数字图像处理期末项目———基于PCA的人脸识别

姓名：高俊杰

学号：15331087

一、算法描述

这次项目人脸识别只要靠PCA算法实现。这种算法的主要思想就是，去除部分无关的或者关系较小的向量，保留影响较大的向量作基，这样即减少了基向量的数目从而减少了运算量，同时又减少了图像细节，能避免无关的向量和测试图像主人公的表情、脸朝向和配饰等变化对测试准确性产生不良干扰。算法的主要步骤如下（在这次项目的测试中我后来发现，不求平均脸的测试准确率反而更高，比求平均脸的做法准确率大约高上15%，所以过了一个多星期后又修改了一下原来的代码和报告）：

1.先求平均脸的做法(写于2018.1.27)

1.首先，应题目要求，对于所有的Faces集里的40个人，各随机取出其10张图像中的7张用作训练集，剩余3张用作后续的测试。这里我的做法是先将每个人的随机抽出的7张图像作一次平均从而得到一张平均后的图像。然后把40个人对应的平均图像都分别缓存起来，成为train_imgs。

2.然后，将40张train_imgs都拉伸成列向量并将所有列拼在一起，由于每张图像的总像素数都为10304，这样就得到了10304*40的矩阵X。

3.X的每列再减去均值向量，从而中心化。

4.求出X的转置和X的矩阵乘积，并求出乘积40*40矩阵的特征向量，这里用的是matlab的eig函数。

5.滤出前K大的特征值对应的特征向量W，再将X乘上W映射得到V，将V的每一列向量作为后续映射关系的一组基向量，共有K个基向量，也可以称为K个特征脸。

6.将X每一列都通过基向量矩阵V映射到对应的特征空间中。这样相当于将每张图像train_imgs都在新的空间中找到了对应的位置。

7.对于每个测试图像，也进行类似上述的变换：转成列向量，减去均值向量而中心化，然后用基向量矩阵映射到特征空间中。

8.要判断测试图像和40张train_imgs的哪张最匹配，只需对比测试图像在特征空间的新坐标和40张train_imgs在特征空间的坐标直接的欧几里得距离（或二范数）的大小，找到二范数最小的对应的train_img，就找到了最匹配训练图像了。

2.不求平均脸的做法(补充于2018.2.5)

不同点在上述步骤1中，即40个人的7张图像都保存下来而不是保存它们的均值图像，而算法的后续过程则完全一致，不过要修改一下部分预先设置的常量参数。

二.Matlab代码

1.先求平均脸的代码(写于2018.1.27)

函数Get_Training_Set.m(用来随机读取40*7张图像并分别作平均产生训练集)：

```
function [ mean_img ] = Get_Training_Set( input_path, index, height, width, output_path
    imgs = zeros(length(index), height, width);
    for i = 1 : length(index)
        imgs(i, :, :) = imread([input_path '/' num2str(index(i)) '.pgm']);
    end
    mean_img = zeros(height, width);
    for i = 1 : height
       for j = 1 : width
           mean_img(i, j) = sum(imgs(:, i, j)) / length(index);
       end
    end
    mean_img = uint8(mean_img);
%     imwrite(mean_img, [output_path '.bmp'], 'bmp');
end
```

函数Test_Case.m（用来测试单张图像，循环调用即可测试所有的测试集图像）：

```
function [ found ] = Test_Case( V, eigenfaces, indexes, i, j, mean_img )
    f = imread(['Faces/S' num2str(i) '/' num2str(indexes(i, j)) '.pgm']);
    [height, width] = size(f);
    f = double(reshape(f, [height * width, 1])) - mean_img;
    f = V' * f;
    [~, N] = size(eigenfaces);
    distance = Inf;
    found = 0;
    for k = 1 : N
        d = norm(double(f) - eigenfaces(:, k), 2);
        if distance > d
            found = k;
            distance = d;
        end
    end
end
```

