
# The Experiment Report of Machine Learning

**SCHOOL:** SCHOOL OF SOFTWARE ENGINEERING

**SUBJECT:** SOFTWARE ENGINEERING

Author:
Zijie Shu
Wenjie Pan
Qiaoyuan Wen

Supervisor:
Qingyao Wu

Student ID:
201530612712
201530612590
201530612989

Grade:
Undergraduate

December 28, 2017

# Recommender System Based on Matrix Decomposition

**Abstract—**With the development of network, there is a large amount of valuable information produced by network user. In this experiment, we use matrix decomposition with gradient descent to solve the construction of recommended problem. And finally, we get a satisfactory result.

## I. Introduction

Recommender Systems attempt to suggest items (movies, books, music, news, Web pages, images, etc.) that are likely to interest the users. Typically, recommender systems are based on Collaborative Filtering, which is a technique that automatically predicts the interest of an active user by collecting rating information from other similar users or items.

The underlying assumption of collaborative filtering is that the active user will prefer those items which the similar users prefer. Based on this simple but effective intuition, collaborative filtering has been widely employed in some large, famous commercial systems, such as Amazon1. However, due to the nature of collaborative filtering, recommender systems based on this technique suffer from the following inherent weaknesses: (1) Due to the sparsity of the user-item rating matrix (the density of available ratings in commercial recommender systems is often less than 1% [19]), memory-based [10, 12, 13, 24] collaborative filtering algorithms fail to find similar users, since the methods of computing similarities, such as the Pearson Correlation Coefficient (PCC) or the Cosine method, assume that two users have rated at least some items in common. Moreover, almost all of the memory-based and model-based [8, 9, 18, 20] collaborative filtering algorithms cannot handle users who have never rated any items. (2) In reality, we always turn to friends we trust for movie, music or book recommendations, and our tastes and characters can be easily affected by the company we keep. Hence, traditional recommender systems, which purely mine the user-item rating matrix for recommendations, give somewhat unrealistic output.

Recommendations can be generated by a wide range of algorithms. While user-based or item-based collaborative filtering methods are simple and intuitive, matrix factorization techniques are usually more effective because they allow us to discover the latent features underlying the interactions between users and items. Of course, matrix factorization is simply a mathematical tool for playing around with matrices, and is therefore applicable in many scenarios where one would like to find out something hidden under the data.

## II. Methods and Theory

The mathematics of matrix factorization are shown as follow:

Having discussed the intuition behind matrix factorization, we can now go on to work on the mathematics. Firstly, we have a set U of users, and a set D of items. Let R of size |U| * |D| be the matrix that contains all the ratings that the users have assigned to the items. Also, we assume that we would like to discover $K$ latent features. Our task, then, is to find two matrixes, P (a |U| * K matrix) and Q (a |D| * K matrix) such that their product approximates R:

$$R \approx P \times Q^T = \hat{R}$$

In this way, each row of P would represent the strength of the associations between a user and the features. Similarly, each row of Q would represent the strength of the associations between an item and the features. To get the prediction of a rating of an item $d_j$ by $u_i$, we can calculate the dot product of the two vectors corresponding to $u_i$ and $d_j$:

$$\hat{r}_{ij} = p_i^T q_j = \sum_{k=1}^{k} p_{ik} q_{kj}$$

Now, we have to find a way to obtain P and Q. One way to approach this problem is the first intialize the two matrices with some values, calculate how different their product is to M, and then try to minimize this difference iteratively. Such a method is called gradient descent, aiming at finding a local minimum of the difference.

The difference here, usually called the error between the estimated rating and the real rating, can be calculated by the following equation for each user-item pair:

$$e_{ij}^2 = (r_{ij} - \hat{r}_{ij})^2 = (r_{ij} - \sum_{k=1}^{K} p_{ik}\, q_{kj})^2$$

Here we consider the squared error because the estimated rating can be either higher or lower than the real rating.

To minimize the error, we have to know in which direction we have to modify the values of $p_{ik}$ and $q_{kj}$. In other words, we need to know the gradient at the current values, and therefore we differentiate the above equation with respect to these two variables separately:

$$\frac{\partial}{\partial_{p_{ik}}} e_{ij}^2 = -2(r_{ij} - \hat{r}_{ij})(p_{ik}) = -2e_{ij}q_{kj}$$

$$\frac{\partial}{\partial_{q_{ik}}} e_{ij}^2 = -2(r_{ij} - \hat{r}_{ij})(p_{ik}) = -2e_{ij}p_{ik}$$

Having obtained the gradient, we can now formulate the update rules for both $p_{ik}$ and $q_{kj}$:

$$p'_{ik} = p_{ik} + \alpha \frac{\partial}{\partial_{p_{ik}}} e_{ij}^2 = p_{ik} + 2\alpha e_{ij} q_{kj}$$

$$q'_{kj} = q_{kj} + \alpha \frac{\partial}{\partial_{q_{kj}}} e_{ij}^2 = q_{kj} + 2\alpha e_{ij} p_{ik}$$

Here, α is a constant whose value determines the rate of approaching the minimum. Usually we will choose a small value for α, say 0.0002. This is because if we make too large a step towards the minimum we may run into the risk of missing the minimum and end up oscillating around the minimum.

