Implementing Linear Kernel Matrix Factorization for a recommender systems with online-updates

Claudio Cimarelli

University of Luxembourg
6, rue Richard Coudenhove-Kalergi, L-1359 Luxembourg

Abstract. In this paper i will present an implementation of Regularized Matrix Factorization for Recommender Systems. In particular this method has been adapted to generate prediction for new users coming in the system with few updates to the model. In most of the proceedings i followed the guidelines written in [1].

1 Introduction

In this paper every results is based on the dataset of Movielens with 1M ratings. Loading the data with python i build up a dense matrix with 6040 rows, corresponding to the users, and with 3952 columns, one per item. From this "batch matrix" 10 % of the users are taken to test online updates while the rest is used to train the batch phase of matrix factorization.

```
try:
    train_mask = np.load('data/train_mask.npy')
    test_mask = np.load('data/test_mask.npy')
except:
    train_mask = np.ones((N, M))
    test_mask = np.zeros((N,M))
    users_test = np.random.choice(N, int((N-1)/10), replace=False)
    test_mask[users_test, :] = 1
    train_mask -= test_mask
    np.save('data/train_mask', train_mask)
    np.save('data/test_mask', test_mask)
```

With this randomly chosen users we can separate the "batch_matrix" in two part with the aid of two mask to build them.
The "train_mask" is used in the first phase of the process, in which we train a decomposition of the matrix with the 90% of users in two features vectors.
The "test_mask" will be useful in the second phase, when we'll learn some changes on the user feature vector to predict ratings for new users. These are the random 10% chosen from the totality, corresponding to 603 user.

2 Phase 1: Batch learning matrix factorization

The first thing part of the process consist on learning two features vectors, U and V, such that their product $U \times V^T$ should approximate the train matrix of ratings.
This means that the resulting approximated matrix, let's call $\hat{R}$, should be as similar as possible to the original R, matrix of ratings. Then we are expected to minimize the error between these two, we'll go further with code in a later section, which is regulated by this function : $E(R, \hat{R}) = \sum_{r_{u,i} \in R} (r_u, i - \hat{r}_{u,i})^2$.
Thus the task is to find the features vectors that optimize this error formula. But to avoid overfitting, in which we could incur with an high number of features, a regularization term is added. We use Tikhonov regularization which is controlled by a parameter β, that will be decided by trial and error as most of the hyperparameter involved in the training.
So the task of optimization is to find U, V that minimize: $E(R, \hat{R}) + \beta(||U||_K^2 + ||V||_K^2)$

Finally we have to choose the K value, which is the number features (or the number of columns of both vectors). With an higher value, let say K=40, we could achieve better results finding more latent variable, but for now we'll remain stick to the evaluation proposed in [1], for comparison reasons, and choose K=10.

```
N = len(batch_matrix)
M = len(batch_matrix[0])
K = 10
u_batch, v_batch = train(batch_matrix, train_mask, N, M, K)
```

2.1 The Kernel choice

Before starting to introduce more in deep the implementation code and all the decision made on the parameters, is important to give an overview on the methodology whereby we decompose the matrix.
The interaction between the vector is affected by the choice of an appropriate function, the Kernel. The right one could permit, without going much deeper, a better separation of the solution space, in the case the values for U and V, possibly leading to a lower error of approximation.
Then this function, in the case of the following code, is linear. This mean a simple dot product between the vector plus a value of *bias* will get us the resulting approximated matrix $\hat{R}$ which contains the predictions. Thus we have $\hat{R} = bias + U \times V^T$.
The *bias* value is the global average of the ratings contained in the matrix used for training these vectors.
The *non_zero_matrix*() method provides a mask of ones where the passed matrix has non-zero values.

```
nz_ratings = non_zero_matrix(train_mat)
bias = np.sum(train_mat) / np.sum(nz_ratings)
```

Then, using the average as *bias*, we could initialize the features vectors with small random numbers around zero, in the semi-open interval $[-0.05, 0.05)$

```
u_b = np.random.uniform(-0.05, 0.05, (N, K))
v_b = np.random.uniform(-0.05, 0.05, (M, K))
```

