ANLY 503 Eigenface Assignment Write-up:

Yifan Wu (yw515)

This program will explore and test the PCA on the well-known case of EigenFace decomposition.

EigenFace Generation:

I downloaded all the face image, 10 for every subject and 40 subjects total, onto my local drive. They were located in the current directory in this format: ".\\FACEdata\\s1\\1.pgm".

(Each picture looks like the example above with pixel dimension 112 x 92)

I read in all the images in by listing all the file name in the current directory tree using os.walk, filtering out all that is an image(end with ".pgm"), and appending all the image paths into a list for later iteration.

```python
all_path = []
for root, d_names, f_names in os.walk('.'):
    for f in f_names:
        full_path = os.path.join(root, f)
        if full_path.endswith('.pgm'): all_path += [full_path]
```

Next, I created a function ("ReadAndNorm") that reads in a list of image path, row-wise appends them into a matrix, takes transpose of this matrix so each column is an image, and normalizes the whole face matrix by subtracting the mean of all faces. The ReadAndNorm function will return the original row face matrix, normalized column matrix, the mean face of the matrix, and the original dimension of each image.

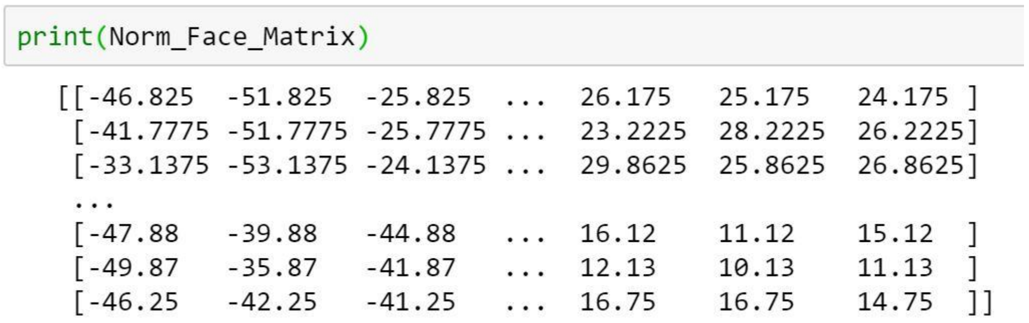
```python
def ReadAndNorm(all_path):
    Face_Matrix = np.empty((0,10304), int)
    for iamge_path in all_path:
        img = Image.open(iamge_path).convert('L')
        imagearray = np.array(img)
        original_shape = imagearray.shape
        flat = imagearray.ravel()
        facevector = np.matrix(flat)
        #Row_wize append face matrix
        Face_Matrix = np.r_[Face_Matrix, facevector]
    #Take transpose so each column is an image
    Face_Matrix = Face_Matrix.T
    #get mean face
    mean_face = Face_Matrix.sum(axis=1)/Face_Matrix.shape[1]
    #Normalize face
    Norm_Face_Matrix = Face_Matrix - mean_face
    return Face_Matrix, Norm_Face_Matrix, mean_face, original_shape

```

```
___, Norm_Face_Matrix, mean_face, original_shape = ReadAndNorm(all_path)
```

Note that the last line of the above code block calls the function and generates the normalized face matrix, the mean face of all the faces, and the original pixel shape of each image. The mean face across all the 400 faces should look like the following:

Mean Face Across All 400 Training Images



```
print(Norm_Face_Matrix)
```

```
[[-46.825   -51.825   -25.825   ...  26.175    25.175    24.175  ]
 [-41.7775  -51.7775  -25.7775  ...  23.2225   28.2225   26.2225]
 [-33.1375  -53.1375  -24.1375  ...  29.8625   25.8625   26.8625]
 ...
 [-47.88    -39.88    -44.88    ...  16.12     11.12     15.12   ]
 [-49.87    -35.87    -41.87    ...  12.13     10.13     11.13   ]
 [-46.25    -42.25    -41.25    ...  16.75     16.75     14.75   ]]
```

Normalized Column-wise Face Matrix

Next, I computed the Covariance Matrix. I took the dot product of the transpose of the normalized face matrix with itself, and the resulting Covariance Matrix have the shape of 400 x 400.

```
#Get Coveriance matrix
Norm_Face_Matrix_t = np.transpose(Norm_Face_Matrix)
CovMatrix = np.matmul(Norm_Face_Matrix_t, Norm_Face_Matrix)
```