A question might have come to your mind by now: if we find two matrices P and Q such that P × Q approximates R, isn't that our predictions of all the unseen ratings will all be zeros? In fact, we are not really trying to come up with P and Q such that we can reproduce R exactly. Instead, we will only try to minimise the errors of the observed user-item pairs. In other words, if we let T be a set of tuples, each of which is in the form of $(u_i, d_j, r_{ij})$, such that T contains all the observed user-item pairs together with the associated ratings, we are only trying to minimise every $e_{ij}$ for $(u_i, d_j, r_{ij}) \in$ T. (In other words, T is our set of training data.) As for the rest of the unknowns, we will be able to determine their values once the associations between the users, items and features have been learnt

Using the above update rules, we can then iteratively perform the operation until the error converges to its minimum. We can check the overall error as calculated using the following equation and determine when we should stop the process.

$$\mathrm{E} = \sum_{(u_{i,d_j,r_{ij}})\in T} e_{ij} = \sum_{(u_{i,d_j,r_{ij}})\in T} (r_{ij} - \sum_{k=1}^{K} p_{ik} q_{kj})^2$$

The above algorithm is a very basic algorithm for factorizing a matrix. There are a lot of methods to make things look more complicated. A common extension to this basic algorithm is to introduce regularization to avoid overfitting. This is done by adding a parameter β and modify the squared error as follows:

$$e_{ij}^2 = (r_{ij} - \sum_{k=1}^{K} p_{ik} q_{kj})^2 + \frac{\beta}{2} \sum_{k=1}^{K} (||P||^2 + ||Q||^2)$$

In other words, the new parameter \beta is used to control the magnitudes of the user-feature and item-feature vectors such that P and Q would give a good approximation of R without having to contain large numbers. In practice, β is set to some values in the range of 0.02. The new update rules for this squared error can be obtained by a procedure similar to the one described above. The new update rules are as follows.

$$p'_{ik} = p_{ik} + \alpha \frac{\partial}{\partial_{p_{ik}}} e_{ij}^2 = p_{ik} + \alpha(2e_{ij} q_{kj} - \beta p_{ik})$$

$$q'_{kj} = q_{kj} + \alpha \frac{\partial}{\partial_{q_{kj}}} e_{ij}^2 = p_{kj} + \alpha(2e_{ij} q_{jk} - \beta q_{kj})$$

## III. Experiment

**Dataset:**

We utilized MovieLens-100k dataset. The dataset, this experiment provided, consists 100,000 comments from 943 users out of 1682 movies. At least, each user comment 20 videos. Users and movies are numbered consecutively from number 1 respectively. The data is sorted randomly.

**Experiment Step:**

We didn't choose the ALS or SGD method which introduced in the experiment guide. We use a simple gradient descent method, which may need more time overhead.

**Using gradient descent method(GD):**

1. Read the data set and divide it (or use u1.base / u1.test to u5.base / u5.test directly). Populate the original scoring matrix $R_{nusers,nitems}$ against the raw data, and fill 0 for null values.
2. Initialize the user factor matrix $P_{nusers,K}$ and the item (movie) factor matrix $Q_{nitem,K}$, where K is the number of potential features.
3. Determine the loss function and the hyperparameter learning rate η and the penalty factor ƛ.
4. Use the gradient descent method to decompose the sparse user score matrix, get the user factor matrix and item (movie) factor matrix:
   4.1 Select all samples from scoring matrix in the training data set randomly;
   4.2 Calculate the average of all sample's loss gradient of specific row(column) of user factor matrix and item factor matrix;
   4.3 Use GD to update the specific row(column) of $P_{n_{users},K}$ and $Q_{n_{itesm},K}$;
   4.4 Calculate the $L_{validation}$ on the validation set, comparing with the $L_{validation}$ of the previous iteration to determine if it has converged.
5. Repeat step 4. several times, get a satisfactory user factor matrix P and an item factor matrix Q, Draw a $L_{validation}$ curve with varying iterations.
6. The final score prediction matrix $\hat{R}_{n_{users},n\_items}$ is obtained by multiplying the user factor matrix $P_{n_{users},K}$ and the transpose of the item factor matrix $P_{n_{users},K}$.