2.2 Matrix Factorization: optimization by Gradient Descent

To train the approximated matrix $\hat{R}$, as previously said, we have to minimize an error function. Thus we use the gradient descent technique to find a local optimum of the error formula.
Then for a specified number of *epoch* (it's been chosen as 200*epochs*) we update the feature vectors going in the opposite direction of the gradient of the error. So we have to calculate the gradient with respect to every feature and subtract it to its precedent value.
This means that we calculate a δ that's the variation of the vectors. This value is affected by some hyper-parameter learned by trial and error: $\alpha = 0.23$ the learning rate, $\beta = 0.03$ the regularization factor. Nevertheless the variation depends on the partial derivative with respect to every $u_{i,k}$, in case of the user feature vector variation, and to every $v_{i,k}$, in case of the items.
If the error for every ratings is $e_{ij}^2 = (r_{ij} - \hat{r}_{ij})^2 = (r_{ij} - \sum_{k=1}^{K} u_{ik}v_{kj})^2$
we have for every u_{ik}: $\frac{\partial}{\partial u_{ik}} e_{ij}^2 = -2(r_{ij} - \hat{r}_{ij})(v_{jk}) = -2e_{ij}v_{jk}$
and for every v_{ik}: $\frac{\partial}{\partial v_{jk}} e_{ij}^2 = -2(r_{ij} - \hat{r}_{ij})(u_{ik}) = -2e_{ij}u_{ik}$
Then finally if we add these deltas to the vectors and adding the derivation of the regularization terms:
$\delta\_u_{i,k} = \alpha(2e_{ij}v_{jk} - \beta u_{ik})$
$\delta\_v_{j,k} = \alpha(2e_{ij}u_{ik} - \beta v_{jk})$

Vectorizing the code, to avoid iterating over every $r_{i,j}$, this results in:

```
delta__u = np.dot(2 * alpha * err, alpha * v_ahead) - alpha * beta *
    u_ahead
delta__v = np.dot(2 * alpha * err.T, alpha * u_ahead) - alpha * beta *
    v_ahead
```

2.3 Further techniques: Nesterov Momentum and parameter annealing

An approach widely used to rapidly converge to a local optimum is to use momentum. This technique derives from physical considerations adapted to the local search. We have a sort of acceleration represented by the gradient that increase the speed in every epoch. The position, the points $u_{i,k}, v_{j,k}$, is moved by adding this speed. Thus the gradient acts indirectly on the variations of the vectors.

The μ parameter acts as a friction to slow down the velocity and to avoid skipping the minimum. For this reason it has to be chosen carefully, accordingly with the learning rate.
The range within they fall are: α starts from 0.23 ending with a minimum of 0.01. μ start from 0.7 ending with 0.999. Then we anneal the two parameters with the following code through the epochs:

```
      alpha = max(alpha0 / (1 + (epoch / 250)), 0.01)
      mu = min(0.999, 1.2 / (1 + np.exp(-epoch / 100)))
```

The Nesterov momentum implies to calculate the gradient in the position after the momentum step. We call this *u_ahead* and *v_ahead*, as could be noticed in the previous snippet of code.

```
     u_ahead = u + (mu * vel_u)
     v_ahead = v + (mu * vel_v)
```

After calculating the deltas we can update the velocity and the vectors consequently:

```
     vel_u = (mu * vel_u) + delta__u
     vel_v = (mu * vel_v) + delta__v
     u += vel_u
     v += vel_v
```

2.4 Results and final code

The following utility method are used as for the library numpy (alias np).

```
import numpy as np

def non_zero_matrix(r):
   with np.errstate(divide='ignore', invalid='ignore'):
      nz_r = np.nan_to_num(np.divide(r, r))
   return nz_r


def calc_rmse(real_values, prediction):
   mask = non_zero_matrix(real_values)
   error = (real_values - prediction) * mask
   error **= 2
   RMSE = (np.sum(error) / np.sum(mask)) ** (1 / 2)
   return RMSE
```

The complete method is as follow.

```
def matrix_factorization(ratings, u, v, epochs= 200, alpha0=0.023, beta=0.03):
   y = np.zeros(epochs)
   nz_ratings = non_zero_matrix(ratings)
   bias = np.sum(ratings) / np.sum(nz_ratings)
### initialization of velocity vectors
   vel_u = np.zeros_like(u)
   vel_v = np.zeros_like(v)

   f = (np.dot(u, v.T) + bias)
   err = ratings - f
   for epoch in range(epochs):
```

```
### parameters annealing
    alpha = max(alpha0 / (1 + (epoch / 150)), 0.01)
    mu = min(0.999, 1.4 / (1 + np.exp(-epoch / 80)))
 ### calc ahead postions
    u_ahead = u + (mu * vel_u)
    v_ahead = v + (mu * vel_v)
### calc gradients
    delta__u = np.dot(2 * alpha * err, alpha * v_ahead) - (alpha * beta *
        u_ahead)
    delta__v = np.dot(2 * alpha * err.T, alpha * u_ahead) - (alpha * beta
        * v_ahead)
### update velocity
    vel_u[rows, :] = (mu * vel_u) + delta__u
    vel_v[cols, :] = (mu * vel_v) + delta__v
### calc new postions
    u += vel_u
    v += vel_v

    f = (np.dot(u, v.T) + bias) * nz_ratings
    r_m_s_e = calc_rmse(ratings, f)
    y[epoch] = r_m_s_e
    err = ratings - f

    if epoch >0 and (y[epoch]>y[epoch -1]):
       u[...] = u_prev
       v[...] = v_prev
       vel_u[...] = np.zeros_like(u)
       vel_v[...] = np.zeros_like(v)
    else:
       u_prev[...] = u
       v_prev[...] = v

plt.plot(np.arange(epochs), y)
plt.show()
```