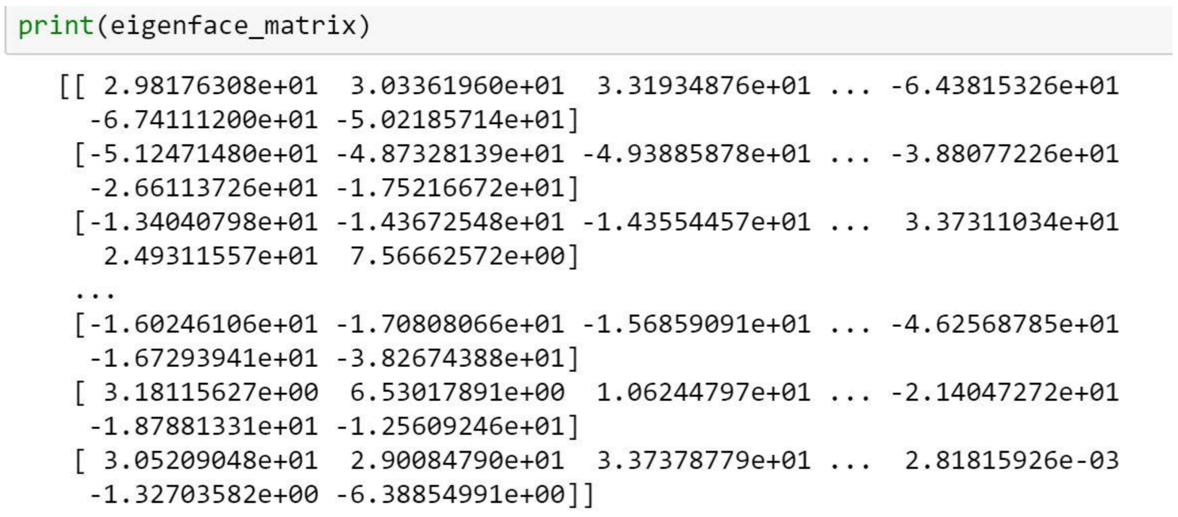
I computed the Eigen Vectors and Eigen Value from the Coveriance Matrix using np.linalg.eig():

```
#Get eigen vector and eigen value
evals, evects = np.linalg.eig(CovMatrix)
```

To get the most representative eigenfaces to be the new reduced face space, I need to sort out the k largest Eigen Values and their corresponding Eigen Vectors. Luckily, the result of np.linalg.eig() is already sorted in descending order of Eigen Values. I just need to multiple the first 30 Eigen Vectors with our normalized face matrix to generate our final EigenFace Matrix:

```python
#matrix of k eigen eigenvectors(eigenfaces matrix)
top30_eig_vecs = evects[:30]
eigenface_matrix = np.matmul(top30_eig_vecs, Norm_Face_Matrix.T)
```

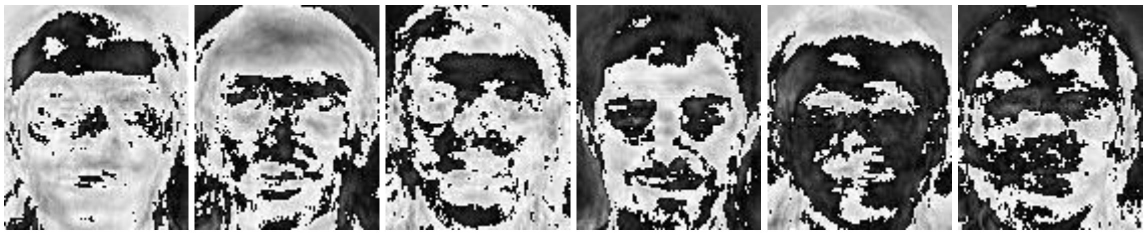
The resulting EigenFaces Matrix should look like the following with shape 30 x 10304:



```python
print(eigenface_matrix)
```

```
[[ 2.98176308e+01  3.03361960e+01  3.31934876e+01 ... -6.43815326e+01
  -6.74111200e+01 -5.02185714e+01]
 [-5.12471480e+01 -4.87328139e+01 -4.93885878e+01 ... -3.88077226e+01
  -2.66113726e+01 -1.75216672e+01]
 [-1.34040798e+01 -1.43672548e+01 -1.43554457e+01 ...  3.37311034e+01
   2.49311557e+01  7.56662572e+00]
 ...
 [-1.60246106e+01 -1.70808066e+01 -1.56859091e+01 ... -4.62568785e+01
  -1.67293941e+01 -3.82674388e+01]
 [ 3.18115627e+00  6.53017891e+00  1.06244797e+01 ... -2.14047272e+01
  -1.87881331e+01 -1.25609246e+01]
 [ 3.05209048e+01  2.90084790e+01  3.37378779e+01 ...  2.81815926e-03
  -1.32703582e+00 -6.38854991e+00]]
```

Here is the first 6 EigenFace images for taste:



They look roughly human but very unspecific in terms of facial characters.

