**SD211 TP Recommendation System**

**WEI CHEN**

**weichen@whu.edu.cn**

# **Model presentation**

**Question 1.1 Obtaining the data**

filename = 'ml-100k/u.data'

R, mask=tools.load\_movielens(filename, minidata=**False**)

minidata is an option of the load\_movielens function which make the function only return the first 100 lines and 200 columns of R and mask. This option allows to reduce the volume of data, which will be used to limit the time cost of following processing.

**Question 1.2 Data dimension**

print(R.shape)*#get the number of users and films*

print(np.sum(mask))*#get the total number of rating*

The data set consists of 100,000 ratings {1,2,3,4,5} from 943 users on 1682 movies.

**Question 1.3 Convexity and lipschitz**

The objective function is written as

Since the Frobenius norm can be rewritten as

Thus

And is twice differentiable, and we calculate its Hessian matrix

*H* is symmetric, and its determinant is not necessarily positive, so *H* is not positive definite. Thus is not convex.

If the gradient is lipschitzien, it exits a constant C such that for all there is

and

Since we have and , so has no upper bound. Thus the gradient is not lipschitzien.

# **Find *P* with fixed**

**Question 2.1 Gradient of**

The matrix is positive definite, so is convex.

And is constant when is fixed, so has its upper bound, and there exist a constant such that

where

**Question 2.2 Gradient of**

**def** objective(P, Q0, R, mask, rho):

tmp = (R - Q0.dot(P)) \* mask

val = np.sum(tmp \*\* 2)/2. + rho/2. \* (np.sum(Q0 \*\* 2) + np.sum(P \*\* 2))

grad\_P = np.transpose(Q0).dot((Q0.dot(P)-R)\*mask) + rho\*P

**return** val, grad\_P

We want to verify the function defined above is correct, with the help of the function scipy.optimize.check\_grad.

**def** g\_PQ(P,Q):

P=P.reshape(vt.shape)

**return** tools.total\_objective(P, Q, R, mask, rho)[0]

**def** grad\_P(P,Q):

P=P.reshape(vt.shape)

**return** np.ravel(tools.total\_objective(P, Q, R, mask, rho)[1])

**def** grad\_Q(Q,P):

Q=Q.reshape(u.shape)

**return** np.ravel(tools.total\_objective(P, Q, R, mask, rho)[2])

u,s,vt = svds(R,7)

Q0=u

P0=vt

P0=np.ravel(P0)

rho=0.2

check\_grad(g\_P, grad\_P, P0, Q, epsilon=1e-6)

This function will return the 2-norm of the difference between grad\_P(P0, Q) and the finite difference approximation of *grad* using g\_p. The result we obtain is

0.021928157886692275

This difference is small enough, which indicates the gradient of P calculated by function grad\_P is verified.

**Question 2.3 Gradient descent with invariant pace**

Use the gradient of *g(P)*defined in the previous question, update with iteration as following pattern:

|  |
| --- |
|  |

where is the pace length, and in the case of invariant pace it is constant.

**def** gradient(g,P0,gamma,epsilon):

P=P0

grad\_P=tools.objective(P, Q0, R, mask, rho)[1]

G=list([])

C=list([])

**while** np.sqrt(np.sum(grad\_P\*\*2))>epsilon:

P=P-gamma\*grad\_P

grad\_P=tools.objective(P, Q0, R, mask, rho)[1]

G.append(np.sqrt(np.sum(grad\_P\*\*2)))

C.append(tools.objective(P, Q0, R, mask, rho)[0])

**return** P,G,C

**Question 2.4 Minimise with Gradient descent**

we set the pace length as

where is the lipschitz constant of . We’ve proved its convergence in Theorem 3.3.1 in the course.

u,s,vt = svds(R,7)

Q0=u

P0=vt

rho=0.2

L0=np.sqrt(np.sum(np.transpose(Q0).dot(Q0)))+rho

gamma=1/L0

P\_opt,G,C=gradient(0,P0,gamma,1)

plt.figure()

plt.plot(G,linestyle='--',marker='o')

plt.xlabel('iteration')

plt.ylabel('F-norm of grad\_P')

plt.grid()

plt.figure()

plt.plot(C,linestyle='--',marker='s')

plt.xlabel('iteration')

plt.ylabel('Objective value')

plt.grid()

plt.show()

In order to verify the convergence of this algorithm, we show the evolution of the norm of the gradient and the evolution of the objective value for each iteration. From the figure below, we see both and are decreasing with the evolution, which verifies the convergence of the algorithm.