During the epochs we calculate the RMSE between R and $\hat{R}$ and saved in an array. Then plotting it we can see that it converge nearly to 0.75.

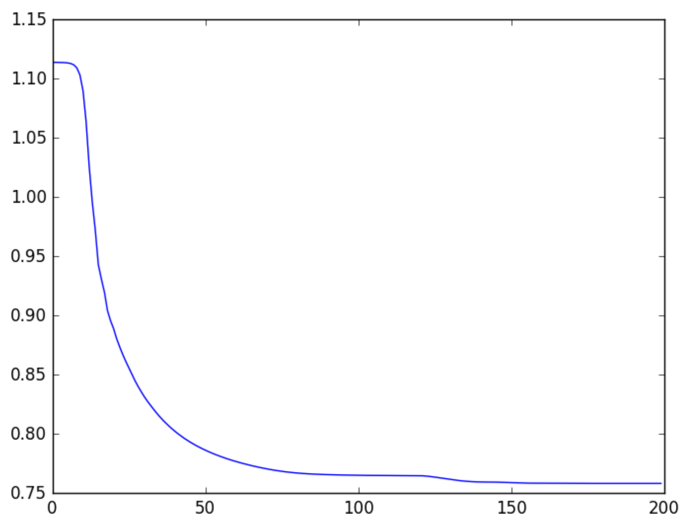

Fig. 1.

3 Phase 2: Online Updates

The main problem is to adapt the previous learned model to predict ratings for user not present at the moment of batch training. So we simulate the situation in which new users are coming in the system, ratings some of the items in the dataset, and trying to predict other items.
In particular, as anticipated before, we will insert 10% of the users in the Movielens dataset one at time, using a part of their ratings to train the online update method and the rest to evaluate it.
We'll use $min(m, |C(u,\cdot)|/2)$ random ratings for every new user u as training set, where $|C(u,\cdot)|$ is the length of the new user u profile of ratings and m is equal to 50, the rest to calculate the RMSE of the learned model.
The evaluation model will be explained deeply in a later section.

3.1 Learning new user-features vectors

Given a profile of ratings, the ones in the training set, for each user we should learn a vector u_i of dim $1 \times K$ such that $u_i \times V^T$ should be an approximation of the given profile plus giving us a prediction for the items not rated by the user.
Then the algorithm is very similar with respect as what we have done for the batch training. The model il trained with the gradient descent method searching a minimization of the error function.

```
def user_update(u_i, v, bias, profile, epochs=200, alpha0=0.02, beta=0.03):
   profile = np.reshape(profile, (1, len(profile)))
   u_i = np.reshape(u_i, (1, len(u_i)))
   nz_profile = non_zero_matrix(profile)
   vel_u = np.zeros_like(u_i)
```

```
    u_prev = np.zeros_like(u_i)
    u_prev[...] = u_i
    rmse_prev = 0
    f = (np.dot(u_i, v.T) + bias) * nz_profile
    err = profile - f
    for epoch in range(epochs):

        alpha = max(alpha0 / (1 + (epoch / 150)), 0.01)
        mu = min(0.8, 1.2 / (1 + np.exp(-epoch / 100)))

        u_ahead = u_i + (mu * vel_u)

        delta__u = np.dot(2 * alpha * err, alpha * v) - (alpha * beta *
             u_ahead)

        vel_u *= mu
        vel_u += delta__u
        u_i += vel_u

        f = (np.dot(u_i, v.T) + bias) * nz_profile
        err = profile - f

        rmse_tot = np.sum(err**2)/np.sum(nz_profile)
        if epoch > 0 and (rmse_tot > rmse_prev):
            u_i[...] = u_prev
            vel_u[...] = np.zeros_like(u_i)
        else:
            u_prev[...] = u_i

        rmse_prev = rmse_tot

    return u_i
```

The same techniques to anneal the learning rate over the epochs is used and also the Nesterov Momentum to accelerate the descent through the minimum. To assure the decreasing monotonicity of the error, in every epoch we check that the RMSE is lower than in the previous; if it's bigger we restore u_i to its precedent value.

