EigenFace Implementation

—Homework3 for computer vision

Name: Wang Jun

Student ID：3170100186

Date: 2021-12-10

Course: Computer Vision

Instructor: Mingli Song

Chapter 1: The purpose and requirements of the experiment

- Assuming that each face image has only one face, and the positions of the two eyes are known (that can be manually labeled). The eye position of each image is stored in a text file with the same name as the image file but with the suffix txt under the corresponding directory. The text file is represented by a line of 4 numbers separated by spaces, corresponding to the center of the two eyes respectively. Position in the image;
- Realize two program processes (two execution files), corresponding to training and recognition respectively
- Build a face database by yourself (at least 40 people, including yourself), and the course homepage provides a face database to choose from.
- You cannot directly call some functions related to Eigenface in OpenCV. The eigenvalue and eigenvector solving functions can be called; only C/C++, Python, and other programming languages cannot be used; GUI can only use OpenCV's own HighGUI, not Use QT or others; the platform can use Win/Linux/MacOS, it is recommended that Win is preferred;
- The format of the training program is roughly: "mytrain.exe energy percentage model file name and other parameters...", use the energy percentage to determine how many eigenfaces are taken, and save the training result output to the model file. At the same time, the first 10 characteristic faces are assembled into an image, and then displayed.
- The format of the recognition program is roughly: "mytest.exe face image file name model file name other parameters...", after loading the model file, the input face image is recognized, and the recognition result is superimposed on the input person The face image is displayed, and the image most similar to the face image in the face library is displayed at the same time.

Chapter 2: Experimental content and principle

Matrix & Eigenvalues & Eigenvectors

Matrix is a form of transformation (square matrices)

This is especially useful when we are trying to use geometry (visualization) to reveal the true nature of many linear algebra concepts. Even if you draw a chessboard, it will still be very helpful, especially for us beginners, to understand some of the most basic concepts in depth, they are just like imprinted in our minds.

Essence of linear algebra

I discovered these masterpiece series (eigenvalues, eigenvectors, eigenbases, and of course eigenfaces) while trying to better understand eigenvalues.

The most important way is:

A matrix (square) can and should be regarded as a transformation, where each column represents the location of the new base. We can visualize this process by dragging [1, 0] to the first column of the transformation matrix, and [0, 1] to the second one, assuming a 2 x 2 matrix. The feature vector is a vector that will not be knocked out of the span during execution conversion described in the previous semester. Generally, for a 2 x 2 matrix, there are two directions. During the previous dragging process item, don't rotate, just zoom in to a certain extent. To a certain extent, it's the eigenvalue.

OpenCV provides us with some APIs for calculating eigenvectors and eigenvalues. Unfortunately, those direct method is a bit outdated and no longer exists in modern interfaces (especially if we use Python as the programming language to perform these CV tasks).

But as we might guess from the importance of intrinsic things, there are plenty of other resources available. And they are all optimized because people often use them.

PCA: Principal Component Analysis

The principal components of a collection of points in a real coordinate space are a sequence of p unit vectors, where the i-th vector is the direction of a line that best fits the data while being orthogonal to the first i-1 vectors. Here, a best-fitting line is defined as one that minimizes the average squared distance from the points to the line. These directions constitute an orthonormal basis in which different individual dimensions of the data are linearly uncorrelated. Principal component analysis (PCA) is the process of computing the principal components and using them to perform a change of basis on the data, sometimes using only the first few principal components and ignoring the rest.

Intuitively speaking, the principal component is the axis. If the data point is projected onto the axis, the distance from the projection to the origin will get the largest variance. Similar to the feature, we found that through some visualizations and a detailed explanation, there are a lot of examples. This is the script that touches me the most. It is indeed a masterpiece. As a technical report, it passed 2500 citations on Google Scholar

The main methods are:

Principal Components are those axes that preserve information about the original data. Take a 2-d checkerboard for example, there's a bunch of points on this plane and they roughly form a straight line. More simplicity, we assume this rough line to be y=x then intuitively, if we memorize the points as the distance to the origin, we'll lose one degree of freedom But we'll be able to roughly reconstruct the points by drawing them on the line y=x using there distance to the origin and this y=x would be the Principal Components for those bunch of points.

Mathematically, one can compute the principal components by getting the eigen vectors sorted by their corresponding eigenvalues reversely of the co-variance matrix of those data points a long sentence, right? A co-variance matrix is a convenient way to write down all the co-variance of the data points in question, where its index indicates the original data. For example, the value at [i, j] is the co-variance of data[i] and data[j].

Chapter 3: Experimental procedure and analysis

Steps to take for Eigenface-based recognizer

1. Find the eye position of a given database image: this can be done manually (annotated by humans), or, for convenience, we can also use the haar cascade recognizer. It just comes with OpenCV.

2. Aim all eyes at the given positions of our mask. In this process, we have to do some image conversion: the translation can be determined from the difference between the center of the two eyes of the mask, and the given image can be determined according to the tangent of the vector from the left eye to the right eye. Rotation can be a scale (or vice versa). Selected from the length (norm) of the previous vector

3. Equalize the histogram of the image

For gray images, we only need to perform standard equalization.

But for color images, we must first convert RGB to include grayscale (or brightness) and perform histogram equalization on this channel.

4. Calculate the average face and subtract it from all the original image data

5. Construct the covariance matrix. Let's flatten the aligned and balanced faces into batches and calculate the covariance matrix of all possible pixel indexes

6. Calculate the eigenvalue/eigenvector of the above covariance matrix. The eigenvector with the largest eigenvalue is the eigensurface.

7. In order to express human faces in the form of eigenfaces, we will take the dot product between the flat faces

An image with all the above feature vectors. The pop-up result is the version of the compressed image data relative to the characteristic face.

8. To reconstruct the face, it is as simple as matrix multiplication, summing the weighted feature vector, and adding the average face mentioned in 4. Expand it, and we will get the face.