脚本Eigenface.m（直接运行即可，调用了上述函数，结果数据中accuracy即为单次运行的准确百分比）：

```
N = 40;
K = 10;
height = 112;
```

```
width = 92;

indexes = zeros(N, 10);
train_imgs = zeros(N, height, width);
for i = 1 : N
    indexes(i, :) = randperm(10);
    train_imgs(i, :, :) = Get_Training_Set(['Faces/S' num2str(i)], indexes(i, 1 : 7),
    height, width, ['TrainSet/' num2str(i)]);
end
train_imgs = uint8(train_imgs);

X = zeros(height * width, N);
for i = 1 : N
    X(:, i) = reshape(train_imgs(i, :, :), [height * width, 1]);
end
mean_img = mean(X, 2);
for i = 1: N
   X(:, i) = X(:, i) - mean_img;
end
L = X' * X;
[W, D] = eig(L);
% e = mapminmax(X * W, 0, 255);
% for i = 1 : N
%    imwrite(uint8(reshape(e(:, i), [height, width])), ['eigen/' num2str(i) '.bmp'], 'bm
% end
W = W(:, N - K + 1 : N);
V = X * W;
eigenfaces = V' * X;

accuracy = 0;
for i = 1 : N
    for j = 8 : 10
%         subplot(6, 40, i + (j - 8) * 80);
%         imshow(f);
%         title(['S' num2str(i)]);
        found = Test_Case(V, eigenfaces, indexes, i, j, mean_img);
        if found == i
            accuracy = accuracy + 1;
        end
%         subplot(6, 40, i + 40 + (j - 8) * 80);
%         imshow(['TrainSet/' num2str(found) '.bmp']);
%         title(['S' num2str(found)]);
    end
end
accuracy = double(accuracy / (3 * N));
```

2.不求平均脸的代码(补充于2018.2.5)

函数Get_Training_Set.m

```
function [ imgs ] = Get_Training_Set( input_path, index, height, width, output_path )
    imgs = zeros(length(index), height, width);
    for i = 1 : length(index)
        imgs(i, :, :) = uint8(imread([input_path '/' num2str(index(i)) '.pgm']));
    end
end
```

函数Test_Case.m的代码不变，和求平均脸的做法一样。

脚本Eigenface.m（直接运行即可，并且直接写好了对K取值50-100每个整数值的测试，每个K都将测试100次，故运行需要一定时间）：

```
N = 7 * 40;
K = 90;
Test_Num = 3 * 40;
height = 112;
width = 92;

Test_Times = 100;
accuracy = zeros(100, Test_Times);
for K = 50 : 100
    for T = 1 : Test_Times
        indexes = zeros(N, 10);
        train_imgs = zeros(N, height, width);
        for i = 1 : 40
            indexes(i, :) = randperm(10);
            train_imgs((i - 1) * 7 + 1 : i * 7, :, :) = Get_Training_Set(['Faces/S' num2
        end
        train_imgs = uint8(train_imgs);

        X = zeros(height * width, N);
        for i = 1 : N
            X(:, i) = reshape(train_imgs(i, :, :), [height * width, 1]);
        end
        mean_img = mean(X, 2);
        for i = 1: N
           X(:, i) = X(:, i) - mean_img;
```

```
        end
        L = X' * X;
        [W, D] = eig(L);
        W = W(:, N - K + 1 : N);
        V = X * W;
        eigenfaces = V' * X;

        for i = 1 : 40
            for j = 8 : 10
                found = Test_Case(V, eigenfaces, indexes, i, j, mean_img);
                found = floor((found - 1) / 7) + 1;
                if found == i
                    accuracy(K, T) = accuracy(K, T) + 1;
                end
            end
        end
        accuracy(K, T) = double(accuracy(K, T) / (Test_Num));
    end
end
Mean_Accuracy = mean(accuracy, 2);
plot(Mean_Accuracy);
axis([50 100 0.9 1]);
```

三、测试性能表格

1.先求平均脸的做法(写于2018.1.27)

循环执行上述脚本，对于每个K值（取值在1~40，因为对每7张图像已预先做了一次平均，总图像数目降为40，不再是280，故不能照取50~100）测试分别测试了5次，每次120张测试图像（运行时间较长，故不测试太多次），得到的准确率如下：

