V_user_i.T.dot(V_user_i) + lambda_*np.eye(d)
        b = V_user_i.T.dot(R_joke_i)
        U[i,:] = np.linalg.solve(A,b)

    #loss.append(mse(n,m,U,V,S_mat, S_dict,average = False))
    print("mse train loss:", mse(n,m,U,V,S_mat, S_dict, average = 0))
    print("mae train loss:", mae(n,m,U,V,S_mat, S_dict, average = 0))

    print("mse test loss:", mse(n,m,U,V,S_mat_test, S_dict_test, average = 0))
    print("mae test loss:", mae(n,m,U,V,S_mat_test, S_dict_test, average = 0))
```

```
S_mat.getrow(0).toarray()[S_mat.getrow(0).toarray().nonzero()]

(1,2) in S_dict

len(S_dict)

# Method 1:
U = np.ones(n)
#Predict V
V = np.zeros(m)
for j in range(m):
    if S_mat.getcol(j).nnz == 0:
        V[j] = 0
    else:
        V[j] = S_mat.getcol(j).sum()/S_mat.getcol(j).nnz
#print("MSE is: ", mse(n,m,U,V,S_mat, S_dict,average = True))
#print("MAE is: ", mae(n,m,U,V,S_mat, S_dict))

print("mse_train_loss:", mse(n,m,U,V,S_mat, S_dict, average = 1))
print("mae_train_loss:", mae(n,m,U,V,S_mat, S_dict, average = 1))

print("mse_test_loss:", mse(n,m,U,V,S_mat_test, S_dict_test, average = 1))
print("mae_test_loss:", mae(n,m,U,V,S_mat_test, S_dict_test, average = 1))

print("MAE is: ", mae(n,m,U,V,S_mat, S_dict, average = True))

S_mat[0,0]

A = sparse.csr_matrix([[1, 2, 0], [0, 0, 3], [4, 0, 5]])

print(A)

print(A.toarray())

A.getcol(1).sum()/A.getcol(1).nnz

Q = S_mat.toarray()

Q.shape

Q[0,0]

lambda_ = 0.1
n_factors = 5
m, n = Q.shape
n_iterations = 20

X = 5 * np.random.rand(m, n_factors)
Y = 5 * np.random.rand(n_factors, n)
```

```
def get_error(Q, X, Y, W):
    return np.sum((W * (Q - np.dot(X, Y)))**2)

def get_loss(S,U,V):
    return ((S.toarray()-U.dot(V.T))**2).mean()

def get_abs_loss(S,U,V):
    return (np.abs(S.toarray()-U.dot(V.T))).mean()

W = Q != 0
W[W == True] = 1
W[W == False] = 0
# To be consistent with our Q matrix
W = W.astype(np.float64, copy=False)

weighted_errors = []
for ii in range(n_iterations):
    for u, Wu in enumerate(W):
        X[u] = np.linalg.solve(np.dot(Y, np.dot(np.diag(Wu), Y.T)) + lambda_ * np.ey
                               np.dot(Y, np.dot(np.diag(Wu), Q[u].T))).T
    for i, Wi in enumerate(W.T):
        Y[:,i] = np.linalg.solve(np.dot(X.T, np.dot(np.diag(Wi), X)) + lambda_ * np.
                                 np.dot(X.T, np.dot(np.diag(Wi), Q[:, i])))
    weighted_errors.append(get_error(Q, X, Y, W))
    print(weighted_errors[ii])
    print('{}th iteration is completed'.format(ii))
#weighted_Q_hat = np.dot(X,Y)
#print('Error of rated movies: {}'.format(get_error(Q, X, Y, W)))

# Method 2
d = [1]
for e in d:
    U, S, VT = svds(S_mat, k=e)
    print("SVD {} components".format(e))
    U = U*S
    print("mse train loss:", mse(n,m,U,VT.T,S_mat, S_dict, average = 1))
    print("mae train loss:", mae(n,m,U,VT.T,S_mat, S_dict, average = 1))

    print("mse test loss:", mse(n,m,U,VT.T,S_mat_test, S_dict_test, average = 1))
    print("mae test loss:", mae(n,m,U,VT.T,S_mat_test, S_dict_test, average = 1))

U[0,:].shape

U[:,:1].dot(S[:1]).dot(VT[:,:1])

VT.shape

U.shape
```

```
S.shape

U.dot(np.diag(S)).dot(VT).shape

(U*S).shape

VT.shape

# Method 3
msetrain = [19.83, 16.52, 13.62, 11.04,7.33,0.43]
maetrain = [3.53, 3.18, 2.83, 2.52, 1.99, 0.359]
msetest = [20.87, 18.60, 18.82, 23.63, 38.62, 77.63]
maetest = [3.62, 3.37, 3.32, 3.64, 4.63, 6.41]
d = [1,2,5,10,20,50]
plt.plot(d,msetrain, label = "mse train")
#plt.plot(d,maetrain, label = "mae train")
plt.plot(d,msetest, label = "mse test")
#plt.plot(d,maetest, label = "mae test")
plt.legend()
plt.title("Method 3: Matrix completion with gradient descent MSE")
plt.xlabel("d")
plt.ylabel("MSE loss")
plt.show()

# Method 3
msetrain = [19.83, 16.52, 13.62, 11.04,7.33,0.43]
maetrain = [3.53, 3.18, 2.83, 2.52, 1.99, 0.359]
msetest = [20.87, 18.60, 18.82, 23.63, 38.62, 77.63]
maetest = [3.62, 3.37, 3.32, 3.64, 4.63, 6.41]
d = [1,2,5,10,20,50]
#plt.plot(d,msetrain, label = "mse train")
plt.plot(d,maetrain, label = "mae train")
#plt.plot(d,msetest, label = "mse test")
plt.plot(d,maetest, label = "mae test")
plt.legend()
plt.title("Method 3: Matrix completion with gradient descent MAE ")
plt.xlabel("d")
plt.ylabel("MAE loss")
plt.show()

# Method 2
msetrain = [22.59, 20.37, 18.04, 15.66, 12.53, 6.06]
maetrain = [3.96, 3.76, 3.52, 3.24, 2.79, 1.53]
msetest = [23.14, 21.44, 20.52,20.56, 21.74, 25.28]
maetest = [4.01, 3.86, 3.77, 3.76, 3.86, 4.21]
d = [1,2,5,10,20,50]
plt.plot(d,msetrain, label = "mse train")
#plt.plot(d,maetrain, label = "mae train")
```

```
plt.plot(d,msetest, label = "mse test")
#plt.plot(d,maetest, label = "mae test")
plt.legend()
plt.title("Method 2: SVD MSE loss")
plt.xlabel("d")
plt.ylabel("MSE loss")
plt.show()

# Method 2
msetrain = [22.59, 20.37, 18.04, 15.66, 12.53, 6.06]
maetrain = [3.96, 3.76, 3.52, 3.24, 2.79, 1.53]
msetest = [23.14, 21.44, 20.52,20.56, 21.74, 25.28]
maetest = [4.01, 3.86, 3.77, 3.76, 3.86, 4.21]
d = [1,2,5,10,20,50]
#plt.plot(d,msetrain, label = "mse train")
plt.plot(d,maetrain, label = "mae train")
#plt.plot(d,msetest, label = "mse test")
plt.plot(d,maetest, label = "mae test")
plt.legend()
plt.title("Method 2: SVD MAE loss")
plt.xlabel("d")
plt.ylabel("MAE loss")
plt.show()
```