9. Recognize human faces. Make a Euclidean distance between the weight of the face and the weight calculated in 8 in the database. The one with the smallest distance will be the most similar face in the database

Implementation

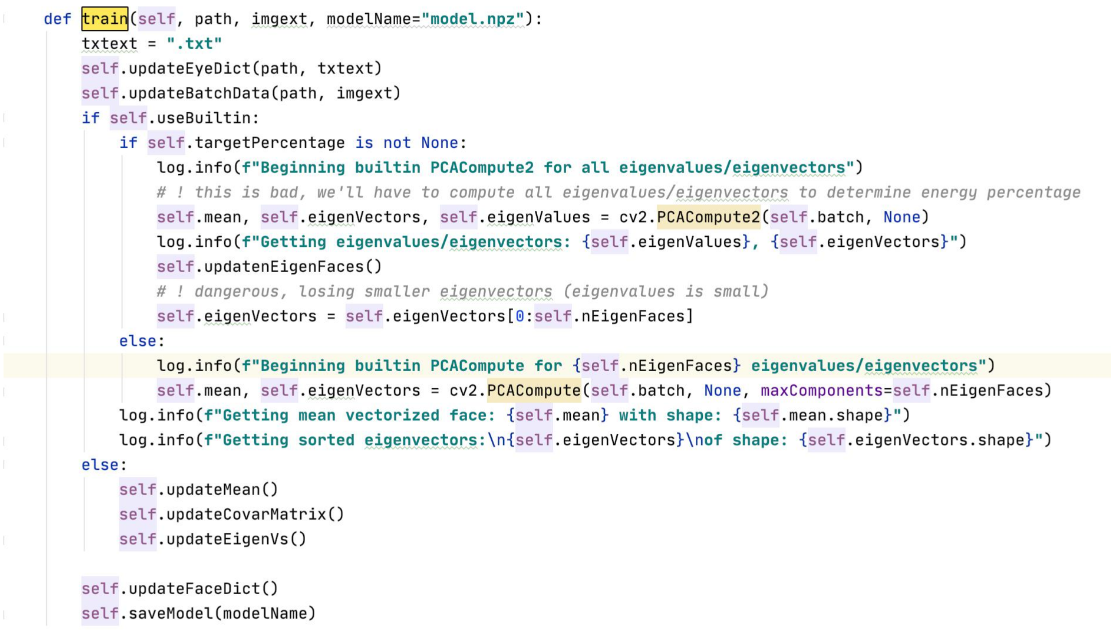
- **Main Steps:**

1. You can choose to process the image into a grayscale image or a color image (the image is still reconstructed as a color image)
2. You can choose to preset the number of characteristic faces or set only one target information Retention rate
3. You can choose the size of the mask and the position of the left and right eyes
4. You can choose to use OpenCV's PCA implementation or our

5. You can also choose to use HighGUI or just matplotlib, but this part is clear
6. Removed due to supervision. In theory, you can use any data set you want, even if the eyes/faces are not annotated or their sizes are weird.
You don’ t have to stick to the data set we provide, because we have carefully aligned
7. Our implementation does not have to be limited to a certain data set, as long as it has some recognizable eyes
8. The EigenFaceUtils class can be configured using a configuration file and the configuration can also be saved from the instantiated object
9. The configuration file does not have to have a clearly associated model. The only requirement is that the size of the mask matches and the isColor field is set correctly
So you can actually link multiple models to one profile

Training:

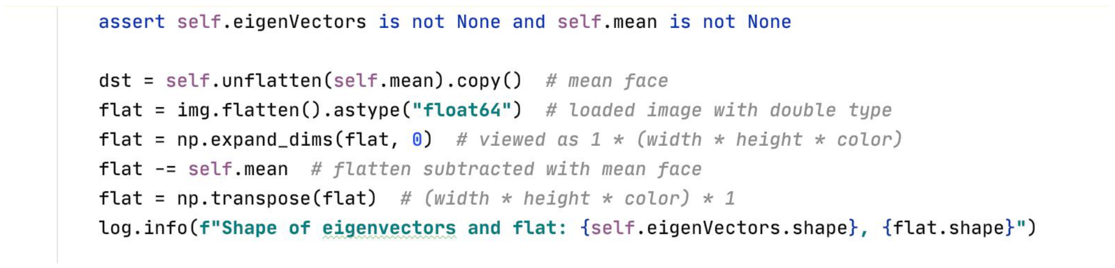
Essentially, this is the training procedure illustated in the previous section:



```
    def train(self, path, imgext, modelName="model.npz"):
        txtext = ".txt"
        self.updateEyeDict(path, txtext)
        self.updateBatchData(path, imgext)
        if self.useBuiltin:
            if self.targetPercentage is not None:
                log.info(f"Beginning builtin PCACompute2 for all eigenvalues/eigenvectors")
                # ! this is bad, we'll have to compute all eigenvalues/eigenvectors to determine energy percentage
                self.mean, self.eigenVectors, self.eigenValues = cv2.PCACompute2(self.batch, None)
                log.info(f"Getting eigenvalues/eigenvectors: {self.eigenValues}, {self.eigenVectors}")
                self.updatenEigenFaces()
                # ! dangerous, losing smaller eigenvectors (eigenvalues is small)
                self.eigenVectors = self.eigenVectors[0:self.nEigenFaces]
            else:
                log.info(f"Beginning builtin PCACompute for {self.nEigenFaces} eigenvalues/eigenvectors")
                self.mean, self.eigenVectors = cv2.PCACompute(self.batch, None, maxComponents=self.nEigenFaces)
            log.info(f"Getting mean vectorized face: {self.mean} with shape: {self.mean.shape}")
            log.info(f"Getting sorted eigenvectors:\n{self.eigenVectors}\nof shape: {self.eigenVectors.shape}")
        else:
            self.updateMean()
            self.updateCovarMatrix()
            self.updateEigenVs()

        self.updateFaceDict()
        self.saveModel(modelName)
```

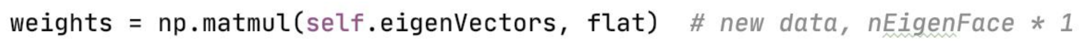
Reconstruction:

Initial:



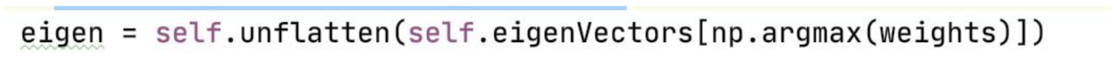
```
        assert self.eigenVectors is not None and self.mean is not None

        dst = self.unflatten(self.mean).copy()  # mean face
        flat = img.flatten().astype("float64")  # loaded image with double type
        flat = np.expand_dims(flat, 0)  # viewed as 1 * (width * height * color)
        flat -= self.mean  # flatten subtracted with mean face
        flat = np.transpose(flat)  # (width * height * color) * 1
        log.info(f"Shape of eigenvectors and flat: {self.eigenVectors.shape}, {flat.shape}")
```

nEigenFace *(width * height * color) matmal (width * height * color) * 1:



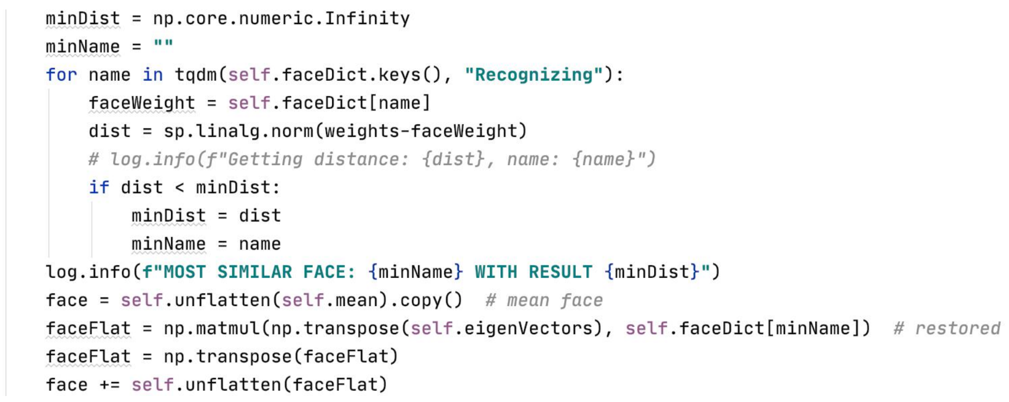
```
weights = np.matmul(self.eigenVectors, flat)  # new data, nEigenFace * 1
```

getting the most similar eigenface:



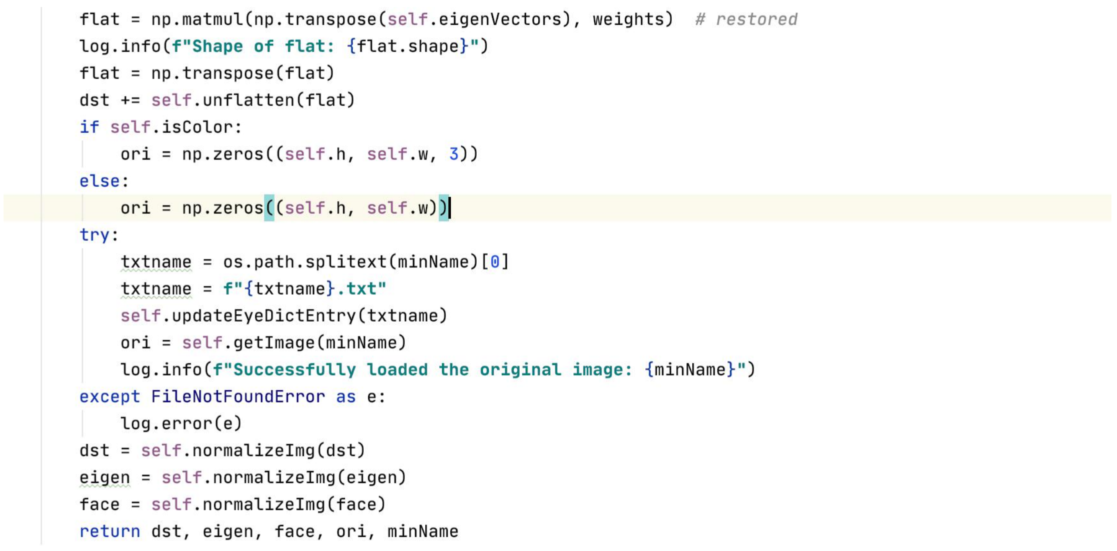
```
eigen = self.unflatten(self.eigenVectors[np.argmax(weights)])
```

getting the most similar face in the database:



```
minDist = np.core.numeric.Infinity
minName = ""
for name in tqdm(self.faceDict.keys(), "Recognizing"):
    faceWeight = self.faceDict[name]
    dist = sp.linalg.norm(weights-faceWeight)
    # log.info(f"Getting distance: {dist}, name: {name}")
    if dist < minDist:
        minDist = dist
        minName = name
log.info(f"MOST SIMILAR FACE: {minName} WITH RESULT {minDist}")
face = self.unflatten(self.mean).copy()  # mean face
faceFlat = np.matmul(np.transpose(self.eigenVectors), self.faceDict[minName])  # restored
faceFlat = np.transpose(faceFlat)
face += self.unflatten(faceFlat)
```

Eigenvectors of real symmetric matrices are orthogonal.
data has been lost because nEigenFaces is much smaller than the image dimension span which is width * height * color, but because we're using PCA (principal components), most of the information will still be retained.

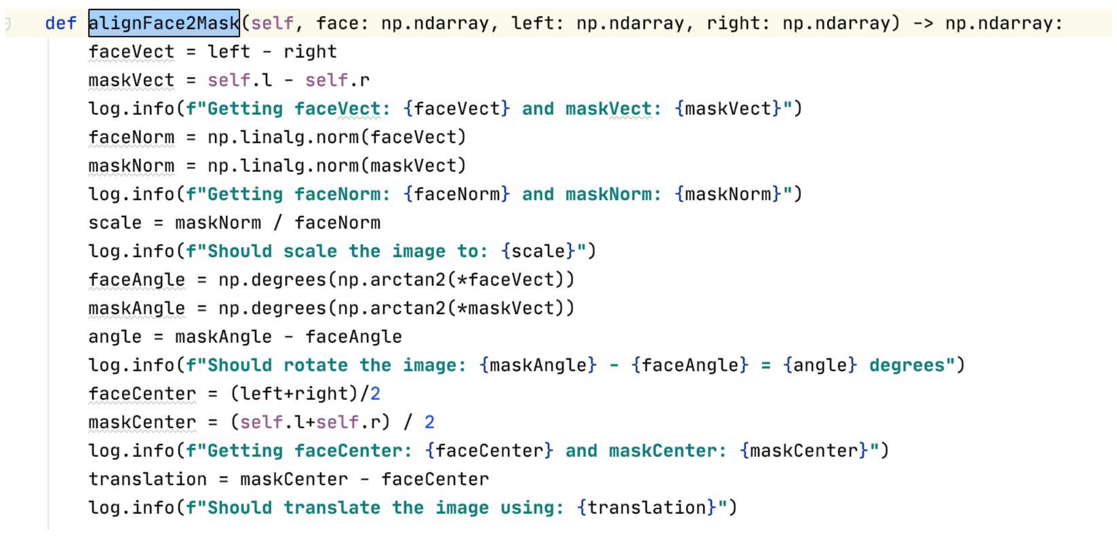


```
flat = np.matmul(np.transpose(self.eigenVectors), weights)  # restored
log.info(f"Shape of flat: {flat.shape}")
flat = np.transpose(flat)
dst += self.unflatten(flat)
if self.isColor:
    ori = np.zeros((self.h, self.w, 3))
else:
    ori = np.zeros((self.h, self.w))
try:
    txtname = os.path.splitext(minName)[0]
    txtname = f"{txtname}.txt"
    self.updateEyeDictEntry(txtname)
    ori = self.getImage(minName)
    log.info(f"Successfully loaded the original image: {minName}")
except FileNotFoundError as e:
    log.error(e)
dst = self.normalizeImg(dst)
eigen = self.normalizeImg(eigen)
face = self.normalizeImg(face)
return dst, eigen, face, ori, minName
```