1	0.13458333	0.20125	0.17625	0.14291667	0.15958333	0.162916667	
2	0.39465278	0.39465278	0.39465278	0.45298611	0.36965278	0.401319444	
3	0.51270833	0.51270833	0.52104167	0.579375	0.54604167	0.534375	
4	0.62986111	0.57986111	0.63819444	0.54652778	0.59819444	0.596527778	
5	0.61347222	0.65513889	0.63847222	0.69680556	0.65513889	0.651805556	
6	0.73902778	0.69736111	0.65569444	0.72236111	0.71402778	0.705694444	
7	0.74770833	0.74770833	0.70604167	0.714375	0.689375	0.721041667	
8	0.72298611	0.80631944	0.73131944	0.73131944	0.83131944	0.764652778	
9	0.83159722	0.78159722	0.74826389	0.71493056	0.83159722	0.781597222	
10	0.72326389	0.78993056	0.79826389	0.78993056	0.78159722	0.776597222	
11	0.76493056	0.75659722	0.74826389	0.79826389	0.76493056	0.766597222	
12	0.72340278	0.74840278	0.79840278	0.69840278	0.75673611	0.745069444	
13	0.77340278	0.82340278	0.79006944	0.79840278	0.79840278	0.796736111	
14	0.79006944	0.79840278	0.78173611	0.78173611	0.79006944	0.788402778	
15	0.74013889	0.84013889	0.81513889	0.79013889	0.79847222	0.796805556	
16	0.75680556	0.77347222	0.84013889	0.79013889	0.77347222	0.786805556	
17	0.79847222	0.74847222	0.75680556	0.73180556	0.77347222	0.761805556	
18	0.76513889	0.79847222	0.79847222	0.81513889	0.81513889	0.798472222	
19	0.73180556	0.78180556	0.74847222	0.79013889	0.81513889	0.773472222	
20	0.79847222	0.75680556	0.75680556	0.81513889	0.75680556	0.776805556	
21	0.76513889	0.79013889	0.78180556	0.78180556	0.78180556	0.780138889	
22	0.74847222	0.81513889	0.78180556	0.77347222	0.77347222	0.778472222	
23	0.79013889	0.79847222	0.76513889	0.77347222	0.79847222	0.785138889	
24	0.78180556	0.81513889	0.81513889	0.77347222	0.85680556	0.808472222	
25	0.80680556	0.82347222	0.75680556	0.78180556	0.79013889	0.791805556	
26	0.77347222	0.75680556	0.86513889	0.75680556	0.79013889	0.788472222	
27	0.79847222	0.75680556	0.84847222	0.83180556	0.81513889	0.810138889	
28	0.78180556	0.73180556	0.72347222	0.83180556	0.76513889	0.766805556	
29	0.84013889	0.82347222	0.79013889	0.84013889	0.72347222	0.803472222	
30	0.83180556	0.81513889	0.77347222	0.77347222	0.78180556	0.795138889	
31	0.76513889	0.74847222	0.75680556	0.84013889	0.77347222	0.776805556	
32	0.79013889	0.81513889	0.77347222	0.79013889	0.86513889	0.806805556	
33	0.78180556	0.81513889	0.79013889	0.79013889	0.74013889	0.783472222	
34	0.80680556	0.76513889	0.81513889	0.83180556	0.82347222	0.808472222	
35	0.80680556	0.79847222	0.83180556	0.74847222	0.77347222	0.791805556	
36	0.79013889	0.80680556	0.75680556	0.82347222	0.80680556	0.796805556	
37	0.78180556	0.79847222	0.80680556	0.79013889	0.77347222	0.790138889	
38	0.80680556	0.79847222	0.80680556	0.82347222	0.75680556	0.798472222	
39	0.79013889	0.79847222	0.87347222	0.82347222	0.84013889	0.825138889	
40	0.74847222	0.80680556	0.83180556	0.75680556	0.80680556	0.790138889	
41			准确率			均值	
42							