3.2 Evaluation protocol

To evaluate the model we train with the precedent method every user profile, but adding one new rating at a time until the maximum. In fact, as explained before, we have for every new user $min(m, |C(u, \cdot)|/2)$ new ratings to train, where $m = 50$. Then the biggest profiles are of 50 ratings.
Before starting we shuffle the users profile, to avoid correlating the results with the order of the items.

```
   shuffled_profiles = []
    for us in unk_user:
        shuffled_profile = np.random.permutation(np.nonzero(ratings[us, :])[0])
        shuffled_profiles.append(shuffled_profile)
```

So we loop for 50 times increasing the profiles length of 1 and in every iteration we consider one new user and it's profile, the length of which is defined by m.

```
for m in range(50):
    for index, i in enumerate(unk_user):
        profile_u = shuffled_profiles[index][:m + 1]
```

To speed-up the process, not every time the profile is trained. In fact only if the error on the last inserted rating in the profile is high enough we have more chances to enter the method for the gradient descent. The probability depend on $tanh(err_{ij}^2)$. Moreover if there are not new ratings we do not train the profile again.

```
     err_rij = ratings[i, profile_u[-1]] - (np.dot(n_u[i, :],
         n_v[profile_u[-1], :].T) + bias)
       prob = np.tanh(err_rij ** 2)
       if prob > min(0.9,rnd.random()) and len(profile_u) == m + 1:
          u_i = user_update(n_u[i, :], n_v[profile_u, :], bias, ratings[i,
               profile_u], epochs=10+(4*m))
          n_u[i, :] = u_i
```

After we calculate the squared error on training and test set, just for the profile with a new rating at step m. We can also control that rmse is monotonically decreasing on training set and if not adjust u_i vector to its precedent value.

```
  if len(profile_u) == m + 1:
      nz_i = non_zero_matrix(test_ratings[i, :])
      err = (test_ratings[i, :] - ((np.dot(n_u[i, :], n_v.T) + bias) *
          nz_i)) ** 2
      se[index] = np.sum(err)
      se_len[index] = np.sum(nz_i)
      nz_i = non_zero_matrix(ratings[i, :])
      err = (ratings[i, :] - ((np.dot(n_u[i, :], n_v.T) + bias) * nz_i)) ** 2
      te[index] = np.sum(err)
      te_len[index] = np.sum(nz_i)
      if m>0 and te[index]/te_len[index] > te_prev[index]/te_len_prev[index]:
          n_u[i, :] = n_u_prev[i, :]
          te[index] = te_prev[index]
          te_len[index] = te_len_prev[index]
```

At the end of the iteration we sum up the errors, divide them by the sum of the profiles length to calculate the average squared error and at the end we get the root square ot the total finding the rmse. This will be plotted at the end of the process showing how is evolving the training and how the prediction are affected after inserting more ratings in every user profile.

```
      rmse_tot = (np.sum(te) / np.sum(te_len)) ** (1 / 2)
      y[m] = rmse_tot
      rmse_test = (np.sum(se) / np.sum(se_len)) ** (1 / 2)
      y_test[m] = rmse_test
   plt.plot(np.arange(50), y_test)
   plt.plot(np.arange(50), y)
   plt.show()
```

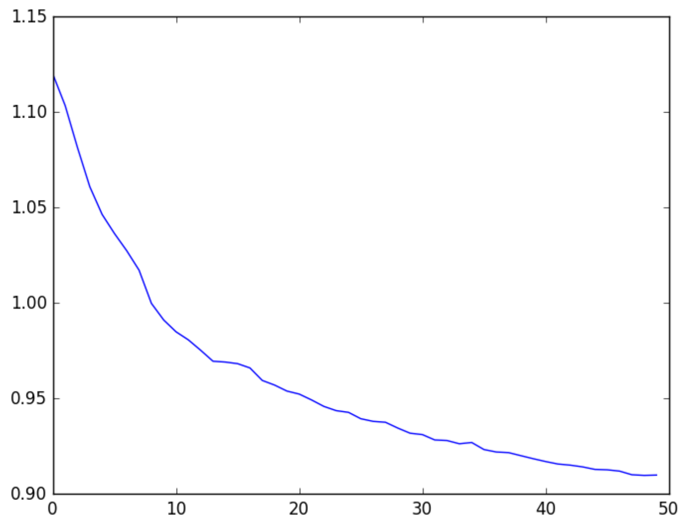

Fig. 2. RMSE Online Updates