Modules:

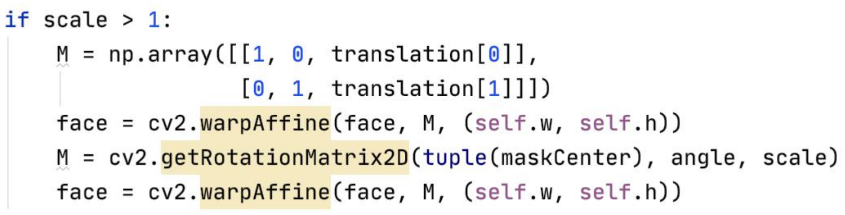
Face Alignment:
Align all the eyes to the given position of our mask.
In this process, we'll have to do some image transform:

- Translation can be determined from the difference of the mask's two eye center and the given image
- Rotation can be determined from the tangent of the vector from the left eye to the right one (or the other way around)
- Scale can be determined from length (norm) of the vector of the previous term



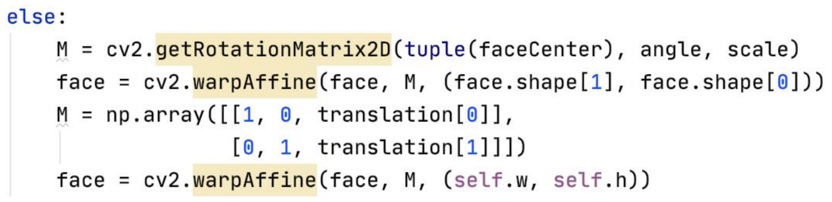
```python
def alignFace2Mask(self, face: np.ndarray, left: np.ndarray, right: np.ndarray) -> np.ndarray:
    faceVect = left - right
    maskVect = self.l - self.r
    log.info(f"Getting faceVect: {faceVect} and maskVect: {maskVect}")
    faceNorm = np.linalg.norm(faceVect)
    maskNorm = np.linalg.norm(maskVect)
    log.info(f"Getting faceNorm: {faceNorm} and maskNorm: {maskNorm}")
    scale = maskNorm / faceNorm
    log.info(f"Should scale the image to: {scale}")
    faceAngle = np.degrees(np.arctan2(*faceVect))
    maskAngle = np.degrees(np.arctan2(*maskVect))
    angle = maskAngle - faceAngle
    log.info(f"Should rotate the image: {maskAngle} - {faceAngle} = {angle} degrees")
    faceCenter = (left+right)/2
    maskCenter = (self.l+self.r) / 2
    log.info(f"Getting faceCenter: {faceCenter} and maskCenter: {maskCenter}")
    translation = maskCenter - faceCenter
    log.info(f"Should translate the image using: {translation}")
```

if we're scaling up, we should first translate then do the scaling, else the image will get cropped and we'd all want to use the larger destination width*height:



```python
if scale > 1:
    M = np.array([[1, 0, translation[0]],
                  [0, 1, translation[1]]])
    face = cv2.warpAffine(face, M, (self.w, self.h))
    M = cv2.getRotationMatrix2D(tuple(maskCenter), angle, scale)
    face = cv2.warpAffine(face, M, (self.w, self.h))
```

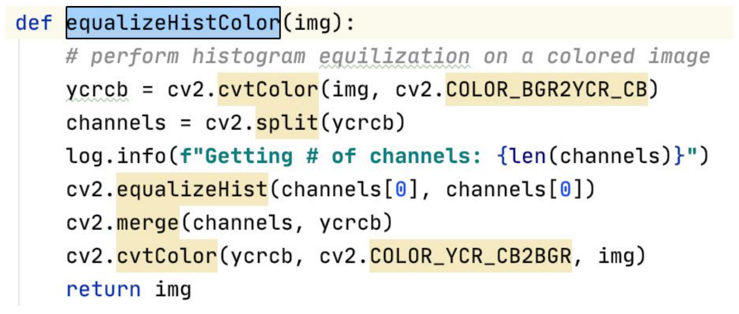
# if we're scaling down, we should first rotate and scale then translate, else the image will get cropped and we'd all want to use the larger destination width*height



```python
else:
    M = cv2.getRotationMatrix2D(tuple(faceCenter), angle, scale)
    face = cv2.warpAffine(face, M, (face.shape[1], face.shape[0]))
    M = np.array([[1, 0, translation[0]],
                  [0, 1, translation[1]]])
    face = cv2.warpAffine(face, M, (self.w, self.h))
```

Histogram Equlization:

Do a histogram equilization on the image
For gray ones, we'd only have to perform a standard equalization, but for the colored image we'll have to firstly convert RGB to a color space including grayscale (or luminance) and perform the histogram equalization on that channel.



```python
def equalizeHistColor(img):
    # perform histogram equilization on a colored image
    ycrcb = cv2.cvtColor(img, cv2.COLOR_BGR2YCR_CB)
    channels = cv2.split(ycrcb)
    log.info(f"Getting # of channels: {len(channels)}")
    cv2.equalizeHist(channels[0], channels[0])
    cv2.merge(channels, ycrcb)
    cv2.cvtColor(ycrcb, cv2.COLOR_YCR_CB2BGR, img)
    return img
```

Automate the above procedure:

```python
def getImage(self, name, manual_check=False) -> np.ndarray:
    # the load the image accordingly
    if self.isColor:
        img = cv2.imread(name, cv2.IMREAD_COLOR)
    else:
        img = cv2.imread(name, cv2.IMREAD_GRAYSCALE)

    # try getting eye position
    basename = os.path.basename(name)  # get file name
    basename = os.path.splitext(basename)[0]  # without ext
    eyes = self.getEyes(basename, img)
    log.info(f"Getting eyes: {eyes}")
    if not len(eyes) == 2:
        log.warning(f"Cannot get two eyes from this image: {name}, {len(eyes)} eyes")
        del self.eyeDict[basename]
        raise EigenFaceException("Bad image")

    # align according to eye position
    dst = self.alignFace2Mask(img, eyes[0], eyes[1])

    # hist equalization
    if self.isColor:
        dst = self.equalizeHistColor(dst)
    else:
        dst = cv2.equalizeHist(dst)

    # should we check every image before/after loading?
    if manual_check:
        cv2.imshow(name, dst)
        cv2.waitKey()
        cv2.destroyWindow(name)
    return dst
```