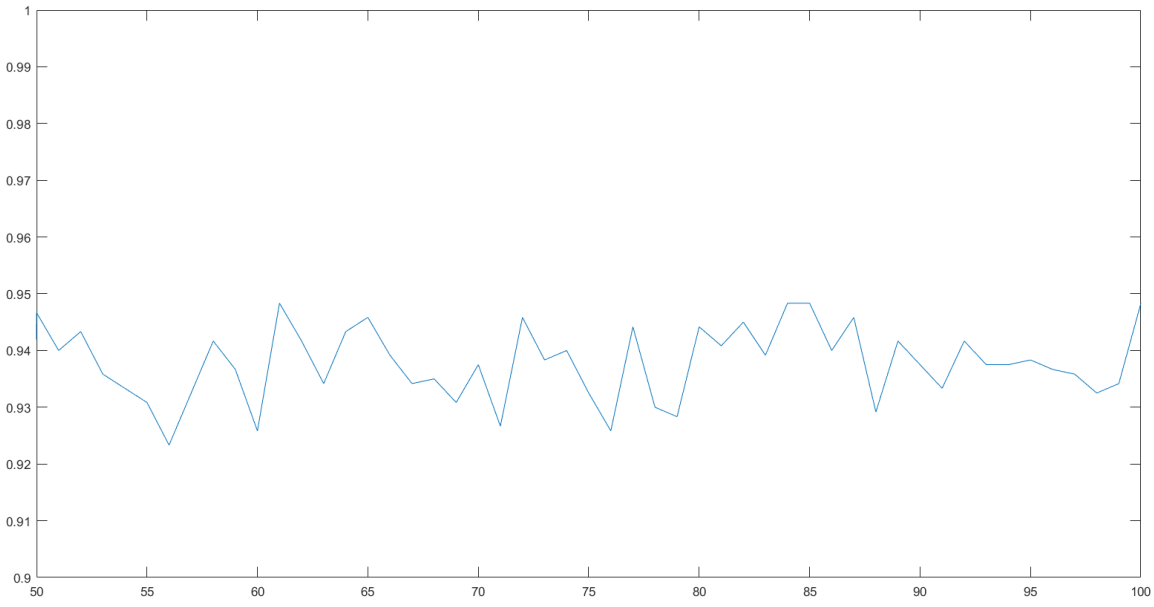
可以看到，随着K增大，准确率先是减速地上升，然后K达到约7以后，准确率的均值就基本在70%到80%间徘徊，部分时候会超过80%。在这里测试的噪声估计不强，所以K较大时准确率还是基本不受影响。

2.不求平均脸的做法(补充于2018.2.5)

如下，Test_Case函数返回1-280之间的整数，表示这一列最能匹配当前测试图片。但还需要转成1-40之间的整数才能表示出匹配到了的是哪一个人。

```
found = Test_Case(V, eigenfaces, indexes, i, j, mean_img);
found = floor((found - 1) / 7) + 1;
```

修改代码为初始不求40张平均脸，而用280张人脸直接训练后再对每个K值测试10次，得到的测试折线图如下，可以看到，测试准确度随着K从50到100取值，大约在93%到95%间波动，并且大约在K取83的附近得到较稳定的最大值：



测试表格：

	1	2	3	4	5	6	7	8	9	10
50	0.9667	0.9083	0.9583	0.9667	0.9667	0.9417	0.9667	0.9167	0.9583	0.9500
51	0.8833	0.9667	0.9417	0.9583	0.9250	0.9333	0.9583	0.9000	0.9750	0.9333
52	0.9417	0.9167	0.9667	0.9250	0.9333	0.9500	0.9083	0.9500	0.9083	0.9750
53	0.9250	0.9333	0.8917	0.9500	0.9583	0.9167	0.9750	0.9500	0.9417	0.9417
54	0.9083	0.9500	0.9333	0.9333	0.9250	0.9417	0.9333	0.9667	0.9333	0.9500
55	0.9750	0.9083	0.9417	0.9667	0.9500	0.9583	0.9333	0.9500	0.9000	0.9333
56	0.9250	0.9417	0.9167	0.9417	0.9000	0.9333	0.9417	0.9417	0.9583	0.9333
57	0.8917	0.9333	0.9000	0.9667	0.9583	0.9500	0.9417	0.9833	0.9250	0.9500
58	0.9417	0.9250	0.9250	0.9750	0.9583	0.9417	0.9250	0.9333	0.9500	0.9333
59	0.9583	0.9750	0.9250	0.9500	0.9667	0.9500	0.9333	0.9417	0.9333	0.9333
60	0.9250	0.9500	0.9833	0.9250	0.9250	0.9500	0.9250	0.9333	0.9250	0.9000
61	0.9583	0.9417	0.9333	0.9417	0.9000	0.9500	0.9333	0.9833	0.9583	0.9667
62	0.9500	0.9167	0.9333	0.9167	0.9250	0.9583	0.9167	0.9333	0.9417	0.9250
63	0.9333	0.9500	0.9417	0.9333	0.9833	0.9417	0.9417	0.9667	0.9333	0.9000
64	0.9750	0.9417	0.9250	0.9167	0.9417	0.9583	0.9500	0.9250	0.9583	0.9500
65	0.9167	0.9500	0.9500	0.9333	0.9917	0.9167	0.9250	0.9667	0.9250	0.9250
66	0.9500	0.9250	0.9500	0.9500	0.9083	0.9417	0.9500	0.9250	0.9167	0.9333
67	0.9333	0.9083	0.9500	0.9750	0.9250	0.9583	0.9250	0.9500	0.9250	0.9583
68	0.8917	0.9750	0.9667	0.9250	0.9167	0.9250	0.9667	0.9500	0.9333	0.9083
69	0.9083	0.9250	0.9417	0.9500	0.9250	0.9250	0.9500	0.9333	0.9833	0.9083
70	0.9417	0.9333	0.9333	0.9083	0.9417	0.9500	0.9583	0.9417	0.9500	0.9500
71	0.9083	0.9250	0.9500	0.9583	0.9417	0.9250	0.8917	0.9500	0.9667	0.9583
72	0.9583	0.9500	0.9667	0.9250	0.9500	0.9500	0.9500	0.9417	0.9500	0.9417
73	0.9167	0.9333	0.9583	0.9250	0.9500	0.9417	0.9250	0.9250	0.9250	0.9083
74	0.9417	0.9417	0.9167	0.9000	0.9583	0.9500	0.9667	0.9333	0.9250	0.9417
75	0.9333	0.9583	0.9417	0.9583	0.9083	0.9500	0.9583	0.9250	0.9667	0.9583

