ENGG 4030 Homework_4

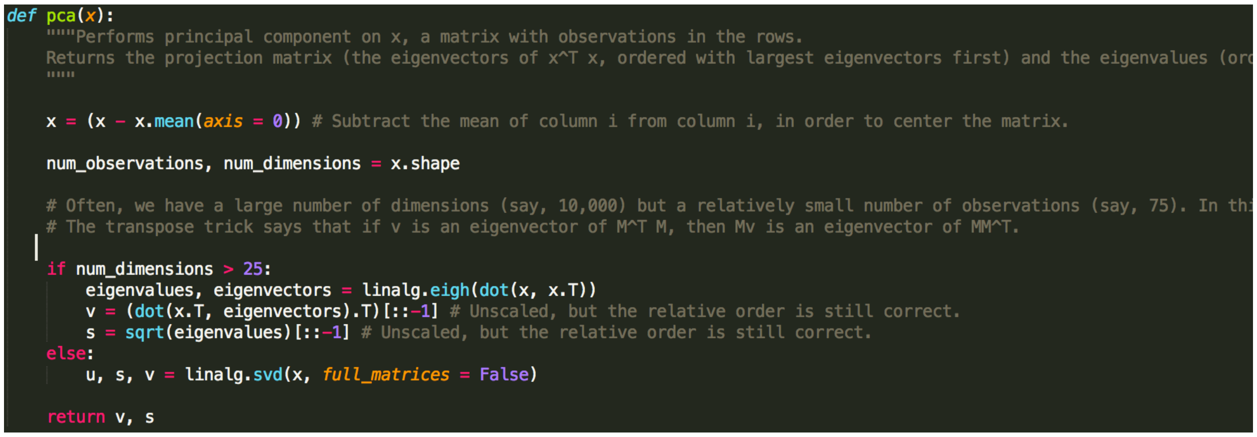
I declare that the assignment submitted on Elearning system is original except for source material explicitly acknowledged, and that the same or related material has not been previously submitted for another course. I also acknowledge that I am aware of University policy and regulations on honesty in academic work, and of the disciplinary guidelines and procedures applicable to breaches of such policy and regulations, as contained in the website http://www.cuhk.edu.hk/policy/academichonesty/ .

Signed (Student 柳一村) Date: 22/04/2018
Name LIU Yicun SID 1155092202

Q1: PCA

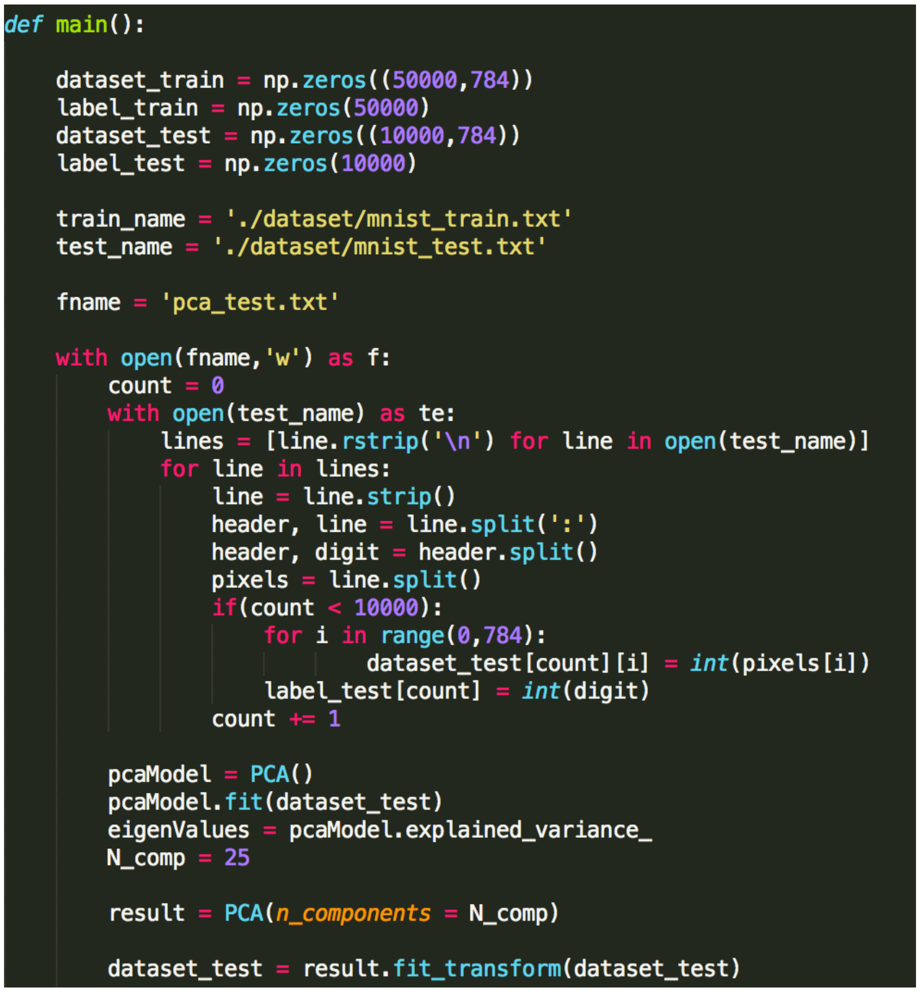
PCA+K-Means:

To implementing Principal Components Analysis, I simply follow the formula provided in the slide and write in python the pca() function:



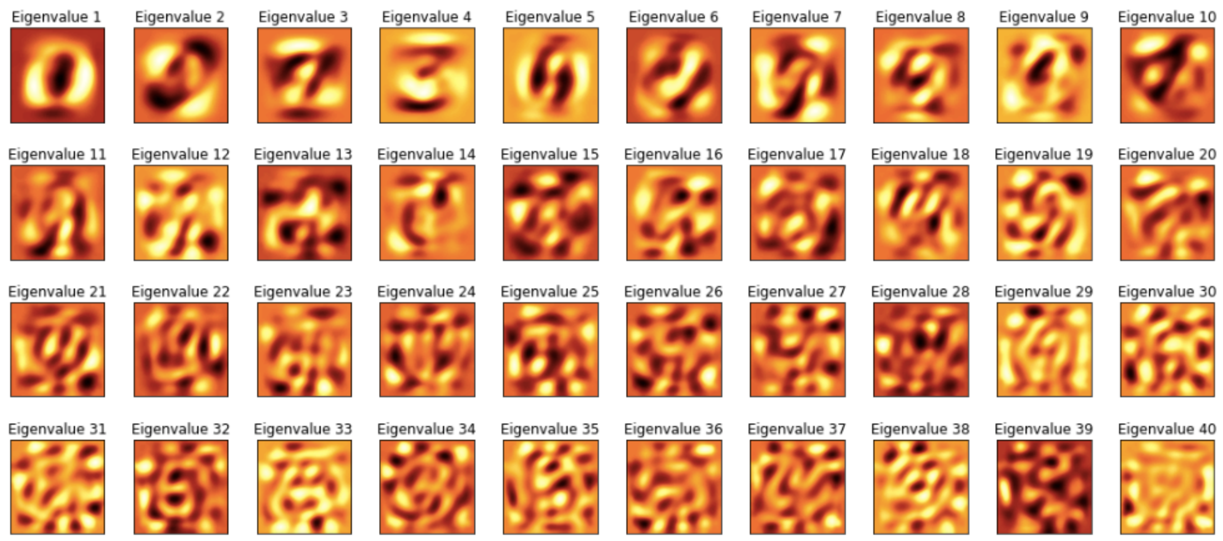
```python
def pca(x):
    """Performs principal component on x, a matrix with observations in the rows.
    Returns the projection matrix (the eigenvectors of x^T x, ordered with largest eigenvectors first) and the eigenvalues (orc
    """

    x = (x - x.mean(axis = 0)) # Subtract the mean of column i from column i, in order to center the matrix.

    num_observations, num_dimensions = x.shape

    # Often, we have a large number of dimensions (say, 10,000) but a relatively small number of observations (say, 75). In thi
    # The transpose trick says that if v is an eigenvector of M^T M, then Mv is an eigenvector of MM^T.

    if num_dimensions > 25:
        eigenvalues, eigenvectors = linalg.eigh(dot(x, x.T))
        v = (dot(x.T, eigenvectors).T)[::-1] # Unscaled, but the relative order is still correct.
        s = sqrt(eigenvalues)[::-1] # Unscaled, but the relative order is still correct.
    else:
        u, s, v = linalg.svd(x, full_matrices = False)

    return v, s
```

To ensure my implementation is correct, I import PCA form the third party library ‘sklearn’ to double check:

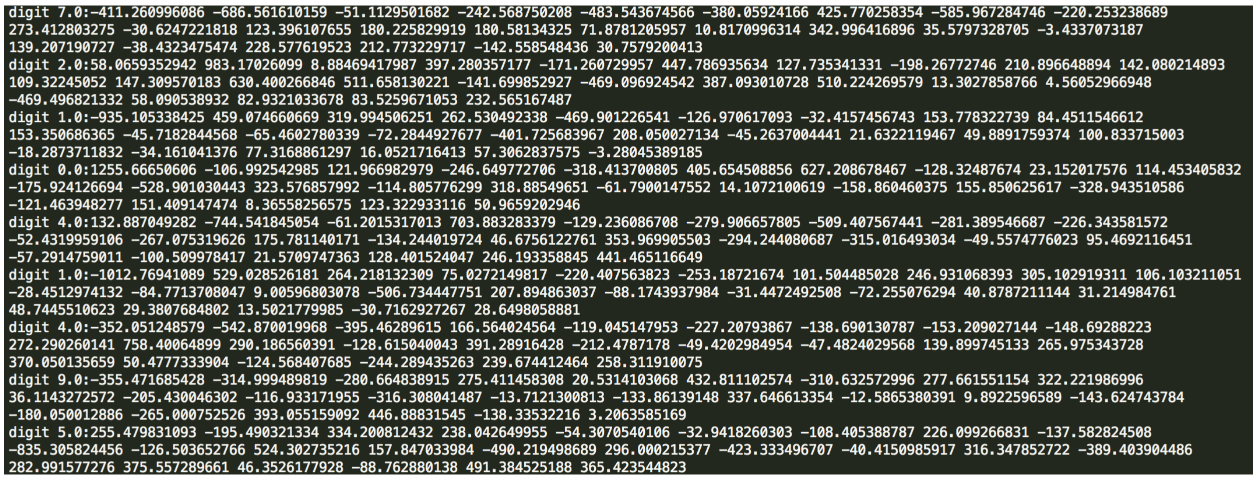


```python
def main():

    dataset_train = np.zeros((50000,784))
    label_train = np.zeros(50000)
    dataset_test = np.zeros((10000,784))
    label_test = np.zeros(10000)

    train_name = './dataset/mnist_train.txt'
    test_name = './dataset/mnist_test.txt'

    fname = 'pca_test.txt'

    with open(fname,'w') as f:
        count = 0
        with open(test_name) as te:
            lines = [line.rstrip('\n') for line in open(test_name)]
            for line in lines:
                line = line.strip()
                header, line = line.split(':')
                header, digit = header.split()
                pixels = line.split()
                if(count < 10000):
                    for i in range(0,784):
                        dataset_test[count][i] = int(pixels[i])
                    label_test[count] = int(digit)
                count += 1

        pcaModel = PCA()
        pcaModel.fit(dataset_test)
        eigenValues = pcaModel.explained_variance_
        N_comp = 25

        result = PCA(n_components = N_comp)

        dataset_test = result.fit_transform(dataset_test)
```

After dimension reduction, the top 40 ‘eigen values’ are visualized as this:



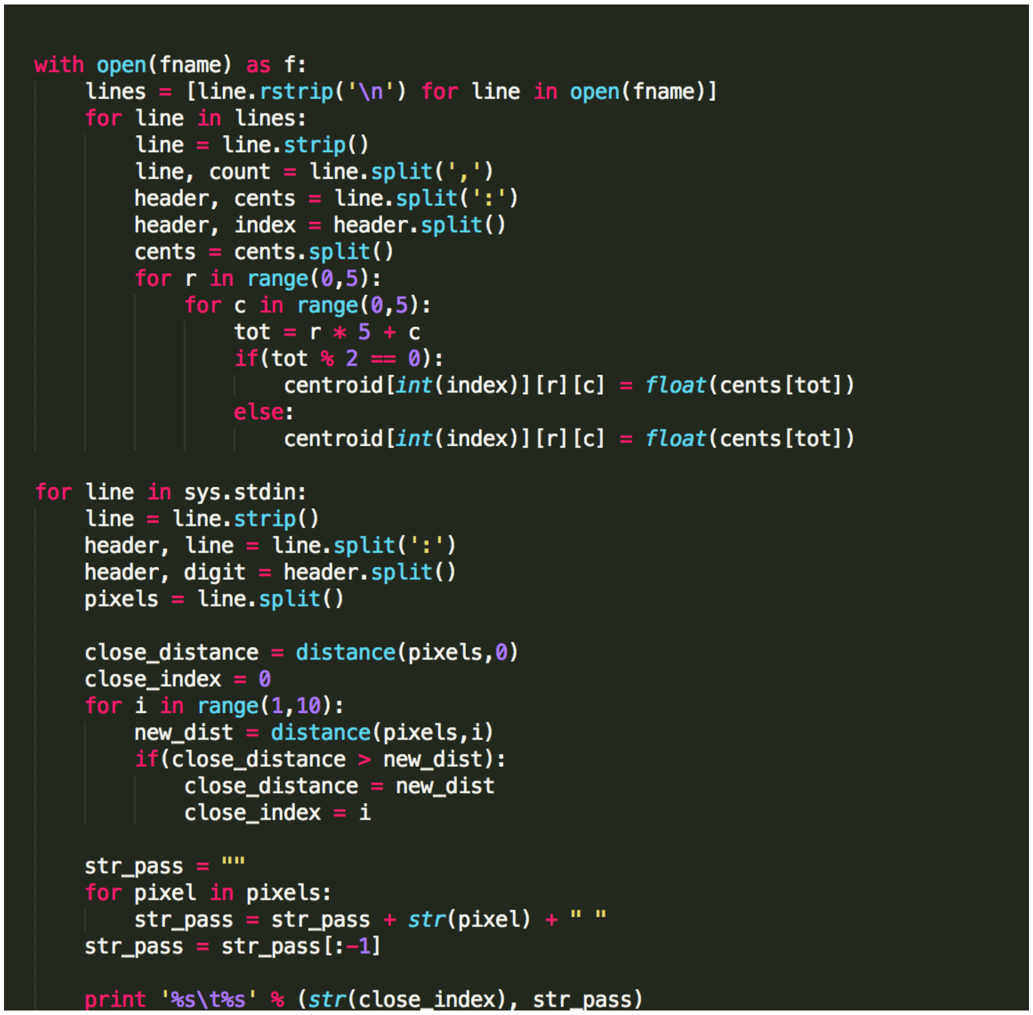
As instructed, I only pick to top 25 of the eigen representations and now the data for each digit is dimensionally reduced from 784 dimensions to 25 dimensions.



```
digit 7.0:-411.260996086 -686.561610159 -51.1129501682 -242.568750208 -483.543674566 -380.05924166 425.770258354 -585.967284746 -220.253238689
273.412803275 -30.6247221818 123.396107655 180.225829919 180.58134325 71.8781205957 10.8170996314 342.996416896 35.5797328705 -3.4337073187
139.207190727 -38.4323475474 228.577619523 212.773229717 -142.558548436 30.7579200413
digit 2.0:58.0659352942 983.17026099 8.88469417987 397.280357177 -171.260729957 447.786935634 127.735341331 -198.26772746 210.896648894 142.080214893
109.32245052 147.309570183 630.400266846 511.658130221 -141.699852927 -469.096924542 387.093010728 510.224269579 13.3027858766 4.56052966948
-469.496821332 58.090538932 82.9321033678 83.5259671053 232.565167487
digit 1.0:-935.105338425 459.074660669 319.994506251 262.530492338 -469.901226541 -126.970617093 -32.4157456743 153.778322739 84.4511546612
153.350686365 -45.7182844568 -65.4602780339 -72.2844927677 -401.725683967 208.050027134 -45.2637004441 21.6322119467 49.8891759374 100.833715003
-18.2873711832 -34.161041376 77.3168861297 16.0521716413 57.3062837575 -3.28045389185
digit 0.0:1255.66650606 -106.992542985 121.966982979 -246.649772706 -318.413700805 405.654508856 627.208678467 -128.32487674 23.152017576 114.453405832
-175.924126694 -528.901030443 323.576857992 -114.805776299 318.88549651 -61.7900147552 14.1072100619 -158.860460375 155.850625617 -328.943510586
-121.463948277 151.409147474 8.36558256575 123.322933116 50.9659202946
digit 4.0:132.887049282 -744.541845054 -61.2015317013 703.883283379 -129.236086708 -279.906657805 -509.407567441 -281.389546687 -226.343581572
-52.4319959106 -267.075319626 175.781140171 -134.244019724 46.6756122761 353.969905503 -294.244080687 -315.016493034 -49.5574776023 95.4692116451
-57.2914759011 -100.509978417 21.5709747363 128.401524047 246.193358845 441.465116649
digit 1.0:-1012.76941089 529.028526181 264.218132309 75.0272149817 -220.407563823 -253.18721674 101.504485028 246.931068393 305.102919311 106.103211051
-28.4512974132 -84.7713708047 9.00596803078 -506.734447751 207.894863037 -88.1743937984 -31.4472492508 -72.255076294 40.8787211144 31.214984761
48.7445510623 29.3807684802 13.5021779985 -30.7162927267 28.6498058881
digit 4.0:-352.051248579 -542.870019968 -395.46289615 166.564024564 -119.045147953 -227.20793867 -138.690130787 -153.209027144 -148.69288223
272.290260141 758.40064899 290.186560391 -128.615040043 391.28916428 -212.4787178 -49.4202984954 -47.4824029568 139.899745133 265.975343728
370.050135659 50.4777333904 -124.568407685 -244.289435263 239.674412464 258.311910075
digit 9.0:-355.471685428 -314.999489819 -280.664838915 275.411458308 20.5314103068 432.811102574 -310.632572996 277.661551154 322.221986996
36.1143272572 -205.430046302 -116.933171955 -316.308041487 -13.7121300813 -133.86139148 337.646613354 -12.5865380391 9.8922596589 -143.624743784
-180.050012886 -265.000752526 393.055159092 446.88831545 -138.33532216 3.2063585169
digit 5.0:255.479831093 -195.490321334 334.200812432 238.042649955 -54.3070540106 -32.9418260303 -108.405388787 226.099266831 -137.582824508
-835.305824456 -126.503652766 524.302735216 157.847033984 -490.219498689 296.000215377 -423.333496707 -40.4150985917 316.347852722 -389.403904486
282.991577276 375.557289661 46.3526177928 -88.762880138 491.384525188 365.423544823
```