Get Mean Face:

Compute the mean face and subtract it from all the original image data

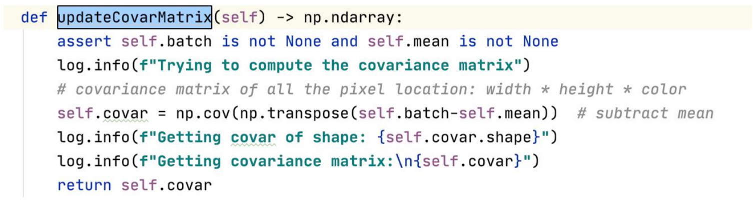
```python
def updateMean(self):
    assert self.batch is not None
    # get the mean values of all the vectorized faces
    self.mean = np.reshape(np.mean(self.batch, 0), (1, -1))
    log.info(f"Getting mean vectorized face: {self.mean} with shape: {self.mean.shape}")
    return self.mean
```

Get Covariance Matrix (if needed) :

Construct the covariance matrix. Let's flatten the aligned, equalized face to batch and compute covariance matrix of all the possible pixel indices.

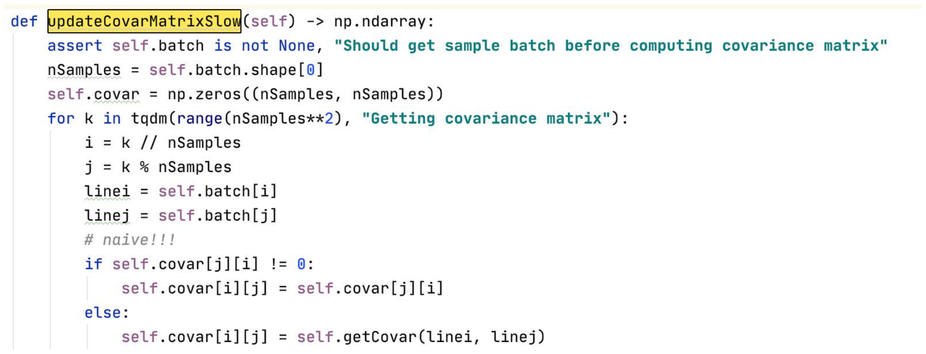
This procedure would not be necessary if we want to use the OpenCV's implementation of

PCACompute:

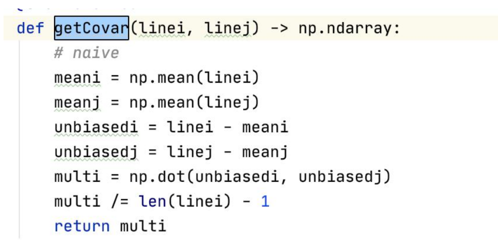


```python
def updateCovarMatrix(self) -> np.ndarray:
    assert self.batch is not None and self.mean is not None
    log.info(f"Trying to compute the covariance matrix")
    # covariance matrix of all the pixel location: width * height * color
    self.covar = np.cov(np.transpose(self.batch-self.mean))  # subtract mean
    log.info(f"Getting covar of shape: {self.covar.shape}")
    log.info(f"Getting covariance matrix:\n{self.covar}")
    return self.covar
```

A custom slow implementation:



```python
def updateCovarMatrixSlow(self) -> np.ndarray:
    assert self.batch is not None, "Should get sample batch before computing covariance matrix"
    nSamples = self.batch.shape[0]
    self.covar = np.zeros((nSamples, nSamples))
    for k in tqdm(range(nSamples**2), "Getting covariance matrix"):
        i = k // nSamples
        j = k % nSamples
        linei = self.batch[i]
        linej = self.batch[j]
        # naive!!!
        if self.covar[j][i] != 0:
            self.covar[i][j] = self.covar[j][i]
        else:
            self.covar[i][j] = self.getCovar(linei, linej)
```

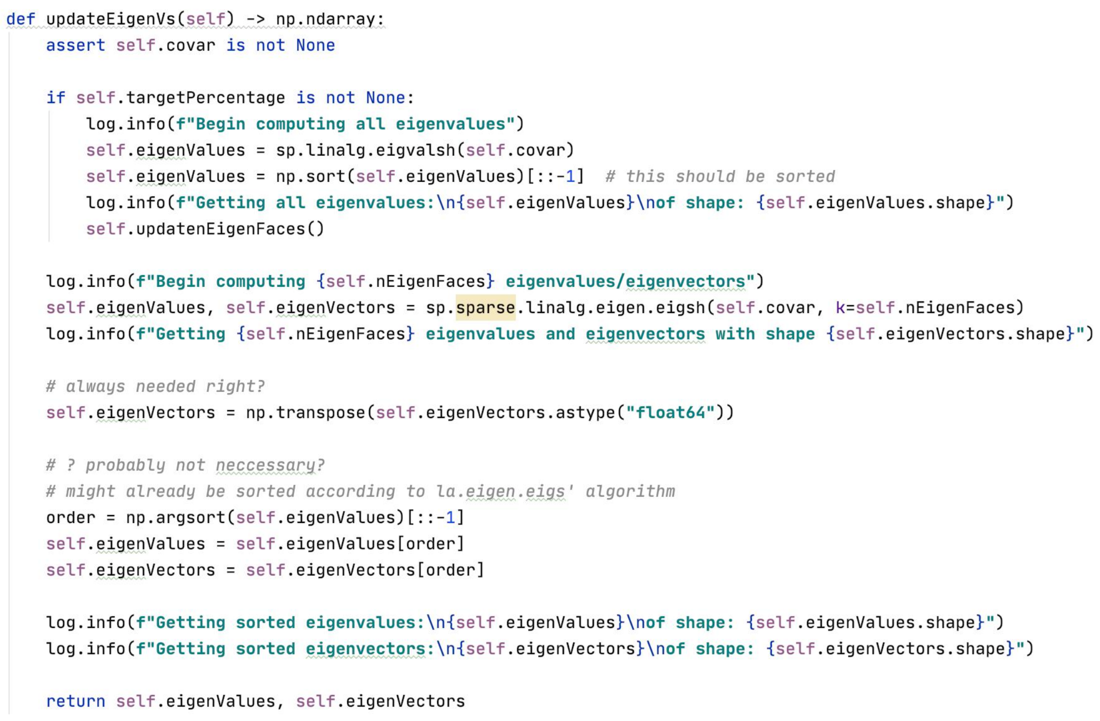


```python
def getCovar(linei, linej) -> np.ndarray:
    # naive
    meani = np.mean(linei)
    meanj = np.mean(linej)
    unbiasedi = linei - meani
    unbiasedj = linej - meanj
    multi = np.dot(unbiasedi, unbiasedj)
    multi /= len(linei) - 1
    return multi
```

Compute Eigenvalues/Eigenvectors:

Compute the eigenvalues/eigenvectors of the above mentioned covariance matrix.