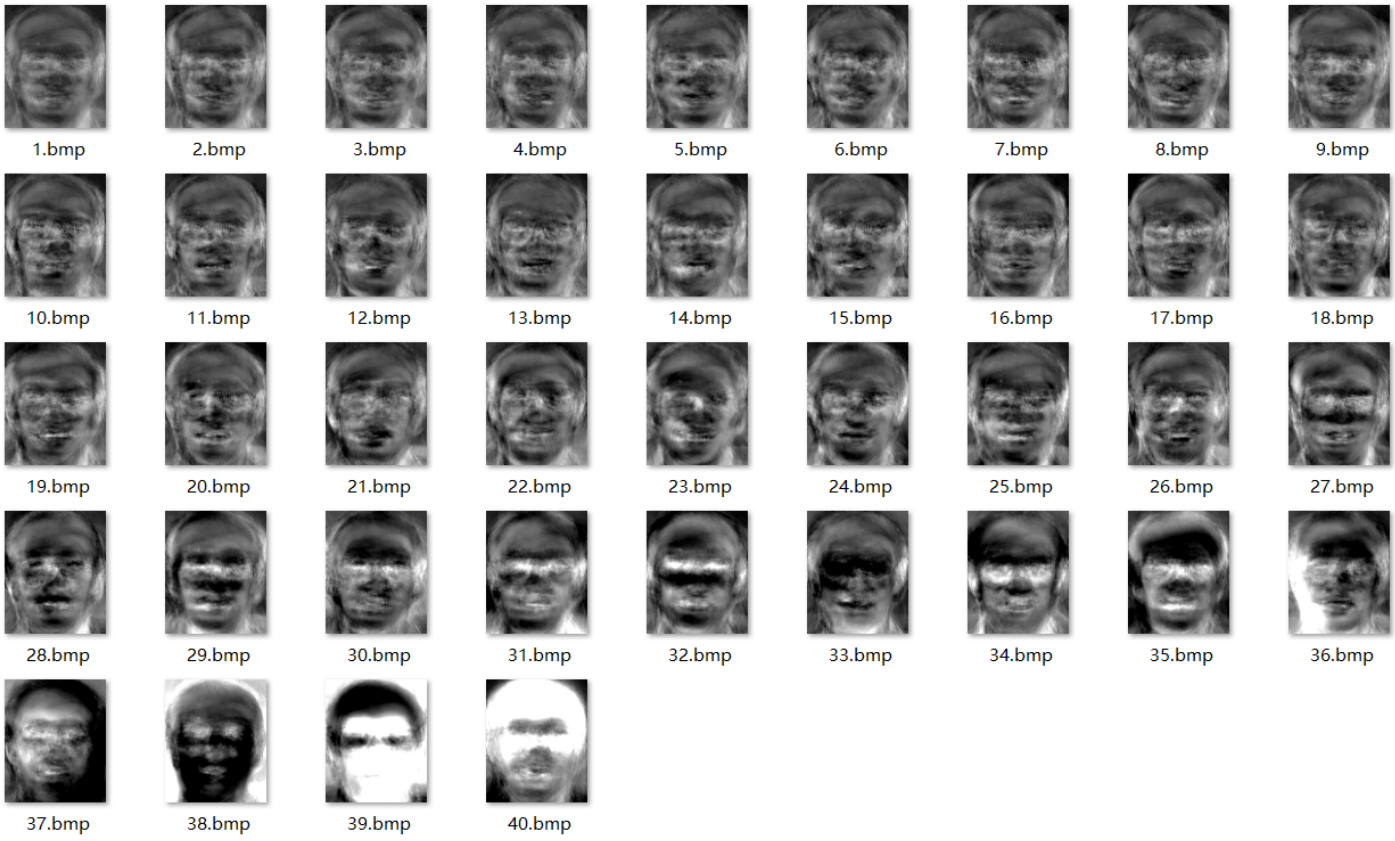
	1	2	3	4	5	6	7	8	9	10
76	0.9750	0.9667	0.9333	0.9583	0.9167	0.9583	0.9667	0.9417	0.9500	0.9167
77	0.9417	0.9583	0.9417	0.9583	0.9333	0.9083	0.8833	0.9417	0.9583	0.9500
78	0.9333	0.9167	0.9500	0.9333	0.9417	0.9167	0.9583	0.9000	0.9250	0.9000
79	0.9167	0.9500	0.9583	0.9250	0.9750	0.9500	0.9500	0.9500	0.8917	0.9250
80	0.9250	0.9417	0.9250	0.9167	0.9250	0.9500	0.9750	0.9500	0.9250	0.9583
81	0.9417	0.9417	0.9667	0.9417	0.9500	0.9500	0.9500	0.9250	0.9500	0.9000
82	0.9417	0.9167	0.9333	0.9500	0.9250	0.9500	0.9500	0.9583	0.9167	0.9583
83	0.9000	0.9167	0.9083	0.9667	0.9250	0.9583	0.9583	0.9167	0.9250	0.9167
84	0.9583	0.9250	0.9583	0.9500	0.9083	0.9167	0.9417	0.9667	0.9750	0.9500
85	0.9083	0.9333	0.9333	0.9333	0.9333	0.9417	0.9083	0.9417	0.9500	0.9333
86	0.9333	0.9333	0.9583	0.9417	0.9583	0.9250	0.9417	0.9417	0.9417	0.9000
87	0.9667	0.9417	0.9833	0.9833	0.9250	0.9750	0.9417	0.9583	0.9083	0.9583