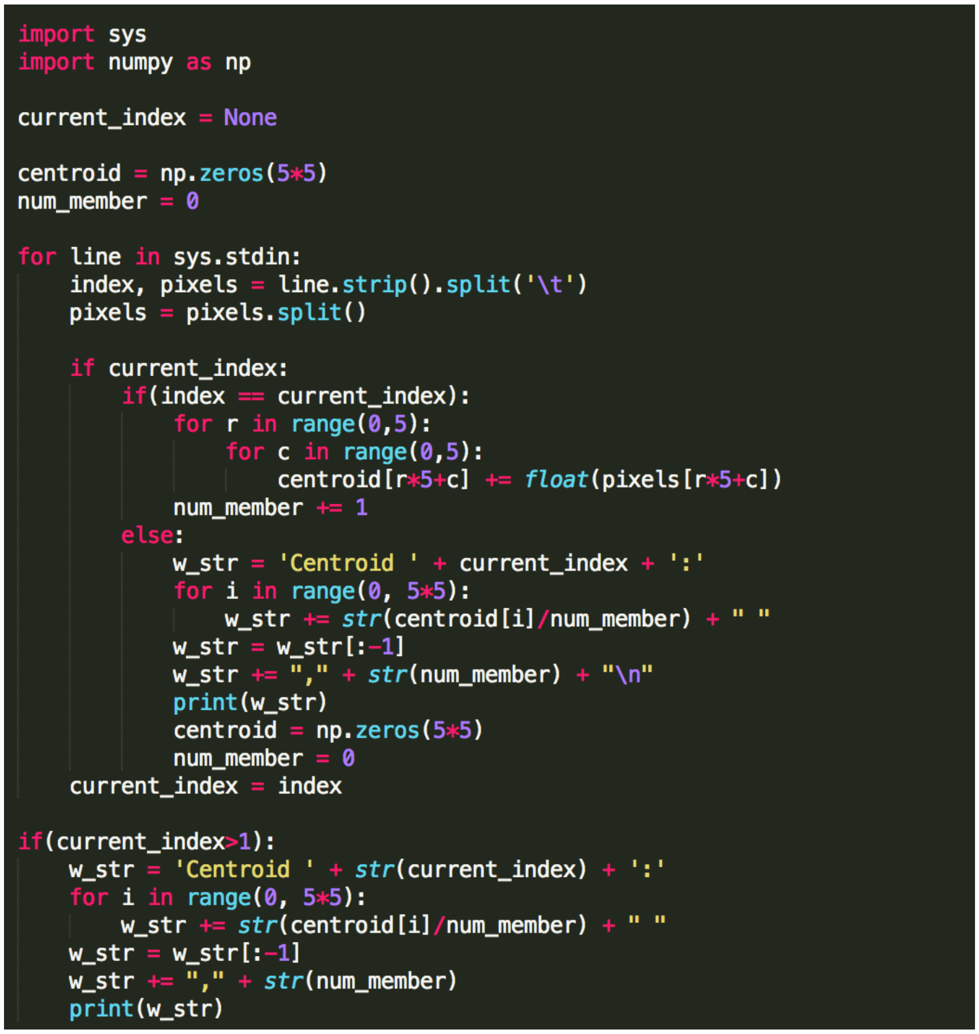
Examples of the 'eigen digits' after PCA (num_components = 25)

Here we use k-means clustering in a distributed manner. The detailed implementation of k-means is similar to the implementation in assignment 3, except for the dimension part. The mapper computes the distance of each digit to the centroid and then assigns the digit to the cluster. The reducer recount the centroid, based on the points received from the mapper.



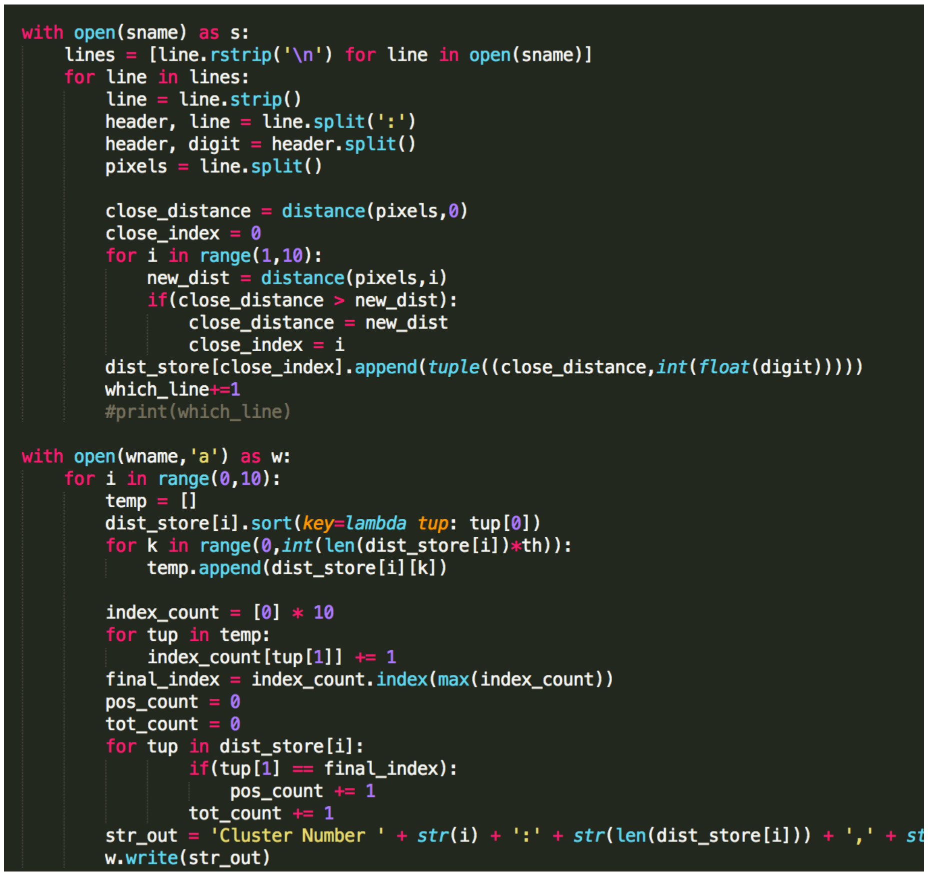
```
with open(fname) as f:
    lines = [line.rstrip('\n') for line in open(fname)]
    for line in lines:
        line = line.strip()
        line, count = line.split(',')
        header, cents = line.split(':')
        header, index = header.split()
        cents = cents.split()
        for r in range(0,5):
            for c in range(0,5):
                tot = r * 5 + c
                if(tot % 2 == 0):
                    centroid[int(index)][r][c] = float(cents[tot])
                else:
                    centroid[int(index)][r][c] = float(cents[tot])

for line in sys.stdin:
    line = line.strip()
    header, line = line.split(':')
    header, digit = header.split()
    pixels = line.split()

    close_distance = distance(pixels,0)
    close_index = 0
    for i in range(1,10):
        new_dist = distance(pixels,i)
        if(close_distance > new_dist):
            close_distance = new_dist
            close_index = i

    str_pass = ""
    for pixel in pixels:
        str_pass = str_pass + str(pixel) + " "
    str_pass = str_pass[:-1]

    print '%s\t%s' % (str(close_index), str_pass)
```

Mapper in K-means



```python
import sys
import numpy as np

current_index = None

centroid = np.zeros(5*5)
num_member = 0

for line in sys.stdin:
    index, pixels = line.strip().split('\t')
    pixels = pixels.split()

    if current_index:
        if(index == current_index):
            for r in range(0,5):
                for c in range(0,5):
                    centroid[r*5+c] += float(pixels[r*5+c])
            num_member += 1
        else:
            w_str = 'Centroid ' + current_index + ':'
            for i in range(0, 5*5):
                w_str += str(centroid[i]/num_member) + " "
            w_str = w_str[:-1]
            w_str += "," + str(num_member) + "\n"
            print(w_str)
            centroid = np.zeros(5*5)
            num_member = 0
    current_index = index

if(current_index>1):
    w_str = 'Centroid ' + str(current_index) + ':'
    for i in range(0, 5*5):
        w_str += str(centroid[i]/num_member) + " "
    w_str = w_str[:-1]
    w_str += "," + str(num_member)
    print(w_str)
```

Reducer in k-means, the centroid info is store in the additional file



```python
with open(sname) as s:
    lines = [line.rstrip('\n') for line in open(sname)]
    for line in lines:
        line = line.strip()
        header, line = line.split(':')
        header, digit = header.split()
        pixels = line.split()

        close_distance = distance(pixels,0)
        close_index = 0
        for i in range(1,10):
            new_dist = distance(pixels,i)
            if(close_distance > new_dist):
                close_distance = new_dist
                close_index = i
        dist_store[close_index].append(tuple((close_distance,int(float(digit)))))
        which_line+=1
        #print(which_line)

with open(wname,'a') as w:
    for i in range(0,10):
        temp = []
        dist_store[i].sort(key=lambda tup: tup[0])
        for k in range(0,int(len(dist_store[i])*th)):
            temp.append(dist_store[i][k])

        index_count = [0] * 10
        for tup in temp:
            index_count[tup[1]] += 1
        final_index = index_count.index(max(index_count))
        pos_count = 0
        tot_count = 0
        for tup in dist_store[i]:
            if(tup[1] == final_index):
                pos_count += 1
            tot_count += 1
        str_out = 'Cluster Number ' + str(i) + ':' + str(len(dist_store[i])) + ',' + st
        w.write(str_out)
```

Post-processing code based on different thresholds

X = 0.05

Cluster Number	Images in the Entire Cluster	Images which are considered (m)	Major labels of the central images	Correctly clustered image	Classification Accuracy
0	796	39	0	705	0.88
1	849	42	8	483	0.57
2	868	43	8	110	0.13
3	935	46	3	738	0.79
4	1380	69	6	863	0.63
5	1299	64	7	698	0.54
6	539	26	9	146	0.27
7	1186	59	1	791	0.67
8	903	45	2	612	0.68
9	1245	62	8	237	0.19
Total	10000			5583	55.8