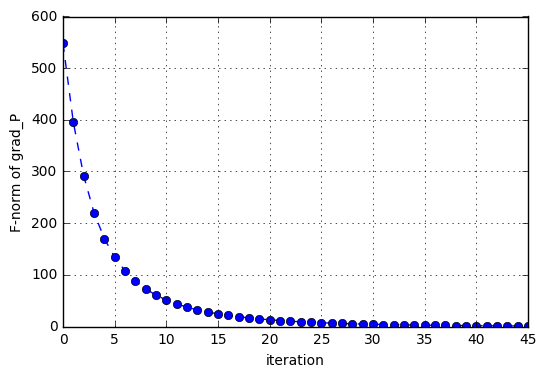
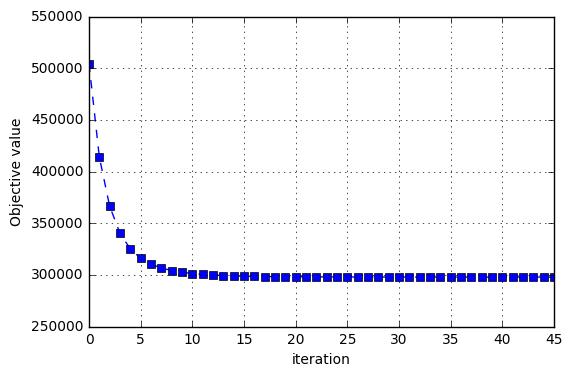
 

Fig. Convolution of and in each iteration

# **Algorithm Refinement with fixed**

**Question 3.1 Gradient descent with line search**

The gradient method with constant pace need to compute the Lipschitz constant of the gradient, which may require much work. Even more, for problem (1) the gradient of is not lipschitzien, so we should consider a new method free from computing Lipschitz constant.

The idea of line search is to choose the pace adaptively using local information. In exact line search we take

|  |
| --- |
|  |

We apply the function scipy.optimize.line\_search to solve this optimisation problem.

**def** linesearch\_gradient(P0,epsilon):

P=P0

grad=tools.objective(P, Q, R, mask, rho)[1]

G=list([])

C=list([])

**while** np.sqrt(np.sum(grad\*\*2))>epsilon:

gamma=line\_search(g\_PQ, grad\_P, np.ravel(P), -grad\_P(np.ravel(P),Q), args=(Q,))[0]

P=P-gamma\*grad

grad=tools.objective(P, Q, R, mask, rho)[1]

G.append(np.sqrt(np.sum(grad\*\*2)))

C.append(tools.objective(P, Q, R, mask, rho)[0])

**return** P,G,C

u,s,vt = svds(R,7)

Q0=u

P0=vt

rho=0.2

P\_opt,G,C=linesearch\_gradient(P0,1)

plt.figure()

plt.plot(G,linestyle='--',marker='o')

plt.xlabel('iteration')

plt.ylabel('F-norm of grad\_P')

plt.grid()

plt.figure()

plt.plot(C,linestyle='--',marker='s')

plt.xlabel('iteration')

plt.ylabel('Objective value')

plt.grid()

plt.show()

Similarly, we show the convolution of for each iteration.