88	0.9667	0.9333	0.9500	0.9250	0.9417	0.9417	0.9250	0.9167	0.9417	0.9417
89	0.9500	0.9583	0.9417	0.9500	0.9333	0.9083	0.9000	0.9333	0.9667	0.9500
90	0.9500	0.9333	0.9583	0.9833	0.9500	0.9167	0.9250	0.9083	0.9583	0.9667
91	0.9667	0.9583	0.9917	0.9667	0.9333	0.9167	0.9583	0.9167	0.9333	0.9333
92	0.8917	0.9833	0.9250	0.9583	0.9667	0.9250	0.9167	0.9333	0.9250	0.9333
93	0.9833	0.9500	0.9333	0.9250	0.9250	0.9583	0.9417	0.9083	0.9000	0.9500
94	0.9333	0.9583	0.9167	0.9250	0.9417	0.9083	0.9417	0.9333	0.9750	0.9500
95	0.9417	0.9417	0.9500	0.9583	0.9667	0.9667	0.9583	0.9333	0.9250	0.9417
96	0.9667	0.9250	0.9083	0.9750	0.9083	0.9500	0.9167	0.9500	0.9083	0.9333
97	0.8917	0.9250	0.9583	0.9417	0.9667	0.9417	0.9000	0.9083	0.9083	0.9583
98	0.9417	0.8583	0.9417	0.8583	0.9750	0.9417	0.9667	0.9333	0.9250	0.9000
99	0.8583	0.9583	0.9333	0.9583	0.9167	0.9333	0.9583	0.9333	0.9667	0.9583
100	0.9667	0.9583	0.9167	0.9583	0.9250	0.9500	0.9167	0.9250	0.9333	0.9333