X = 0.1

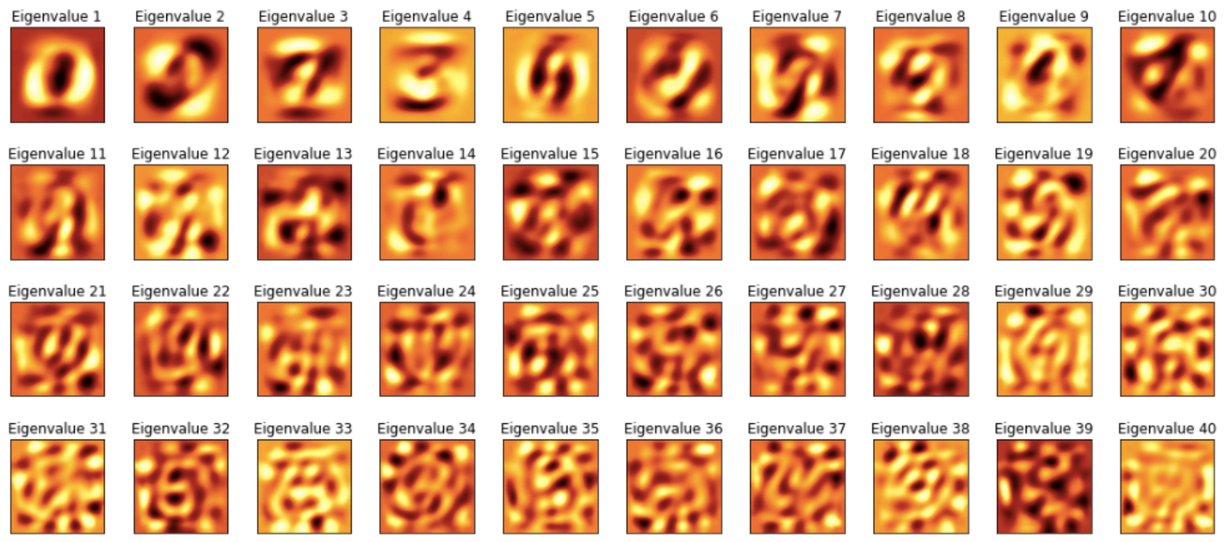
Cluster Number	Images in the Entire Cluster	Images which are considered (m)	Major labels of the central images	Correctly clustered image	Classification Accuracy
0	796	79	0	705	0.88
1	849	84	8	483	0.57
2	868	86	5	348	0.40
3	935	93	3	738	0.79
4	1380	138	6	863	0.63
5	1299	129	7	698	0.54
6	539	53	9	146	0.27
7	1186	118	1	791	0.67
8	903	90	2	612	0.68
9	1245	124	8	237	0.19
Total	10000			5821	58.2

X = 0.5

Cluster Number	Images in the Entire Cluster	Images which are considered (m)	Major labels of the central images	Correctly clustered image	Classification Accuracy
0	796	398	0	705	0.88
1	849	424	8	483	0.57
2	868	434	5	348	0.40
3	935	467	3	738	0.79
4	1380	690	6	863	0.63
5	1299	649	7	698	0.54
6	539	269	4	246	0.45
7	1186	593	1	791	0.67
8	903	451	2	612	0.68
9	1245	622	8	237	0.19
Total	10000			5921	59.2

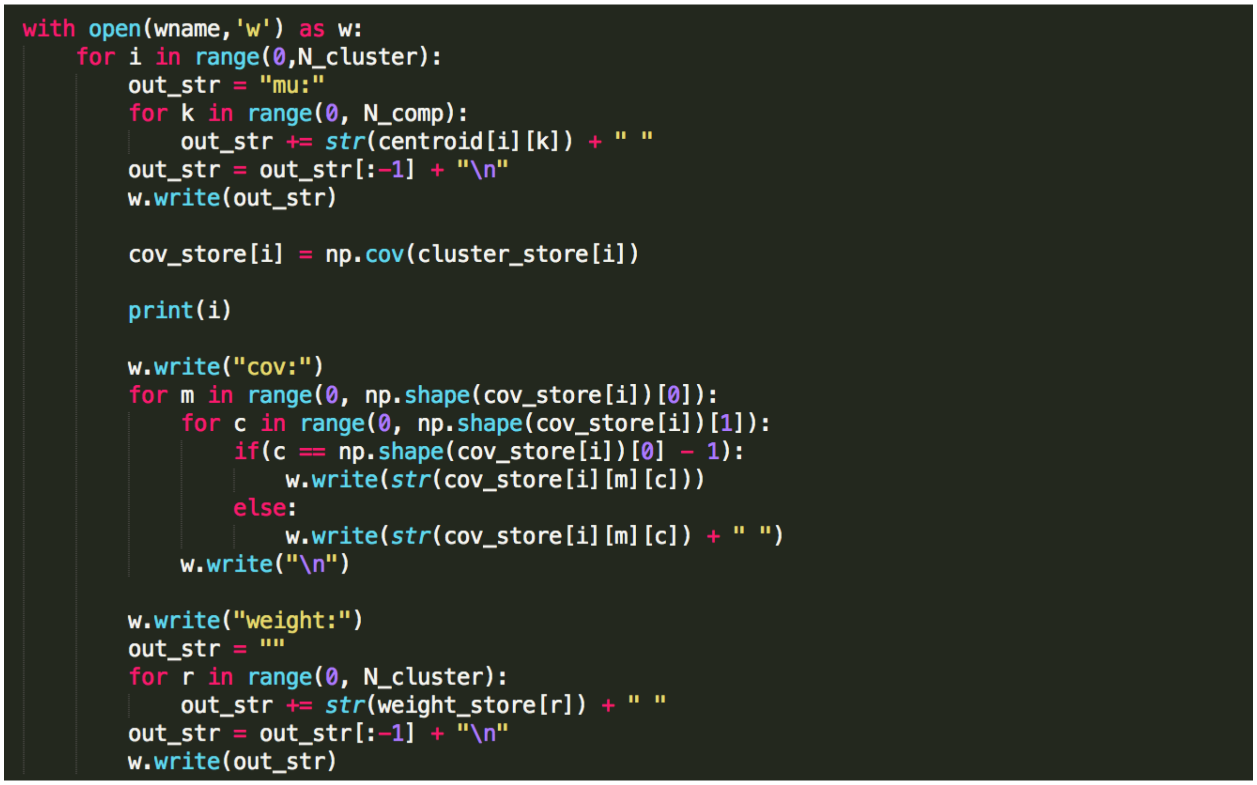
After testing the three threshold, we find that when the threshold is small, one cluster is easier to be classified into ‘8’. As the threshold increases, we observe an increase in the accuracy and several previously wrongly classified clusters was corrected from ‘8’. The best threshold is 0.5 in my testing cases.

The reason why ‘eigen images’ are more prone to trapping in ‘8’ can be partially explained in the result of PCA. As we can see, only the first ten eigen values has significant tendencies (also several exceptions like 32), leaving the rest of the eigen values to be quite info-less (a mess which is more like a ‘8’).



I don’t observe a significant gain in the accuracy after using PCA as prior of K-means. The accuracy is quite similar and may be the limitation of classification mostly is the simple distance measurement in K-means cannot represent the complex relationship in higher dimensional, also the cluster shape might not be a sphere, even after PCA.

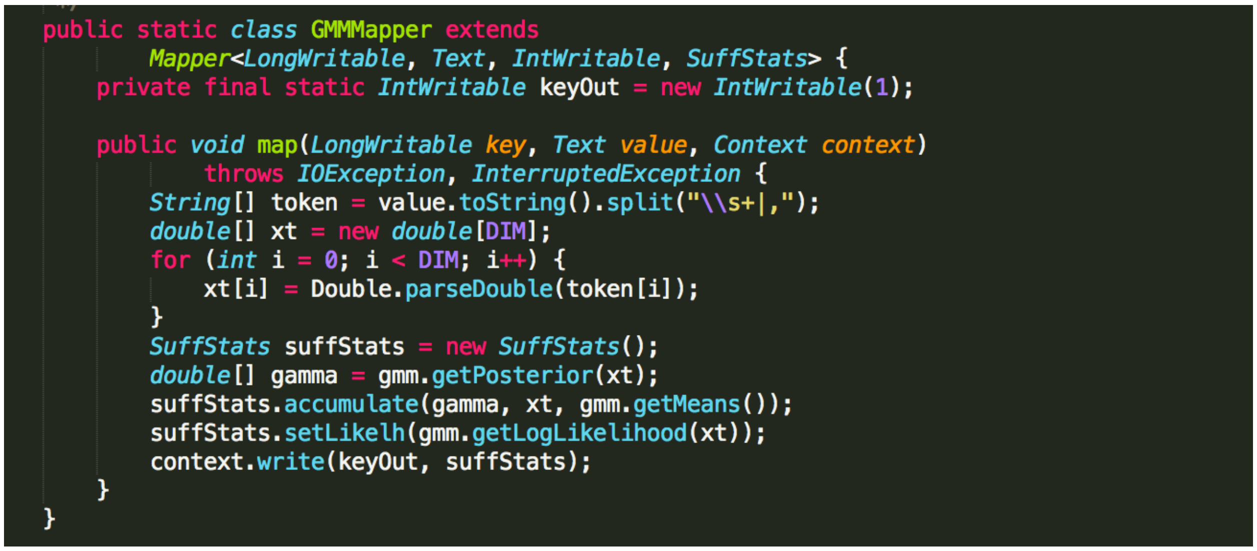
PCA+EM-GMM:
To initialize the EM-GMM, I use the centroid result of the previous step, the compute ‘mu’,’co-variance matrix’ and the weight ‘pi’ of each cluster.



```python
with open(wname,'w') as w:
    for i in range(0,N_cluster):
        out_str = "mu:"
        for k in range(0, N_comp):
            out_str += str(centroid[i][k]) + " "
        out_str = out_str[:-1] + "\n"
        w.write(out_str)

        cov_store[i] = np.cov(cluster_store[i])

        print(i)

        w.write("cov:")
        for m in range(0, np.shape(cov_store[i])[0]):
            for c in range(0, np.shape(cov_store[i])[1]):
                if(c == np.shape(cov_store[i])[0] - 1):
                    w.write(str(cov_store[i][m][c]))
                else:
                    w.write(str(cov_store[i][m][c]) + " ")
            w.write("\n")

        w.write("weight:")
        out_str = ""
        for r in range(0, N_cluster):
            out_str += str(weight_store[r]) + " "
        out_str = out_str[:-1] + "\n"
        w.write(out_str)
```