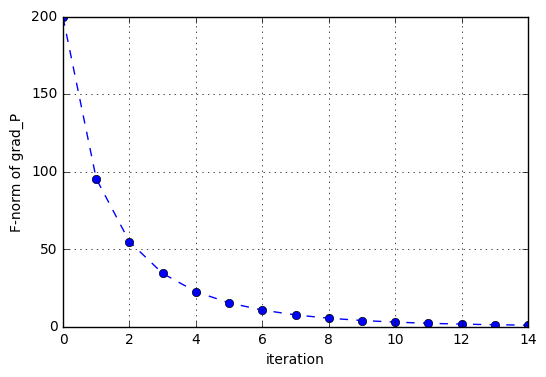
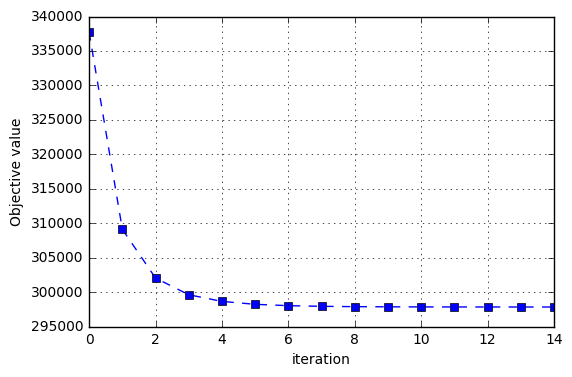
**** ****

Fig. Convolution of and with line search

**Question 3.2 Conjugate gradient method**

The nonlinear conjugate gradient method is generally used to find the local minimum of a nonlinear function using its gradient alone. It works when the function is approximately quadratic near the minimum, which is the case when the function is twice differentiable at the minimum and the second derivative is non-singular there.

Since is continuously twice differentiable and is non-singular at the minimum, so the conjugate gradient method can be applied to the problem when Q is fixed.

The principle of the algorithm is described as follows:

|  |
| --- |
| Initialise    ,  , |
| repeat   * choose * k=k+1 |
| until |

**def** conj\_gradient(P,Q,epsilon):

grad=tools.total\_objective(P, Q, R, mask, rho)[1]

d=-grad

G=list([])

C=list([])

**while** np.sqrt(np.sum(grad\*\*2))>epsilon:

gamma=line\_search(g\_PQ, grad\_P, np.ravel(P), -grad\_P(np.ravel(P),Q), args=(Q,))[0]

P=P+gamma\*d

grad\_old=grad

grad=tools.total\_objective(P, Q, R, mask, rho)[1]

b=np.sum(grad\*\*2)/np.sum(grad\_old\*\*2)

d=-grad+b\*d

G.append(np.sqrt(np.sum(grad\*\*2)))

C.append(tools.objective(P, Q, R, mask, rho)[0])

**return** P,G,C

u,s,vt = svds(R,7)

Q=u

P0=vt

P\_opt,G,C=conj\_gradient(P0,Q,1)

plt.figure()

plt.plot(G,linestyle='--',marker='o')

plt.xlabel('iteration')

plt.ylabel('F-norm of grad\_P')

plt.grid()

plt.figure()

plt.plot(C,linestyle='--',marker='s')

plt.xlabel('iteration')

plt.ylabel('Objective value')

plt.grid()

plt.show()

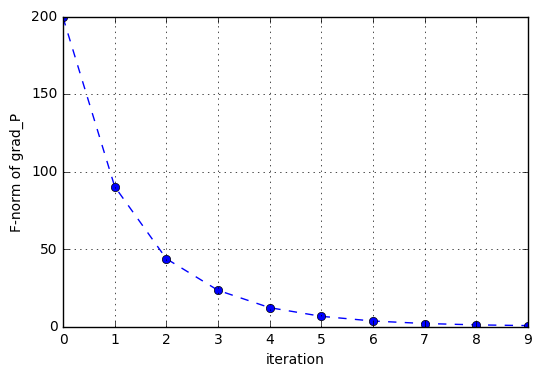
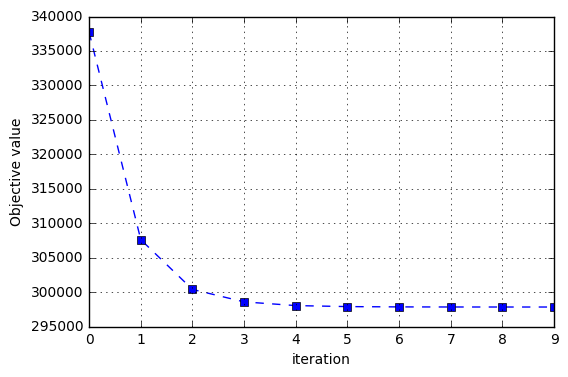
 