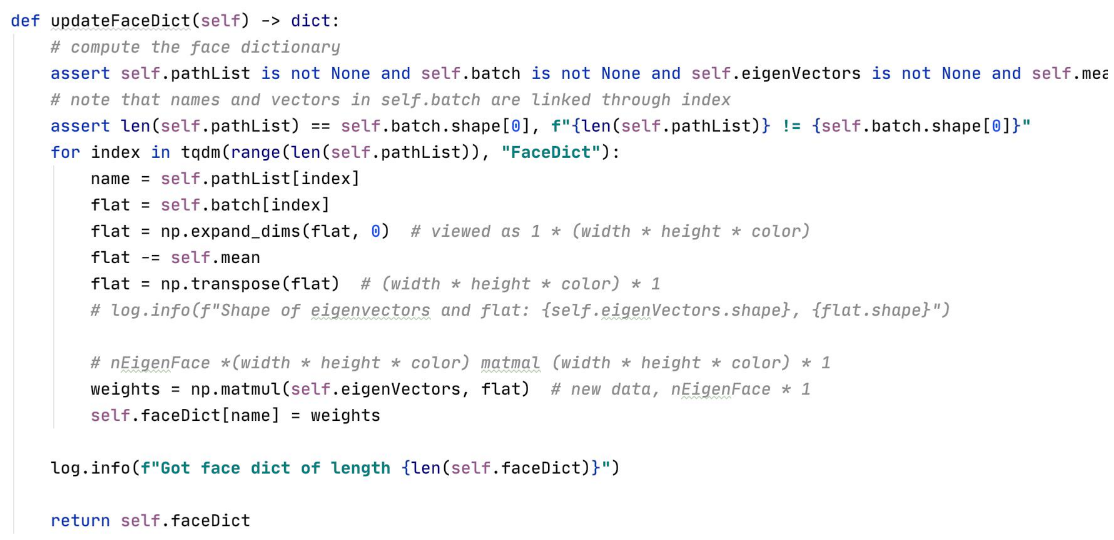
The eigenvectors with the largest eigenvalues are eigenfaces



```python
def updateEigenVs(self) -> np.ndarray:
    assert self.covar is not None

    if self.targetPercentage is not None:
        log.info(f"Begin computing all eigenvalues")
        self.eigenValues = sp.linalg.eigvalsh(self.covar)
        self.eigenValues = np.sort(self.eigenValues)[::-1]  # this should be sorted
        log.info(f"Getting all eigenvalues:\n{self.eigenValues}\nof shape: {self.eigenValues.shape}")
        self.updatenEigenFaces()

    log.info(f"Begin computing {self.nEigenFaces} eigenvalues/eigenvectors")
    self.eigenValues, self.eigenVectors = sp.sparse.linalg.eigen.eigsh(self.covar, k=self.nEigenFaces)
    log.info(f"Getting {self.nEigenFaces} eigenvalues and eigenvectors with shape {self.eigenVectors.shape}")

    # always needed right?
    self.eigenVectors = np.transpose(self.eigenVectors.astype("float64"))

    # ? probably not neccessary?
    # might already be sorted according to la.eigen.eigs' algorithm
    order = np.argsort(self.eigenValues)[::-1]
    self.eigenValues = self.eigenValues[order]
    self.eigenVectors = self.eigenVectors[order]

    log.info(f"Getting sorted eigenvalues:\n{self.eigenValues}\nof shape: {self.eigenValues.shape}")
    log.info(f"Getting sorted eigenvectors:\n{self.eigenVectors}\nof shape: {self.eigenVectors.shape}")

    return self.eigenValues, self.eigenVectors
```

Face Dict:

And of course we'd like to express the database with our new eigenface basis



```python
def updateFaceDict(self) -> dict:
    # compute the face dictionary
    assert self.pathList is not None and self.batch is not None and self.eigenVectors is not None and self.mea
    # note that names and vectors in self.batch are linked through index
    assert len(self.pathList) == self.batch.shape[0], f"{len(self.pathList)} != {self.batch.shape[0]}"
    for index in tqdm(range(len(self.pathList)), "FaceDict"):
        name = self.pathList[index]
        flat = self.batch[index]
        flat = np.expand_dims(flat, 0)  # viewed as 1 * (width * height * color)
        flat -= self.mean
        flat = np.transpose(flat)  # (width * height * color) * 1
        # log.info(f"Shape of eigenvectors and flat: {self.eigenVectors.shape}, {flat.shape}")

        # nEigenFace *(width * height * color) matmal (width * height * color) * 1
        weights = np.matmul(self.eigenVectors, flat)  # new data, nEigenFace * 1
        self.faceDict[name] = weights

    log.info(f"Got face dict of length {len(self.faceDict)}")

    return self.faceDict
```

Chapter 4: Experimental environment and operation method

Development environment

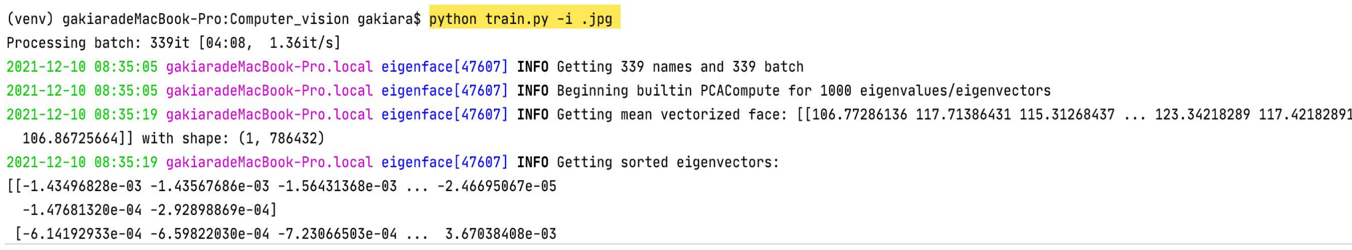
- macOS 10.14.6
- python 3.9
- opencv-python 4.5.1.48