Test:

Part 1:

I took the first image in the database as our test image in this part and copy it out of the database with the original database remains unchanged. The general concept behind this testing step is to check if the test image can recognize itself by obtaining vector distance of 0 with itself after the test image and all the other 399 images are projected into our new EigenFace space constructed with 30 EigenFaces. The purpose of this step is to test the code for later testing.

I read in the first image vector and subtract the mean face, which is generated in the beginning of the program using function ReadAndNorm.

```
#Read in test vector
test_vec, ____, ___, ____ = ReadAndNorm([all_path[0]])
test_norm_vec = test_vec - mean_face
```

After normalizing the test vector, I projected it onto the previously generated EigenFace space by taking the dot product of EigenFace Matrix and our normalized test vector:

```
eig_proj = np.matmul(eigenface_matrix, test_norm_vec)
```

The step is the same for projecting all 400 images onto EigenFace space:

```
all_eig_proj = np.matmul(eigenface_matrix, Norm_Face_Matrix)
```

The projected test vector is of shape 30 x 1 and the projected whole face set is of shape 30 x 400. Next we compute the Euclidean Distance from the projected test vector to each of the 400 image vectors in the projected whole face matrix:

```
from sklearn.metrics.pairwise import euclidean_distances
[round(float(i), 2) for i in euclidean_distances(all_eig_proj.T, eig_proj.T)]
```

The distance is the following list and we can see that the test image is recongnizable by having distance of 0 with itself after all of them being projected:

```
[0.0,
 13618728.39,
 20213906.1,
 24621675.39,
 21164827.42,
 22152256.96,
 15562756.49,
 11165893.14,
 16498005.34,
 19529965.61]
```

The code works!

Part 2:

Now I moved on to the final part of testing for the EigenFace methodology and see if it can correctly recognize our test image to the same person in the database. The logic to this second part of testing is very similar to the first part. The difference is we take out one of the faces to be the test face and replace the empty slot with any other facial vectors from the same person. Doing this will duplicate one face image for that person so we can keep the dimensionality of the original database. We generate EigenFace space of this new database, which has never seen our test image before, and project our test image as well as all the images in new database onto this EigenFace space. We measure the Euclidean Distance with projected test vector and the projected whole dataset and see if our model can recognize the subject of our test image. (Hint: If they belong to the same person, they should come from the same folder!)

Instead of pulling out the test image from the actual folder and filling the empty slot by copying any other image of the same person, I did a slight manipulation of the original list of image path so it will skip the selected test image path and replace it with any other path for the same person. The ReadAndNorm function will read in images according to this new path list and the resulting matrix will be equivalent to the resulting matrix if we hand move and copy those images on the local drive. Here I simply took the very first image as the test image and replace it with the second image in the dataset, which is also from the same person as the test image. In Python list operation that should be replacing the first path in the path list with path of the second image. The resulting list looks like: [path_2, path_2, path_3, …, path_400] as oppose to the original path list: [path_1, path_2, path_3, …, path_400]. I created a new path list called "all_path_test":

```
#takind the first image out and replace it with the second image(.\\FACESdata\\s1\\1.pg
m -->.\\FACESdata\\s1\\2.pgm)
all_path_test = all_path.copy()
all_path_test[0] = all_path[2]
all_path_test[:10] #See there are two 2.pgm in s1?
```

Resulting new image path list to read in images:

```
['.\\FACESdata\\s1\\2.pgm',
 '.\\FACESdata\\s1\\10.pgm',
 '.\\FACESdata\\s1\\2.pgm',
 '.\\FACESdata\\s1\\3.pgm',
 '.\\FACESdata\\s1\\4.pgm',
 '.\\FACESdata\\s1\\5.pgm',
 '.\\FACESdata\\s1\\6.pgm',
 '.\\FACESdata\\s1\\7.pgm',
 '.\\FACESdata\\s1\\8.pgm',
 '.\\FACESdata\\s1\\9.pgm']
```