Fig. Convolution of and with conjugate gradient method

**Question 3.3 Comparison**

With the same stop condition, we see the number of iteration required for convergence for these three methods:

|  |  |  |  |
| --- | --- | --- | --- |
|  | gradient with constant pace length | gradient with line search | conjugate gradient |
| number of iteration required for convergence | 45 | 14 | 9 |
| Objective value after convergence | 297828 | 297828 | 297827 |

We can conclude that the three method above converge to the same solution, but with different convergence speed:

gradient with constant pace length< gradient with line search< conjugate gradient

# **Resolution of the complete problem**

**4.1 Solve problem (1) with the gradient method with line search**

**def** g\_PQvec(PQvec):

**return** tools.total\_objective\_vectorized(PQvec, R, mask, rho)[0]

**def** grad\_PQ(PQvec):

**return** tools.total\_objective\_vectorized(PQvec, R, mask, rho)[1]

**def** linesearch\_gradient\_PQ(P0, Q0, epsilon):

PQvec=np.concatenate([P0.ravel(), Q0.ravel()])

grad=grad\_PQ(PQvec)

G=list([])

C=list([])

**while** np.sqrt(np.sum(grad\*\*2))>epsilon:

gamma=line\_search(g\_PQvec, grad\_PQ, PQvec, -grad\_PQ(PQvec) )[0]

PQvec=PQvec-gamma\*grad

grad=grad\_PQ(PQvec)

G.append(np.sqrt(np.sum(grad\*\*2)))

C.append

(tools.total\_objective\_vectorized(PQvec, R, mask, rho)[0])

**return** PQvec,G,C

u,s,vt = svds(R,7)

Q0=u

P0=vt

rho=0.2

PQ\_opt,G=linesearch\_gradient\_PQ(P0,Q0,100)

plt.plot(G,linestyle='--',marker='o')

plt.xlabel('iteration')

plt.ylabel('F-norm of grad\_P')

plt.grid()

plt.show()

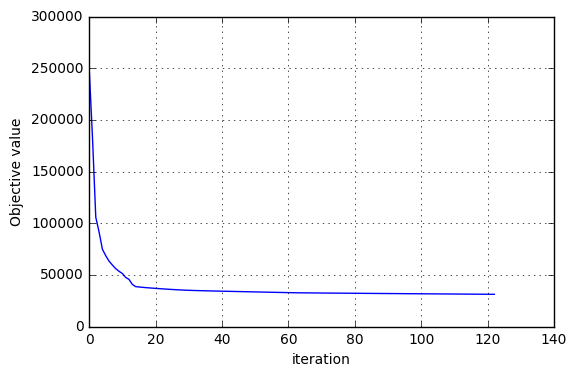
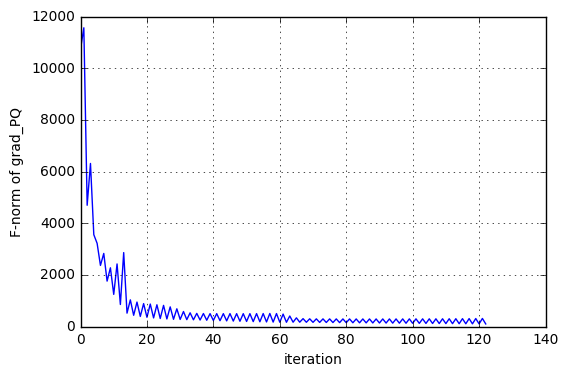


Fig. Convolution of and with line search

**4.2 Convergence of alternating least square method**

Since , we have

Similarly from , we have

From the two inequality above we have

In the same way, we can prove that

This indicates that the objective functions decrease with the iteration, and finally converge.