**Main code:**

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

X_train = np.loadtxt('u1.base')
X_test = np.loadtxt('u1.test')
X_train = X_train[:,0:3]
X_test = X_test[:,0:3]
num_user = 943
num_item = 1682

#Populate the original scoring matrix
rating_train = np.zeros((num_user,num_item))
for i in range(80000):
    user_id = int(X_train[i,0])
    item_id = int(X_train[i,1])
    rating = X_train[i,2]
    rating_train[user_id-1,item_id-1] = rating
rating_test = np.zeros((num_user,num_item))
for i in range(20000):
    user_id = int(X_test[i,0])
```

```
    item_id = int(X_test[i,1])
    rating = X_test[i,2]
    rating_test[user_id-1,item_id-1] = rating

def matrix_factorization(R_train, R_valid, P, Q,
K, iterations, learning_rate,lamda):
    R = R_train
    loss = []
    epoch_set=[]
    e_pre = 100000
    for epoch in range(iterations):
        for i in range(num_user):
            for j in range(num_item):
                if R[i][j] > 0:
                    eij = R[i][j] -
np.dot(P[i,:],Q[:,j])
                    for k in range(K):
                        P[i][k] = P[i][k] +
learning_rate * (2 * eij * Q[k][j] - lamda *
P[i][k])
                        Q[k][j] = Q[k][j] +
learning_rate * (2 * eij * P[i][k] - lamda *
Q[k][j])
        eR = np.dot(P,Q)
        e = 0
        for i in range(num_user):
            for j in range(num_item):
                if R_valid[i][j] > 0:
                    e = e + pow(R_valid[i][j] -
np.dot(P[i,:],Q[:,j]), 2)
                    for k in range(K):
                        e = e + (lamda/2) *
(pow(P[i][k],2) + pow(Q[k][j],2))
        if e < e_pre:
            e_pre = e
        elif e - e_pre > 0.01:
            learning_rate *= 0.9
        loss.append(e)
        print("step:",epoch,"  loss", e)
        if e < 0.001:
            break
    return P, Q, loss

K=3
iterations = 100
learning_rate = 0.001
lamda = 0.02
P = np.random.rand(num_user,K)
Q = np.random.rand(K,num_item)

nP,nQ,n_loss =
matrix_factorization(rating_train,rating_test,P,
Q,K,iterations,learning_rate,lamda)
```

```
R_pre = np.dot(nP,nQ)
print(R_pre)
#paint
plt.xlabel('step')
plt.ylabel('loss')
plt.plot(n_loss)
plt.show()
```

**Predicted results:**

```
 [[        3.57934734         3.86837589
3.28464141  ...,  2.27160216  2.34056495
3.24321114]   [  4.18340881   3.27319768
3.15928619  ...,  1.93094063  2.07415686
2.8010578   ]   [  3.71502986   3.43633892
2.84445428  ...,  2.07999783  1.85814142
2.76558781]     ...,     [     4.24285428
3.55831678  3.37880194  ...,  2.08642141
2.27743101   3.05541878]   [   4.57100716
3.7752209   3.40104948   ...,  2.26215086
2.20295244   3.12772893]   [   3.77576654
3.63685538  3.00898407  ...,  2.18857689
2.00678992 2.94763202]]
```

**Loss curve with varying iterations:**

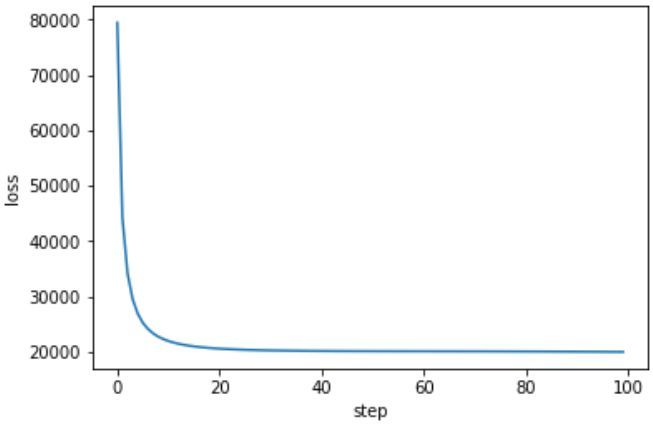


Figure 1 Loss curve with varying iterations

## IV. CONCLUSION

Through this experiment, we explore the construction of recommended system, understand the principle of matrix decomposition. Besides, we are familiar to the use of gradient descent and construct a recommendation system under small-scale dataset, cultivate engineering ability.