四、过程及结果图像

1.先求平均脸的做法(写于2018.1.27)

如图，最先通过对每人7张图像求均值得到的训练集图像如下：

如图，由上述40张平均图像通过上述算法转换而来的特征图像如下(已按特征值从小到大排序)：



下图是选取部分K的取值时的结果，奇数行为测试图像，偶数行为上一行输入所匹配的人脸均值图像。若测试行的标题和匹配行的标题一致，则说明匹配到了正确的人脸。

选取K为8时得到的结果图像1：

选取K为10时得到的结果图像2：

选取K为12时得到的结果图像3：

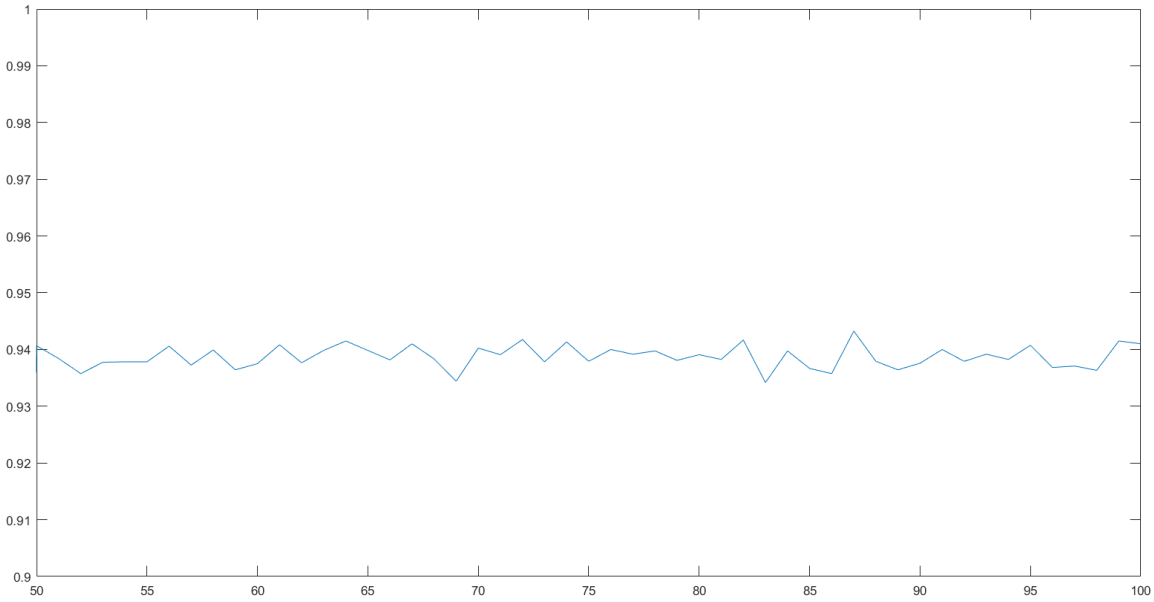
选取K为14时得到的结果图像4：

选取K为16时得到的结果图像5：

由于测试有一定的随机性，且测出来的准确率也较符合预期，故认为人脸识别的准确率在以上K的取值情况下变化不大，均在78%左右。同时，在随机选择训练集和测试集的情况下，经过数十次的测试，每次测试的120张图像的匹配结果大多能克服细节上的干扰（如仪态、角度、配饰等无关元素），故认为本次人脸识别项目的目标已基本达成。

2.不求平均脸的做法(补充于2018.2.5)

下图是用280张人脸直接训练后再对每个K值测试100次，得到的测试折线图如下，可以看到，测试次数增多后（10倍于之前的测试量），测试准确度随着K从50到100取值，在94%上下波动，并且大约在K取87的附近得到较稳定的最大值94.5%：



考虑到每个K值测试100次，不能完全排除偶然性，总体上K值对准确率的影响较不明显。

所以最终的测试结果表明，K取87左右，按上述做法可以得到的测试准确率大约为94.5%。