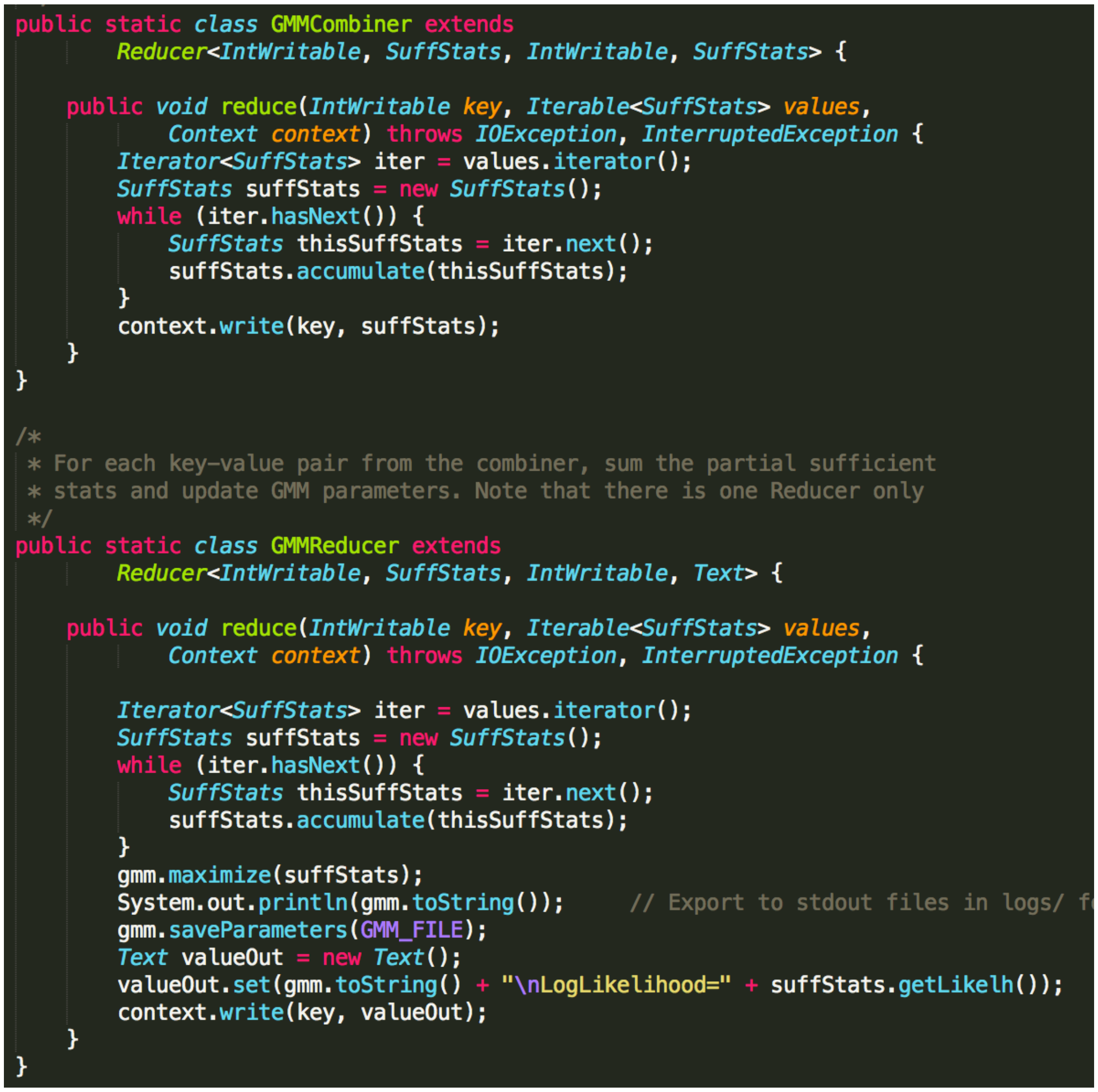
Initialize the mu, cov and weight for EM-GMM

For simplicity, I wrote the EM-GMM in java with mapper, combiner and reducer.



```java
public static class GMMMapper extends
        Mapper<LongWritable, Text, IntWritable, SuffStats> {
    private final static IntWritable keyOut = new IntWritable(1);

    public void map(LongWritable key, Text value, Context context)
            throws IOException, InterruptedException {
        String[] token = value.toString().split("\\s+|,");
        double[] xt = new double[DIM];
        for (int i = 0; i < DIM; i++) {
            xt[i] = Double.parseDouble(token[i]);
        }
        SuffStats suffStats = new SuffStats();
        double[] gamma = gmm.getPosterior(xt);
        suffStats.accumulate(gamma, xt, gmm.getMeans());
        suffStats.setLikelh(gmm.getLogLikelihood(xt));
        context.write(keyOut, suffStats);
    }
}
```

Part of the mapper in EM-GMM, E-step



```java
public static class GMMCombiner extends
        Reducer<IntWritable, SuffStats, IntWritable, SuffStats> {

    public void reduce(IntWritable key, Iterable<SuffStats> values,
            Context context) throws IOException, InterruptedException {
        Iterator<SuffStats> iter = values.iterator();
        SuffStats suffStats = new SuffStats();
        while (iter.hasNext()) {
            SuffStats thisSuffStats = iter.next();
            suffStats.accumulate(thisSuffStats);
        }
        context.write(key, suffStats);
    }
}

/*
 * For each key-value pair from the combiner, sum the partial sufficient
 * stats and update GMM parameters. Note that there is one Reducer only
 */
public static class GMMReducer extends
        Reducer<IntWritable, SuffStats, IntWritable, Text> {

    public void reduce(IntWritable key, Iterable<SuffStats> values,
            Context context) throws IOException, InterruptedException {

        Iterator<SuffStats> iter = values.iterator();
        SuffStats suffStats = new SuffStats();
        while (iter.hasNext()) {
            SuffStats thisSuffStats = iter.next();
            suffStats.accumulate(thisSuffStats);
        }
        gmm.maximize(suffStats);
        System.out.println(gmm.toString());     // Export to stdout files in logs/ f
        gmm.saveParameters(GMM_FILE);
        Text valueOut = new Text();
        valueOut.set(gmm.toString() + "\nLogLikelihood=" + suffStats.getLikelh());
        context.write(key, valueOut);
    }
}
```

Combiner and the Reducer (M step)

X = 0.05

Cluster Number	Images in the Entire Cluster	Images which are considered (m)	Major labels of the central images	Correctly clustered image	Classification Accuracy
0	1430	71	3	1005	0.70
1	911	45	2	512	0.56
2	1294	64	7	772	0.60
3	1235	61	8	738	0.60
4	792	39	1	525	0.66
5	1040	52	4	578	0.56

6	1119	55	6	769	0.69
7	501	25	8	193	0.39
8	774	38	0	612	0.79
9	904	45	9	602	0.67
Total	10000			6306	0.63

X = 0.1

Cluster Number	Images in the Entire Cluster	Images which are considered (m)	Major labels of the central images	Correctly clustered image	Classification Accuracy
0	1430	143	3	1005	0.70
1	911	91	2	512	0.56
2	1294	129	7	772	0.60
3	1235	123	8	738	0.60
4	792	79	1	525	0.66
5	1040	104	4	578	0.56
6	1119	119	6	769	0.69
7	501	50	8	193	0.39
8	774	77	0	612	0.79
9	904	90	9	602	0.67
Total	10000			6306	0.63

X = 0.5

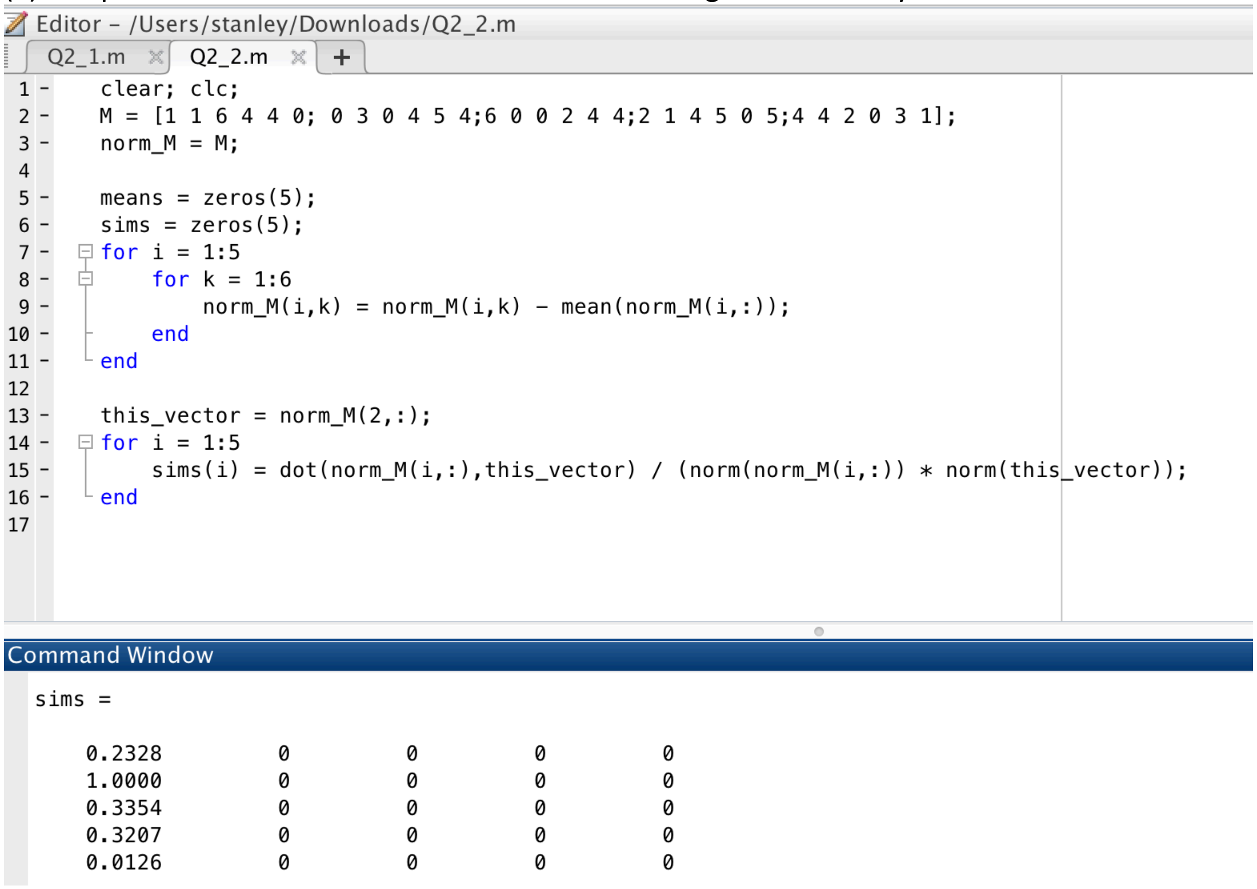
Cluster Number	Images in the Entire Cluster	Images which are considered (m)	Major labels of the central images	Correctly clustered image	Classification Accuracy
0	1430	715	3	1005	0.70
1	911	456	2	512	0.56
2	1294	647	7	772	0.60
3	1235	617	8	738	0.60
4	792	396	1	525	0.66
5	1040	570	4	578	0.56
6	1119	559	6	769	0.69
7	501	250	9	301	0.39