**4.3 Alternating Least square**

Optimise P and Q alternatively, by minimising the following two equations

The principle of the algorithm is shown as follows.

|  |
| --- |
| Initialise |
| for do  for *i=*1,…,1682      for *j=*1,…,943       * k=k+1 |
| until |

u,s,vt = svds(R,7)

Q=u

P=vt

C=list([])

epsilon=100

rho=0.2

k=0

**while** k<2 **or** np.abs(C[-1]-C[-2])>10:

**for** i **in** range(P.shape[1]):

P[:,i]=np.linalg.inv (Q.T.dot( np.diag(mask[:,i]) ).dot(Q)+rho\*np.ey e(Q.shape[1])).dot(Q.T).dot(R[:,i].dot(np.diag(mask[:,i])))

**for** u **in** range(Q.shape[0]):

Q[u,:]=R[u,:].dot(np.diag(mask[u,:])).dot(P.T).dot(np.linalg.inv(P.d ot( np.diag(mask[u,:]) ).dot(P.T)+rho\*np.eye(P.shape[0])))

C.append(tools.total\_objective(P, Q, R, mask, rho)[0])

k=k+1

plt.plot(C)

plt.xlabel('iteration')

plt.ylabel('Objective value')

plt.grid()

plt.show()

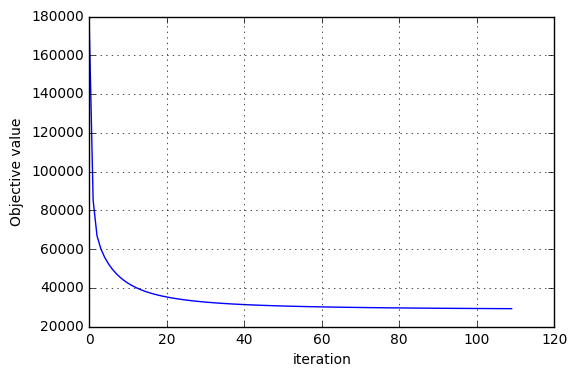


Fig. Convolution of with alternating least square

**4.4 Comparison**

|  |  |  |
| --- | --- | --- |
|  | gradient with line search | alternating least square |
| NMSE of P,Q between two methods | 2.066 | |
| NMSE of R between two methods | 1.366 | |
| number of iteration required for convergence | 123 | 110 |
| Objective value after convergence | 31148 | 29198 |
| Calculation time | 195s | 5939s |

We define the normalized mean square error (NMSE) to measure the difference of P, Q and R,

From the NMSE we can see there is great difference between the results of these two methods.

As for the performance, alternating least square convergence a little faster than gradient method, and

**4.4 Recommendation**

We estimate R by

If we recommend a film to user 449, we will choose the film which he/she hasn’t rated but would give an estimated highest rating.

P\_opt=PQ\_opt[:P0.shape[0]\*P0.shape[1]].reshape(P0.shape)

Q\_opt=PQ\_opt[P0.shape[0]\*P0.shape[1]:].reshape(Q0.shape)

R\_est=Q\_opt.dot(P\_opt)

print(np.argmax(R\_est[449,:]\*(1-mask)[449,:]))

print(np.max(R\_est[449,:]))

plt.figure(figsize=(15,6))

plt.bar(range(R.shape[1]),R\_est[449,:]\*(1-mask)[449,:])

plt.xlabel('Film')

plt.ylabel('Rate of user 449')

plt.grid()

plt.show()

We use both gradient method with line search and alternating least square method to estimate P and Q, and get the recommendation for user 449. (We notice that in the data sets, users and films are numbered consecutively from 1.)