Operation mode

Training:

python train.py -i .png

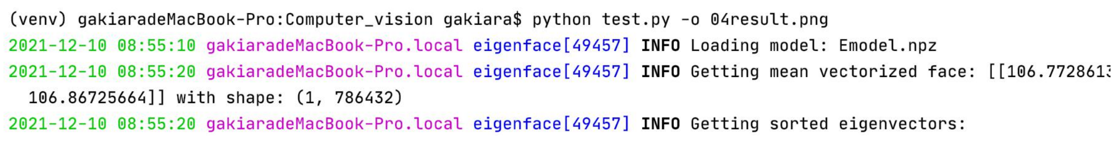
python train.py -i .jpg



```
(venv) gakiaradeMacBook-Pro:Computer_vision gakiara$ python train.py -i .jpg
Processing batch: 339it [04:08,  1.36it/s]
2021-12-10 08:35:05 gakiaradeMacBook-Pro.local eigenface[47607] INFO Getting 339 names and 339 batch
2021-12-10 08:35:05 gakiaradeMacBook-Pro.local eigenface[47607] INFO Beginning builtin PCACompute for 1000 eigenvalues/eigenvectors
2021-12-10 08:35:19 gakiaradeMacBook-Pro.local eigenface[47607] INFO Getting mean vectorized face: [[106.77286136 117.71386431 115.31268437 ... 123.34218289 117.42182891
  106.86725664]] with shape: (1, 786432)
2021-12-10 08:35:19 gakiaradeMacBook-Pro.local eigenface[47607] INFO Getting sorted eigenvectors:
[[-1.43496828e-03 -1.43567686e-03 -1.56431368e-03 ... -2.46695067e-05
  -1.47681320e-04 -2.92898869e-04]
 [-6.14192933e-04 -6.59822030e-04 -7.23066503e-04 ...  3.67038408e-03
```

Testing:

python test.py -o result.png

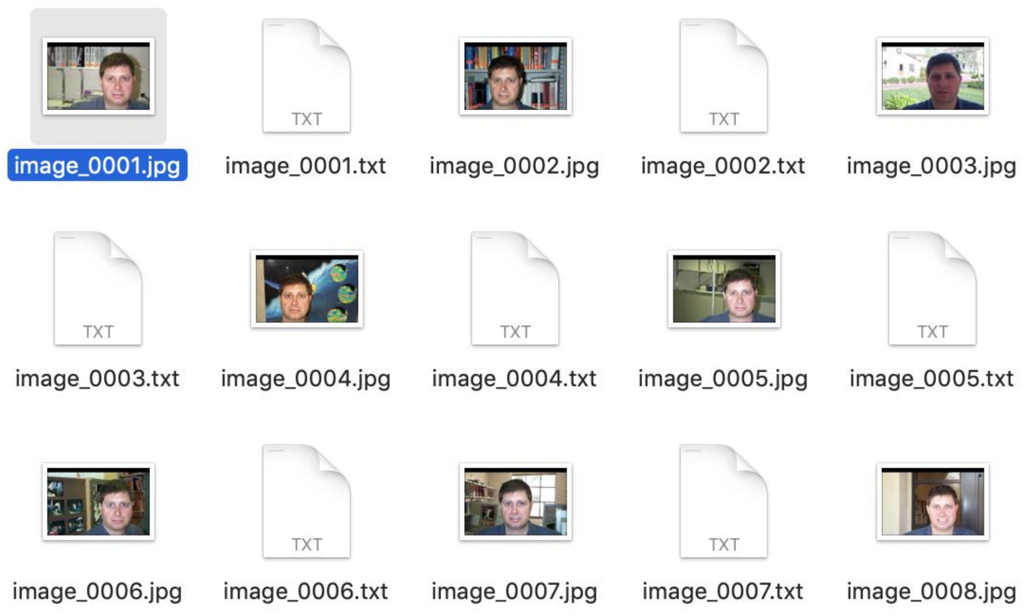


```
(venv) gakiaradeMacBook-Pro:Computer_vision gakiara$ python test.py -o 04result.png
2021-12-10 08:55:10 gakiaradeMacBook-Pro.local eigenface[49457] INFO Loading model: Emodel.npz
2021-12-10 08:55:20 gakiaradeMacBook-Pro.local eigenface[49457] INFO Getting mean vectorized face: [[106.772861
  106.86725664]] with shape: (1, 786432)
2021-12-10 08:55:20 gakiaradeMacBook-Pro.local eigenface[49457] INFO Getting sorted eigenvectors:
```