8	774	387	0	612	0.79
9	904	452	9	602	0.67
Total	10000			6412	0.64

EM-GMM performs better with PCA. The final performance reaches 0.64 when x =0.5. The result is probably because the cluster shape is more likely to be complex Gaussian mixture in reduced dimensional space. Note that 5 is absent in the clustering centroid and 9 is repeated.

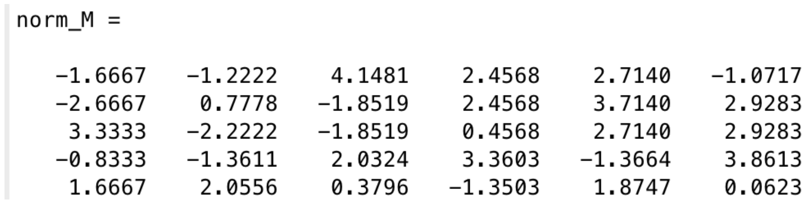
Q2 Recommendation System

(a) I implement the Item-to-Item collaborative filtering. The similarity is like:



```
clear; clc;
M = [1 1 6 4 4 0; 0 3 0 4 5 4;6 0 0 2 4 4;2 1 4 5 0 5;4 4 2 0 3 1];
norm_M = M;

means = zeros(5);
sims = zeros(5);
for i = 1:5
    for k = 1:6
        norm_M(i,k) = norm_M(i,k) - mean(norm_M(i,:));
    end
end

this_vector = norm_M(2,:);
for i = 1:5
    sims(i) = dot(norm_M(i,:),this_vector) / (norm(norm_M(i,:)) * norm(this_vector));
end
```

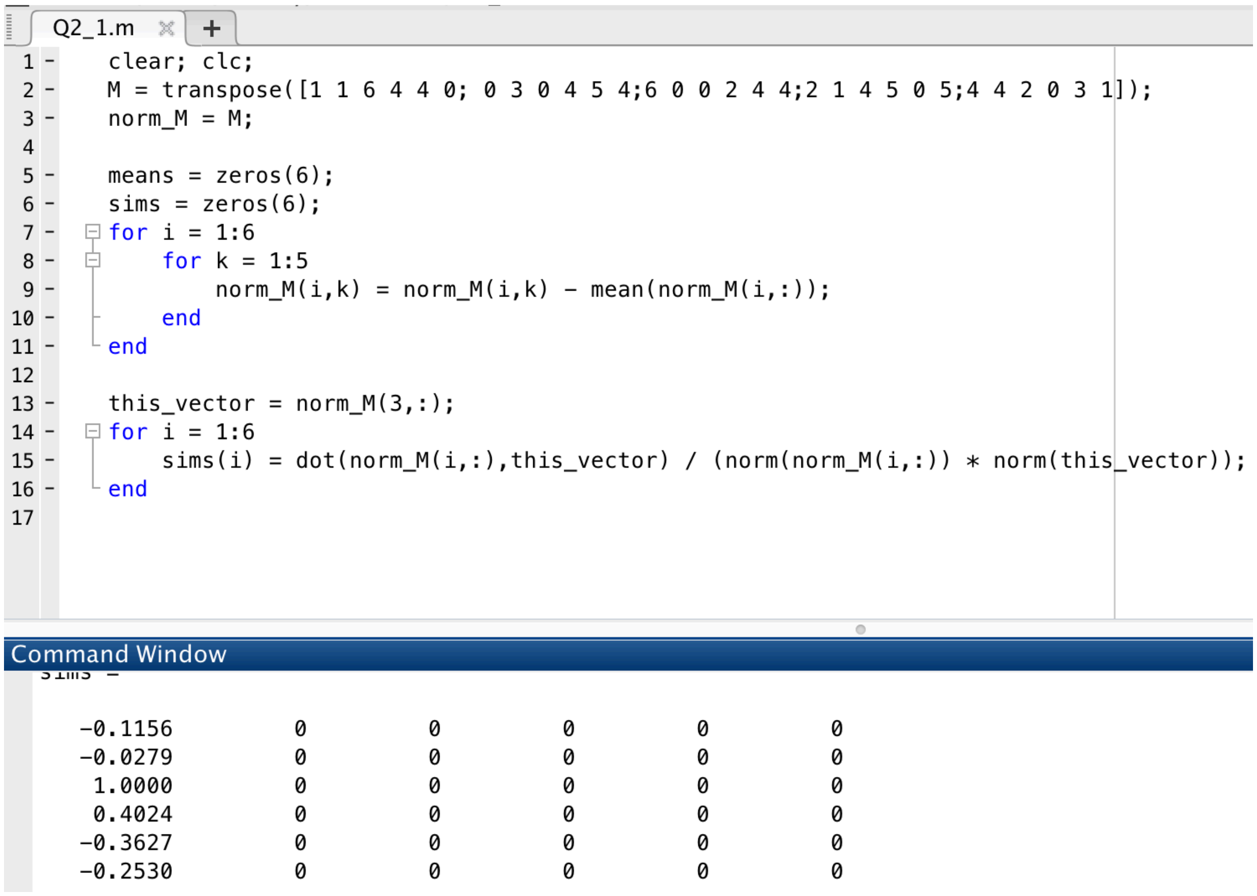
```
sims =

    0.2328         0         0         0         0
    1.0000         0         0         0         0
    0.3354         0         0         0         0
    0.3207         0         0         0         0
    0.0126         0         0         0         0
```



```
norm_M =

   -1.6667   -1.2222    4.1481    2.4568    2.7140   -1.0717
   -2.6667    0.7778   -1.8519    2.4568    3.7140    2.9283
    3.3333   -2.2222   -1.8519    0.4568    2.7140    2.9283
   -0.8333   -1.3611    2.0324    3.3603   -1.3664    3.8613
    1.6667    2.0556    0.3796   -1.3503    1.8747    0.0623
```

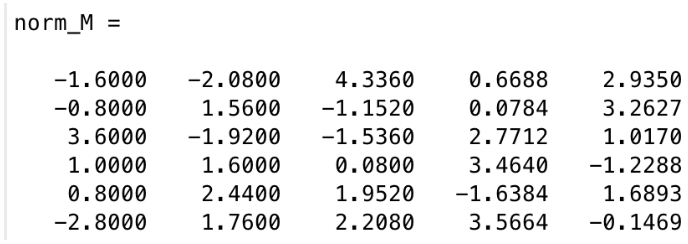
Because the last similarity is quite close to zero. I choose 0.3 as the threshold and choose user 3 and 4 as the most similar user and count their average rating to movie C, which is 4.5 points.

(b) I implement the User-to-User collaborative filtering. The similarity is like:



```
clear; clc;
M = transpose([1 1 6 4 4 0; 0 3 0 4 5 4;6 0 0 2 4 4;2 1 4 5 0 5;4 4 2 0 3 1]);
norm_M = M;

means = zeros(6);
sims = zeros(6);
for i = 1:6
    for k = 1:5
        norm_M(i,k) = norm_M(i,k) - mean(norm_M(i,:));
    end
end

this_vector = norm_M(3,:);
for i = 1:6
    sims(i) = dot(norm_M(i,:),this_vector) / (norm(norm_M(i,:)) * norm(this_vector));
end
```