Resulting test image path, note they both from the same folder s1 and hence the same person:

```
'.\\FACESdata\\s1\\1.pgm'
```

Now the replacement procedure is done. We can move on to testing. First, I had to compute the EigenFace Matrix by using ReadAndNorm function and projection FIRST before moving on to reading in the test image. This is because we need the mean face vectors generated in this first step to normalize the test image vector. The steps of generating new EigenFace Matrix is the same as before. In a nutshell:

Read in and normalize the whole modified database -> Compute Coveriance Matrix -> Compute Eigen Values and Eigen Vectors -> Project the original normalized database matrix onto the Eigen Vectors that corresponds to the largest 30 Eigen Values to get the new EigenFace Matrix:

```
#new eigenface matrix using dataset after replacement for test vector:
___, test_norm_face, test_mean_face, original_shape = ReadAndNorm(all_path_test)
#test_norm_face.T[:5] #We can see that the first image and the 2rd image(3rd element in
 the list) are the same.
test_norm_face_t = np.transpose(test_norm_face)
CovMatrix_test = np.matmul(test_norm_face_t, test_norm_face)
evals_test, evects_test = np.linalg.eig(CovMatrix_test)
```

```
top30_eig_vecs_test = evects_test[:30]
eigenface_matrix_test = np.matmul(top30_eig_vecs_test, test_norm_face_t)
```

The new EigenFace Matrix looks like this:

```
matrix([[ -5.75586222,  -3.97524173,  -5.18441766, ...,   0.46696848,
          10.52483181,  75.46590805],
        [ 34.9957851 ,  40.30182428,  31.67794313, ..., -33.47318585,
         -35.55193883, -76.57536378],
        [ 37.79535635,  33.39729941,  44.96842661, ...,   6.40136675,
          -0.42794594, -75.24166264],
        ...,
        [  3.38626611,  -4.04710001,  -2.65373841, ..., -12.89115118,
         -18.27624281, -18.39318583],
        [ -7.60248563,  -8.48290156,  -8.61602311, ..., -29.26906189,
         -25.87176964, -16.69417752],
        [-14.84241454, -22.89186163, -20.17341647, ...,  20.02706049,
          -3.38539478,   8.06464068]])
```

Now, I read in the first image, which the new EigenFace Matrix has never seen before, as the test image vector. I subtract the just generated new mean face from the test image vector for normalization.

```
#Read in the first image that we removed and replaced from the database, normalize it w
ith the new test mean face.
test_vec, ____, ___, ____ = ReadAndNorm([all_path[0]])
test_norm_vec = test_vec - test_mean_face
```

Next, I project the normalized test image vector and the normalized whole dataset matrix onto the new EigenFace image space to generate projected version of them. This is done by taking the dot product between EigenFace Matrix and them:

```
test_eig_proj = np.matmul(eigenface_matrix_test, test_norm_vec)
all_eig_proj = np.matmul(eigenface_matrix_test, test_norm_face)
```

The resulting projected vector and matrix is of shape 30 x 1 and 30 x 400.

I then compute the Euclidean distance using **sklearn.metrics.pairwise.**`euclidean_distances()` and see if the model can recognize the test image as the same person. The index of the min distance should match the index of the predicted image in the image path list. Thus, I can use it to retrieve the original image path and hence the image too:

```
#Compute the euclidean_distances and find out which picture in database is cloest to th
e test image, hopefully from the 1st
test_distance = [round(float(i), 2) for i in euclidean_distances(all_eig_proj.T, test_e
ig_proj.T)]
predict_face_path = all_path_test[test_distance.index(min(test_distance))]
```

The final predicted image has the path:

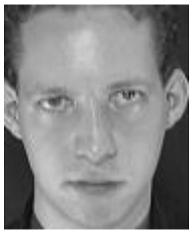
```
'.\\FACESdata\\s1\\7.pgm'
```

Compare to the original test image path:

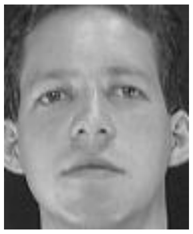
```
'.\\FACESdata\\s1\\1.pgm'
```

This is exactly what we expected to see. They are both from the folder of the first person. The model correctly recognized a never-seen test image as the same person from whose folder that we took test image out of!

