华南理工大学

South China University of Technology

The Experiment Report of Machine Learning

SCHOOL: SCHOOL OF SOFTWARE ENGINEERING

SUBJECT: SOFTWARE ENGINEERING

Author: Xianlin Hu, Jin Cao,
Pinchao Huang

Supervisor:
Mingkui Tan

Student ID：201530781104，
201530611111, 201530611760

Grade:
Undergraduate

December 9, 2017

Linear Regression, Linear Classification and Gradient Descent

Abstract—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.

I. Introduction

There is probably no need to say that there is too much information on the Web nowadays. Search engines help us a little bit. What is better is to have something interesting recommended to us automatically without asking. Indeed, from as simple as a list of the most popular bookmarks on Delicious, to some more personalized recommendations we received on Amazon, we are usually offered recommendations on the Web.

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.

In this tutorial, we will go through the basic ideas and the mathematics of matrix factorization, and then we will present a simple implementation in Python. We will proceed with the assumption that we are dealing with user ratings (e.g. an integer score from the range of 1 to 5) of items in a recommendation system.

II. Methods and Theory

matrix factorization:
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 matrics matrices P (a |U| * K matrix) and Q (a |D| * K matrix) such that their product approximates R:

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

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:

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

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, \alpha is a constant whose value determines the rate of approaching the minimum. Usually we will choose a small value for \alpha, 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 \mathbf{P} and \mathbf{Q} such that \mathbf{P} \times \mathbf{Q} approximates \mathbf{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 \mathbf{P} and \mathbf{Q} such that we can reproduce \mathbf{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.

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

Regularization:
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 \beta 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, \beta 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.

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

III. EXPERIMENT

1. Motivation
   (1) Explore the construction of recommended system.
   (2) Understand the principle of matrix decomposition.
   (3) Be familiar to the use of gradient descent.
   (4) Construct a recommendation system under small-scale dataset, cultivate engineering ability.

2. data set:
   Utilizing MovieLens-100k dataset.
   u.data -- Consisting 10,000 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

3. Experiment Step
   Using stochastic gradient descent method(SGD):
   (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 Qnitems,K , where is the number of potential features.
   (3) Determine the loss function and hyperparameter learning rate η and the penalty factor λ.
   (4) Use the stochastic gradient descent method to decompose the sparse user score matrix, get the user factor matrix and item (movie) factor matrix:
   4.1 Select a sample from scoring matrix randomly;
   4.2 Calculate this sample's loss gradient of specific row(column) of user factor matrix and item factor matrix;
   4.3 Use SGD to update the specific row(column) of Pn)users,K and Qn_item,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 R_nusers,nitems is obtained by multiplying the user factor matrix P_nusers,K and the transpose of the item factor matrix Q_nitem,K.

4. Code:

```
def matrix_factorization(R, R_t, P, Q, K, steps=500, alpha=0.0002, beta=0.02):
    Q = Q.T
    #L_v = []
    e_list = []
    for step in range(steps):
      # L_v.append(loss(np.dot(P, Q), P, Q, lam_p, lam_q))
        for i in range(len(R)):
            for j in range(len(R[i])):
                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] + alpha * (2 * eij * Q[k][j] - beta * P[i][k])
                        Q[k][j] = Q[k][j] + alpha * (2 * eij * P[i][k] - beta * Q[k][j])
        if step%10 == 0:
            e = 0
            for i in range(len(R_t)):
                for j in range(len(R_t[i])):
                    if R_t[i][j] > 0:
                        e = e + pow(R_t[i][j] - np.dot(P[i,:],Q[:,j]), 2)
                        for k in range(K):
                            e = e + (beta/2) * (pow(P[i][k],2) + pow(Q[k][j],2))
            e = e/(R.shape[0]*R.shape[1])
            e_list.append(e)
            print(e)
            if e < 0.001:
```

```
                break
        return P, Q.T, e_list
riting_b , riting_t = makeRitingMatrix('1')
N = len(riting_b)
M = len(riting_b[0])
K = 2

P = np.random.rand(N,K)
Q = np.random.rand(M,K)


nP, nQ, L_v = matrix_factorization(riting_b, riting_t, P, Q, K)

plt.xlabel('times')
plt.ylabel('loss')
plt.plot(np.arange(len(L_v)),np.array(L_v),label="L_validation")
plt.legend(bbox_to_anchor=(1.0, 1), loc=1, borderaxespad=0.)
plt.show()
```

5. Selection of validation: hold out

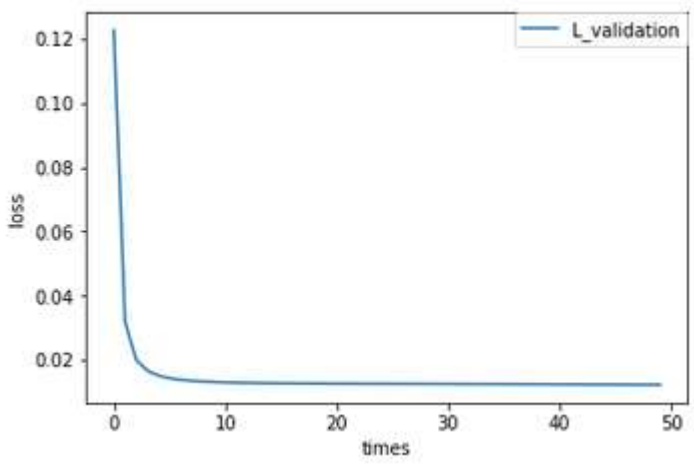
6. The initialization method of model parameters:
   Random.
7. The selected loss function and its derivatives:
   Loss:

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

   Derivatives:

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

8. Experimental results and curve:
   Epoch=500
   Study rite=0.0002
   Regularization parameter:0.02
   Assessment Results (based on selected validation): Loss curve converges
   Predicted Results (Best Results): Loss curve converges below 0.013
   Loss and accurancy curve:



9. .Results analysis:
   Time complexity per iteration of SGD:O(|Ω|k)
   SGD is scalable to large-scale datasets.

IV. CONCLUSION

Through this experiment, I understand the difference and connection between ALS and SGD, and practice on larger data. ALS is not scalable to large-scale datasets while SGD is scalable to large-scale datasets.
During the experiment I encountered many problems, such as code bugs, not familiar with the python library, do not use tools. It took me a lot of time , but eventually I finished it and learned a lot from it