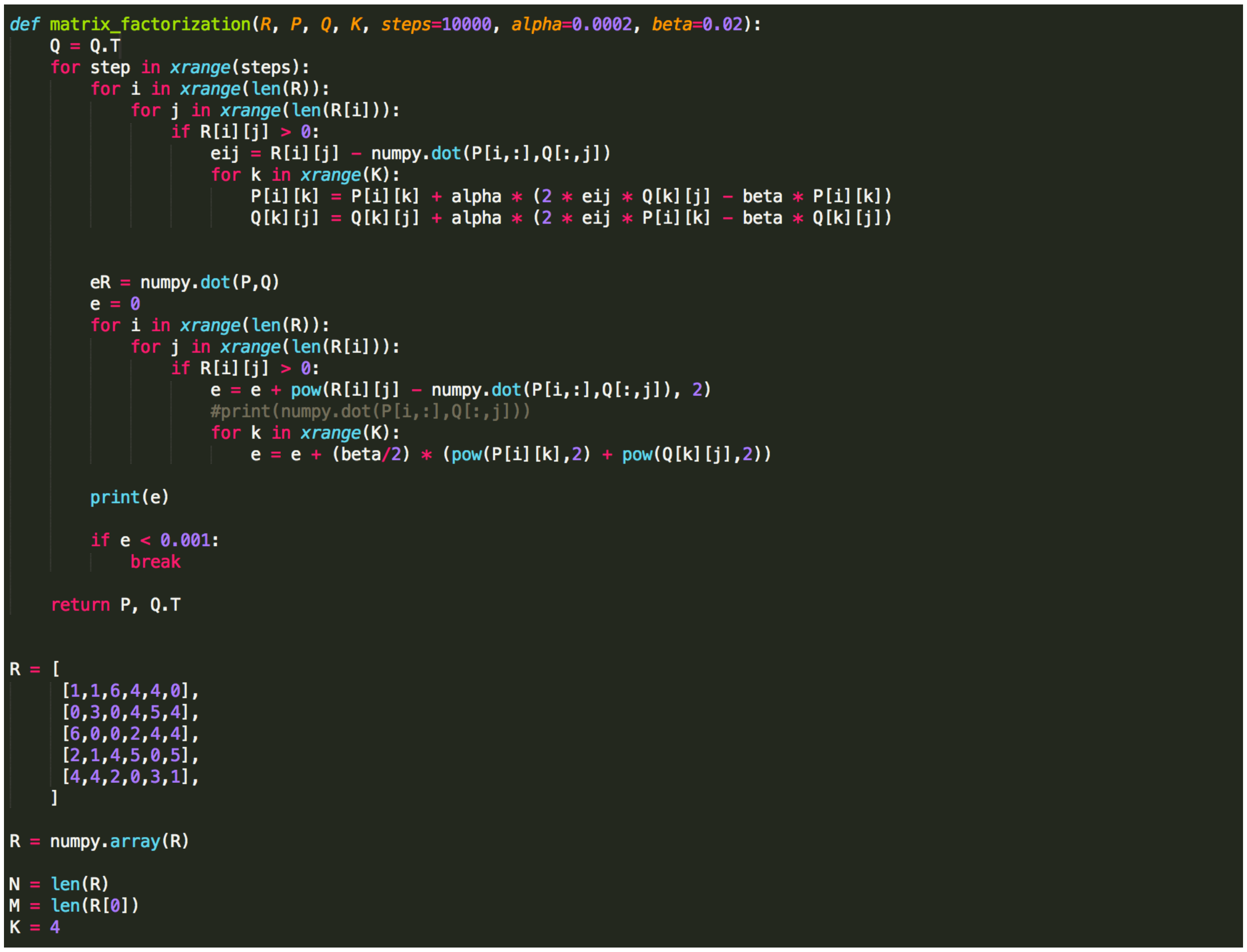
```
sims =

   -0.1156         0         0         0         0         0
   -0.0279         0         0         0         0         0
    1.0000         0         0         0         0         0
    0.4024         0         0         0         0         0
   -0.3627         0         0         0         0         0
   -0.2530         0         0         0         0         0
```



```
norm_M =

   -1.6000   -2.0800    4.3360    0.6688    2.9350
   -0.8000    1.5600   -1.1520    0.0784    3.2627
    3.6000   -1.9200   -1.5360    2.7712    1.0170
    1.0000    1.6000    0.0800    3.4640   -1.2288
    0.8000    2.4400    1.9520   -1.6384    1.6893
   -2.8000    1.7600    2.2080    3.5664   -0.1469
```

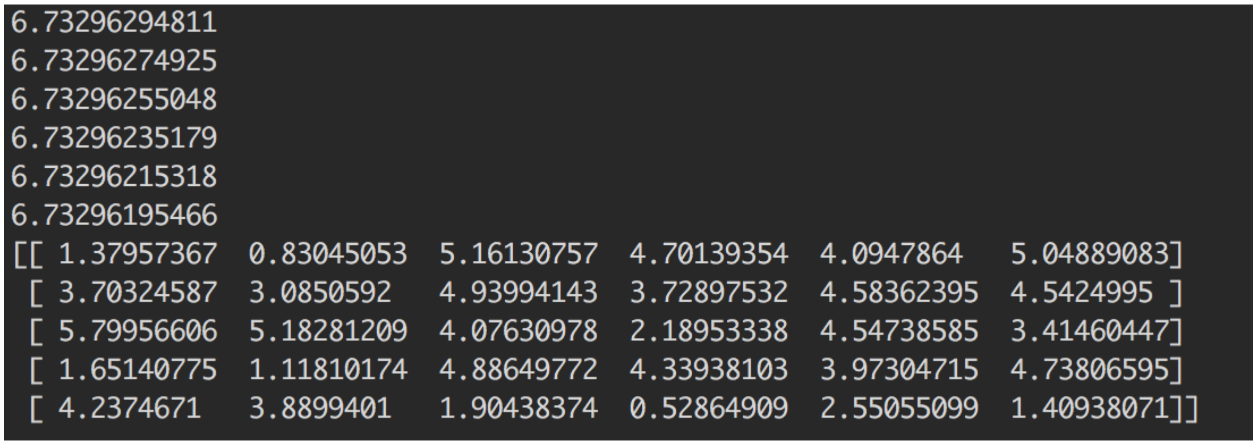
Because the rest is all negative, I choose the only positive item (Movie D) as the most-similar candidate. The predicted rating should be 4.7.

(c) Here I test with the number of latent factors to be 2,3,4,5. I set the maximum iteration to 10000 and try to see what latent factor number results in the fastest gradient descent.



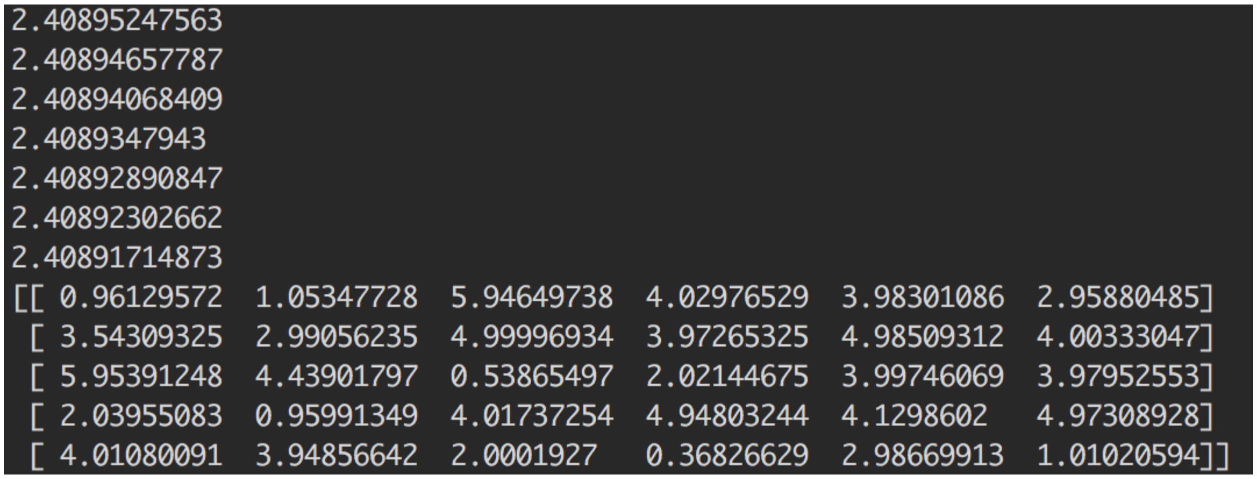
```
def matrix_factorization(R, P, Q, K, steps=10000, alpha=0.0002, beta=0.02):
    Q = Q.T
    for step in xrange(steps):
        for i in xrange(len(R)):
            for j in xrange(len(R[i])):
                if R[i][j] > 0:
                    eij = R[i][j] - numpy.dot(P[i,:],Q[:,j])
                    for k in xrange(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])


        eR = numpy.dot(P,Q)
        e = 0
        for i in xrange(len(R)):
            for j in xrange(len(R[i])):
                if R[i][j] > 0:
                    e = e + pow(R[i][j] - numpy.dot(P[i,:],Q[:,j]), 2)
                    #print(numpy.dot(P[i,:],Q[:,j]))
                    for k in xrange(K):
                        e = e + (beta/2) * (pow(P[i][k],2) + pow(Q[k][j],2))

        print(e)

        if e < 0.001:
            break

    return P, Q.T


R = [
     [1,1,6,4,4,0],
     [0,3,0,4,5,4],
     [6,0,0,2,4,4],
     [2,1,4,5,0,5],
     [4,4,2,0,3,1],
    ]

R = numpy.array(R)

N = len(R)
M = len(R[0])
K = 4
```

Latent factor: N=2, final loss is 6.73.