To better visualize the result, the test image looks like:



And the predicted image from minimal post-projection Euclidean Distance looks like:



It's an image of the same person from a different angle.

The complete version of code can be found here:

https://github.com/yifanwu0909/PCA-and-EigenFaces/blob/master/EigenFaces.ipynb

Mathematical Background:

The theory behind this assignment is PCA (Principle Component Analysis). It aims for extracting dimensions where our data varies the most to capture as many variations in the data with as few top representative dimensions as possible. To achieve this goal, the algorithm will start finding the direction the data varies the most and each of the successive dimensions will be perpendicular to any of the previous dimensions extracted to ensure max representational capacity. Each such extracted dimension is called one Principle Component and the ceiling number of Principle Component is the same with the dimensions of the original data. For example, if we have a two-dimensional data, which can be well represented in a cartesian coordinate, the first principle component will be the direction along which all the points in the coordinate scatter the furthest away and thus have the highest variance. To capture the second principle component with second highest variance in data, we choose the second principle component perpendicular with the first one. Now all the data point can be projected onto these two new axes. You cannot fine the third principle component because the original data point only has two dimensions. See graph below:

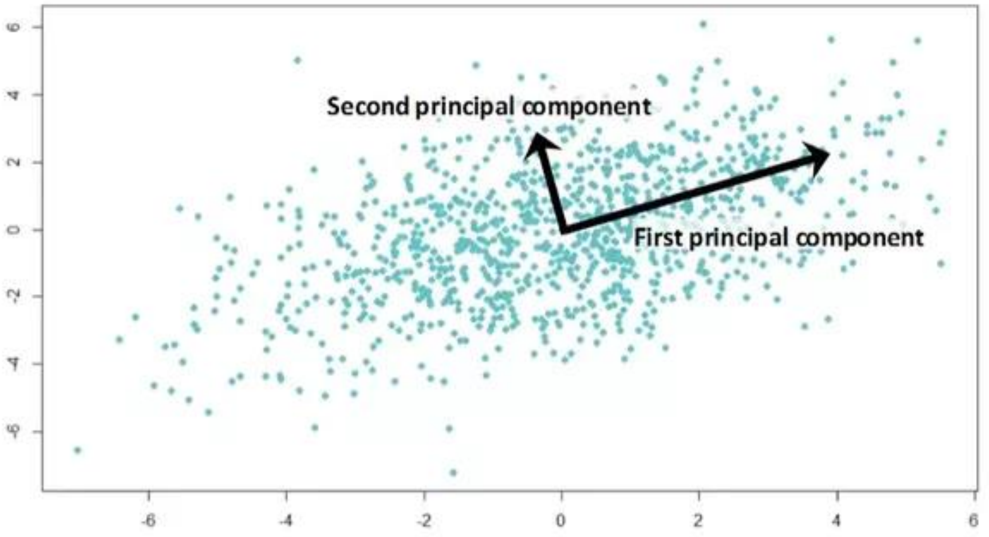


Image Source: https://medium.com/@sadatnazrul/the-dos-and-donts-of-principal-component-analysis-7c2e9dc8cc48

In the case of EigenFace detection, where all the training face is concatenated in one large matrix, a simple linear algebra can be used to extract top principle components in the images space. The principle components (later as PCs), v_i, could be derived using the following equation:

$$AA^T v_i = \lambda v_i$$

(AA^T here is called the Covariance Matrix of A)

Since the covariance matrix taken using AA^T will be too big for eigen vector/value computation, Turk and Pentland in 1991 proposed using $A^T A$ to compute a much smaller covariance matrix and later project the original image back to the eigen vectors to generate PCs for the original image matrix. We take the covariance matrix of the original face space A through $A^T A$, and call the resulting covariance matrix C.

$$C = A^T A$$

If v is a nonzero and λ is a number such that $Cv = \lambda v$. Then v is said to be an eigenvector of C with eigenvalue λ. Here the eigen vector v_i is the principle component of the matrix A^T,

$$Cv = \lambda v$$

$$A^T A v_i = \lambda v_i$$

To get the principle component of the original matrix A, we then multiple both side by A to preserve the equation:

$$AA^T Av_i = A\lambda v_i$$

$$(AA^T)(Av_i) = \lambda(Av_i)$$

So, to get the EigenFaces of the original face matrix A, we only need to take the dot product of the original face matrix A and previous derived eigen vector v_i, which is what I did in the python program.