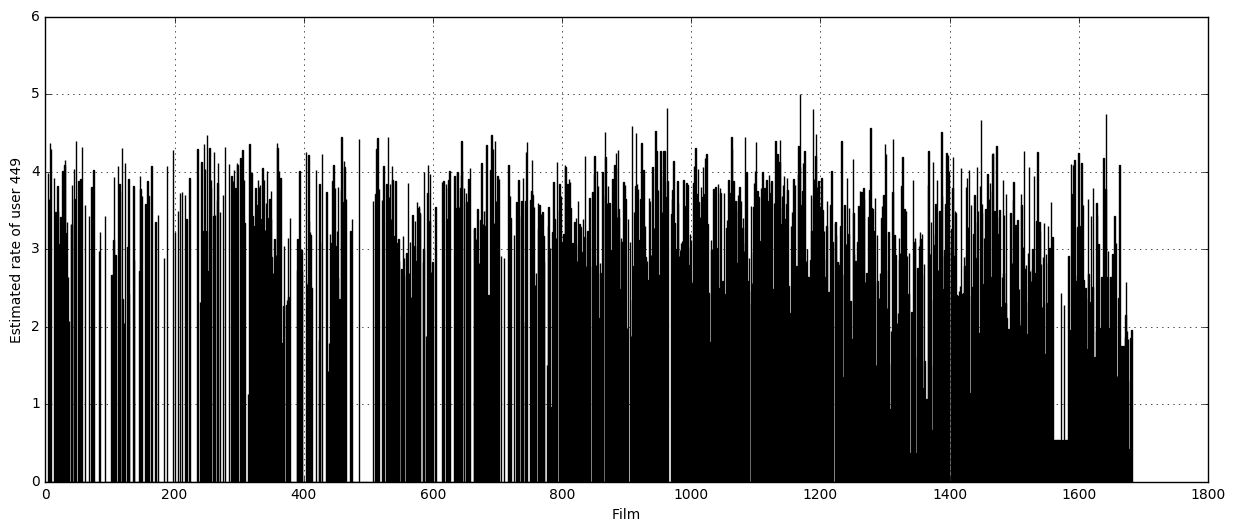


Fig. Estimate by gradient with line search

By gradient method with line search, we get the estimate of R, and we recommend **film 1449** to user 449.

We notice that in the data sets, users and films are numbered consecutively from 1. In the data set u.item we find the information of **film 1449**.



Fig. Film recommended by gradient with line search

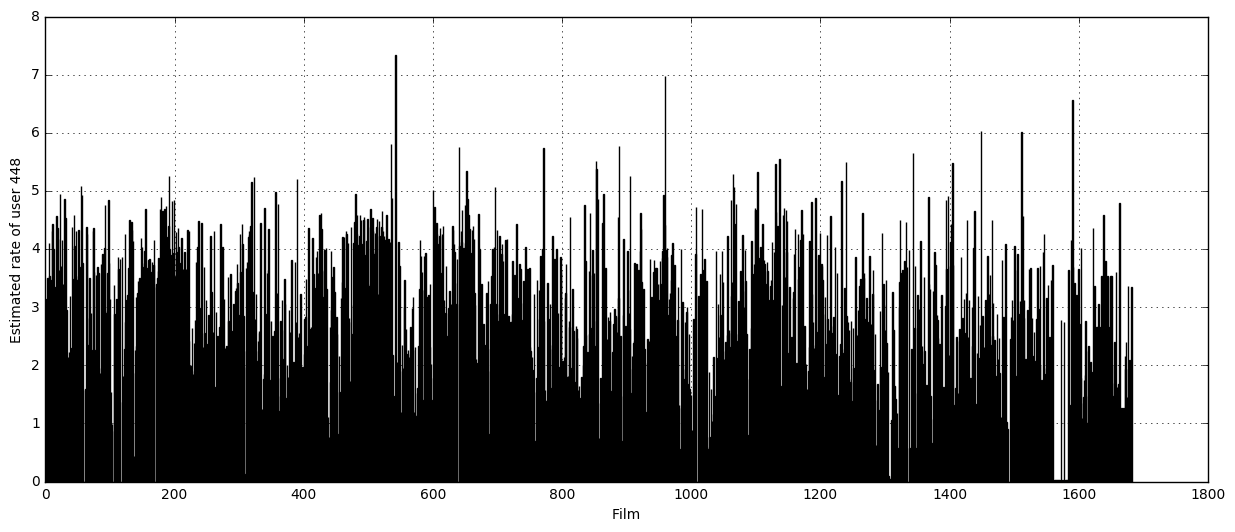


Fig. Estimate by alternating least square

By alternating least square method, we recommend **film 124** to user 449. We can conclude that these two methods will lead to different recommendation result.



Fig. Film recommended by alternating least square