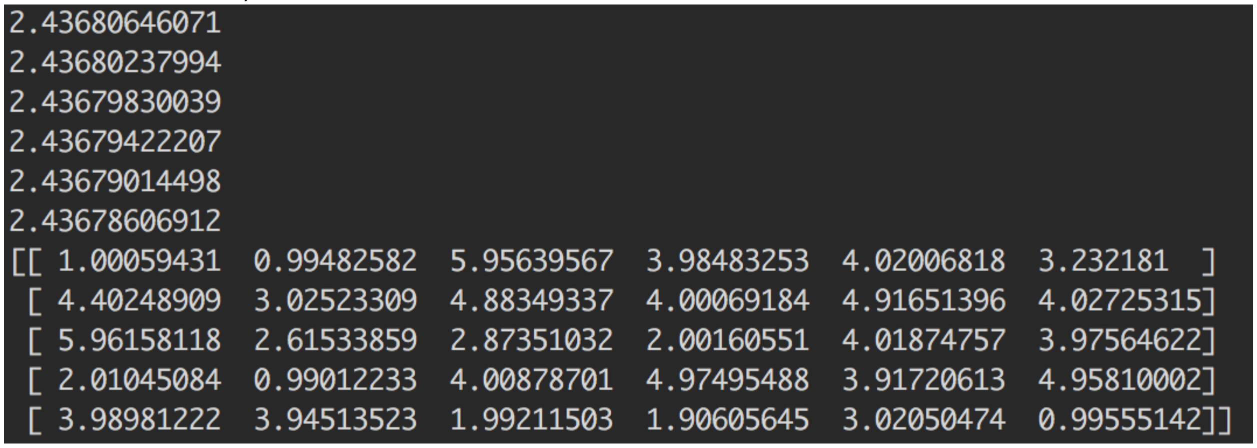
```
6.73296294811
6.73296274925
6.73296255048
6.73296235179
6.73296215318
6.73296195466
[[ 1.37957367  0.83045053  5.16130757  4.70139354  4.0947864   5.04889083]
 [ 3.70324587  3.0850592   4.93994143  3.72897532  4.58362395  4.5424995 ]
 [ 5.79956606  5.18281209  4.07630978  2.18953338  4.54738585  3.41460447]
 [ 1.65140775  1.11810174  4.88649772  4.33938103  3.97304715  4.73806595]
 [ 4.2374671   3.8899401   1.90438374  0.52864909  2.55055099  1.40938071]]
```

Latent factor: N=3, final loss is 2.40.



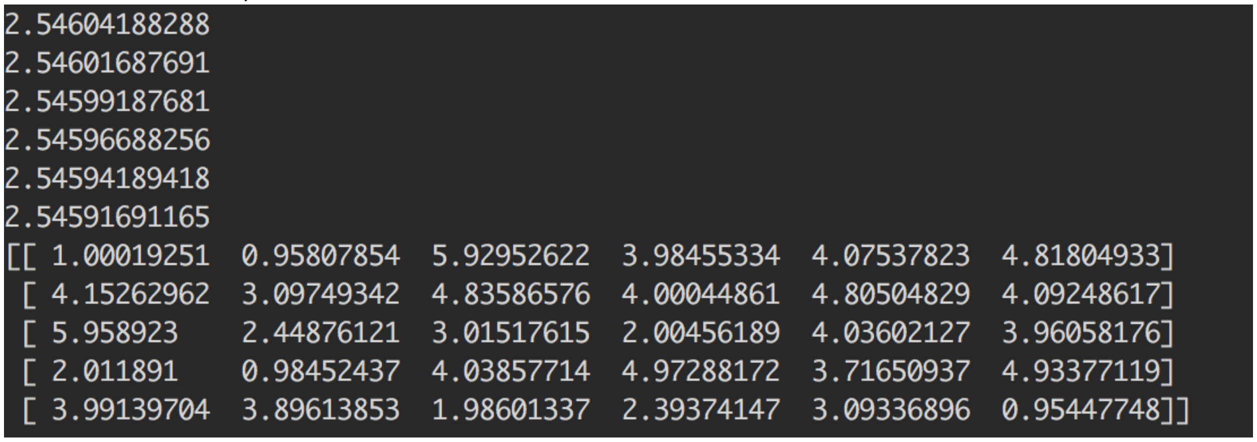
```
2.40895247563
2.40894657787
2.40894068409
2.4089347943
2.40892890847
2.40892302662
2.40891714873
[[ 0.96129572  1.05347728  5.94649738  4.02976529  3.98301086  2.95880485]
 [ 3.54309325  2.99056235  4.99996934  3.97265325  4.98509312  4.00333047]
 [ 5.95391248  4.43901797  0.53865497  2.02144675  3.99746069  3.97952553]
 [ 2.03955083  0.95991349  4.01737254  4.94803244  4.1298602   4.97308928]
 [ 4.01080091  3.94856642  2.0001927   0.36826629  2.98669913  1.01020594]]
```

Latent factor: N=4, final loss is 2.43.



```
2.43680646071
2.43680237994
2.43679830039
2.43679422207
2.43679014498
2.43678606912
[[ 1.00059431  0.99482582  5.95639567  3.98483253  4.02006818  3.232181  ]
 [ 4.40248909  3.02523309  4.88349337  4.00069184  4.91651396  4.02725315]
 [ 5.96158118  2.61533859  2.87351032  2.00160551  4.01874757  3.97564622]
 [ 2.01045084  0.99012233  4.00878701  4.97495488  3.91720613  4.95810002]
 [ 3.98981222  3.94513523  1.99211503  1.90605645  3.02050474  0.99555142]]
```

Latent factor: N=5, final loss is 2.54.

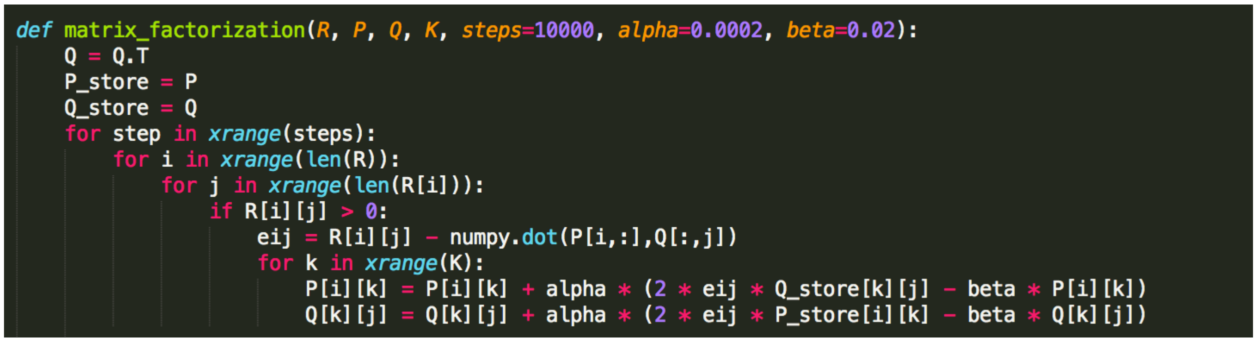


```
2.54604188288
2.54601687691
2.54599187681
2.54596688256
2.54594189418
2.54591691165
[[ 1.00019251  0.95807854  5.92952622  3.98455334  4.07537823  4.81804933]
 [ 4.15262962  3.09749342  4.83586576  4.00044861  4.80504829  4.09248617]
 [ 5.958923    2.44876121  3.01517615  2.00456189  4.03602127  3.96058176]
 [ 2.011891    0.98452437  4.03857714  4.97288172  3.71650937  4.93377119]
 [ 3.99139704  3.89613853  1.98601337  2.39374147  3.09336896  0.95447748]]
```

Consider the influence in L2-norm when we increase the number of N, I think N=3 is the most suitable case for this question. The final prediction is 4.99 points.

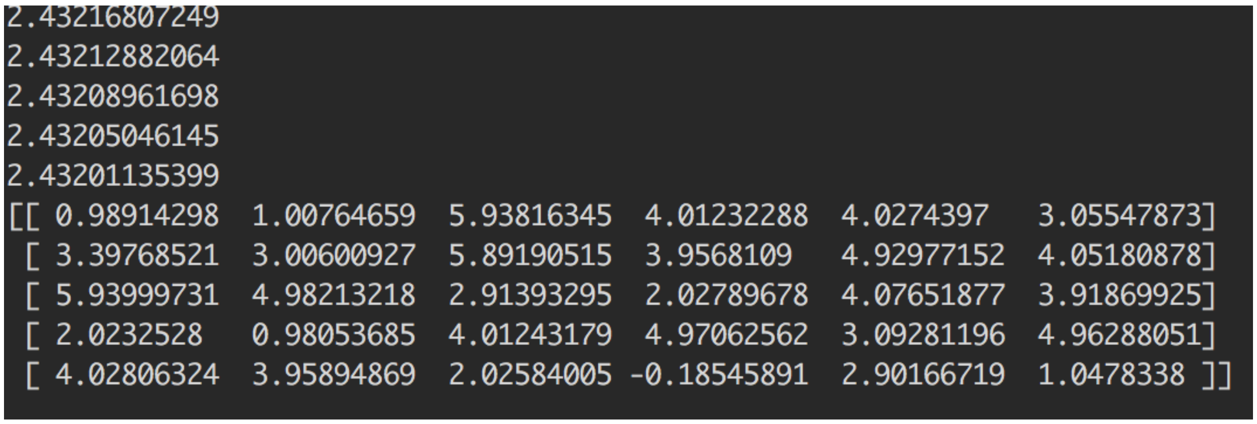
(d) One mistake is that when we are computing the gradient, the partial derivative should be computed at once. However, in the example, the partial derivative of P is already updated when we calculate the new number of Q.

I modify the original code and use P_store and Q_store to keep the original values in one step.



```
def matrix_factorization(R, P, Q, K, steps=10000, alpha=0.0002, beta=0.02):
    Q = Q.T
    P_store = P
    Q_store = Q
    for step in xrange(steps):
        for i in xrange(len(R)):
            for j in xrange(len(R[i])):
                if R[i][j] > 0:
                    eij = R[i][j] - numpy.dot(P[i,:],Q[:,j])
                    for k in xrange(K):
                        P[i][k] = P[i][k] + alpha * (2 * eij * Q_store[k][j] - beta * P[i][k])
                        Q[k][j] = Q[k][j] + alpha * (2 * eij * P_store[i][k] - beta * Q[k][j])
```

Final result when N=3:



```
2.43216807249
2.43212882064
2.43208961698
2.43205046145
2.43201135399
[[ 0.98914298  1.00764659  5.93816345  4.01232288  4.0274397   3.05547873]
 [ 3.39768521  3.00600927  5.89190515  3.9568109   4.92977152  4.05180878]
 [ 5.93999731  4.98213218  2.91393295  2.02789678  4.07651877  3.91869925]
 [ 2.0232528   0.98053685  4.01243179  4.97062562  3.09281196  4.96288051]
 [ 4.02806324  3.95894869  2.02584005 -0.18545891  2.90166719  1.0478338 ]]
```

The predicted points should be 5.89.