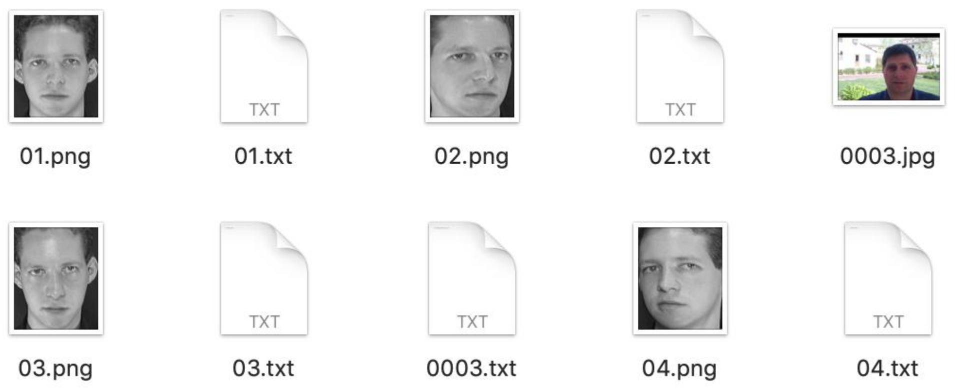
Chapter 5: Experimental results

Input

Train_DataSet: 450 images & txt

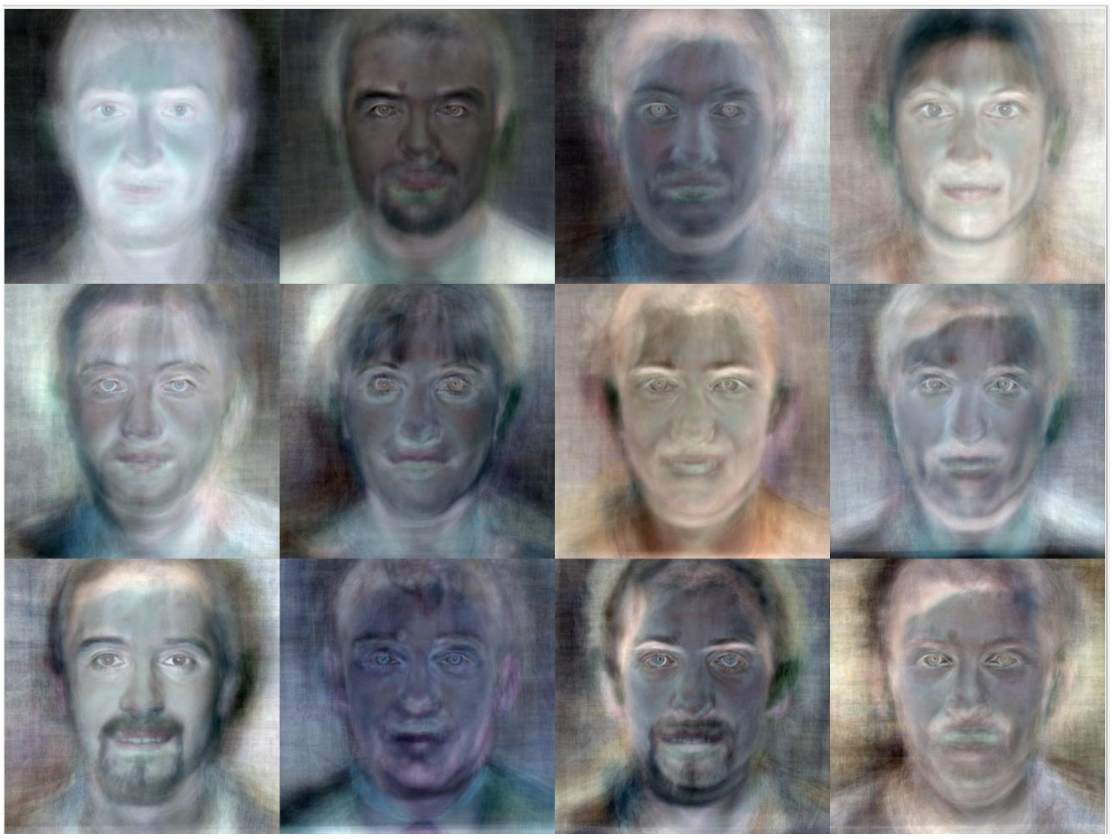


Test_DataSet:



Output

eigenfaces:



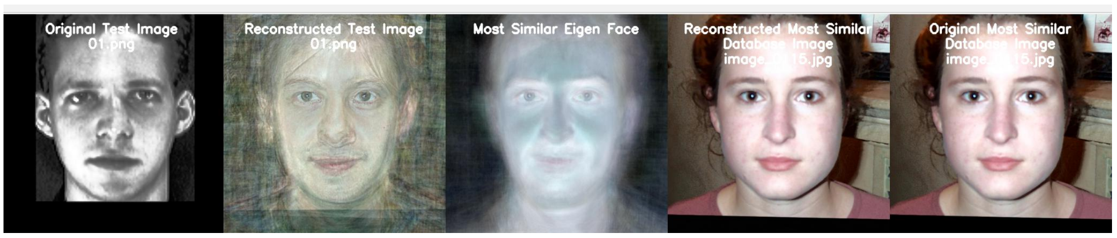
The mean eigenface:

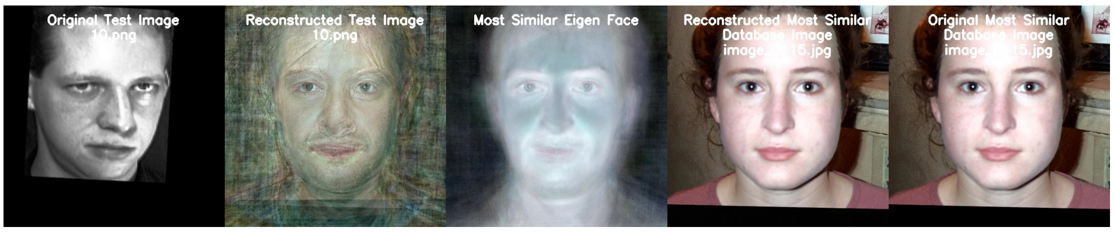
Testing:

1. Load the model, including all eigenvectors(eigenfaces), the mean face vector and all compressed image data
2. Load the test image and go through the typical preprocessing procedure of alignment and histogram equilization
3. Do a matrix multiplication between the eigenvectors and the loaded (processed) test image to retrieve the weight (compressed version of the test image)
4. Do a Euclidean Distance comparison between all the images in the database (compressed using eigenface) to find the most similar image in the database (with smallest Euclidean Distance)
5. Reconstruct the loaded test image using eigenvectors:
6. Reconstruct the most similar image in the database using the same method decribed above
7. Render those image to a canvas and label them, providing with necessary name attributes for you to view them
8. The rendered image are ordered like:

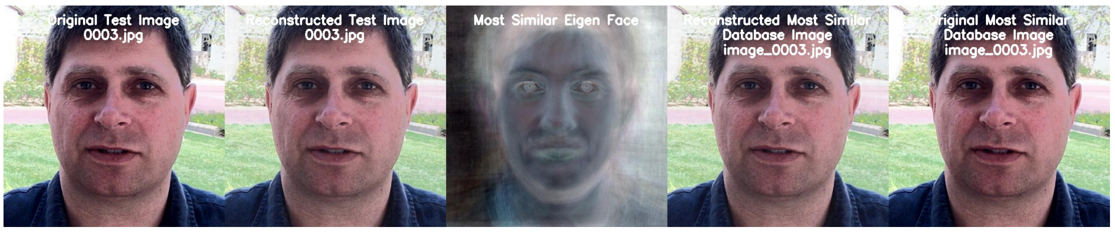
Original::Reconstructed Original::Most Weighted Eigenface::Reconstructed Most Similar Database Image::Original Most Similar Database Images

Testing with an unseen image :





Testing with an seen image:



Chapter 6: Thoughts

OpenCV in python and C++

I use mac-os to finish my work, and find the visual studio in mac is hard to use in C++, so I try to use python instead. And surprisingly, python is much easier than C++ that not only in its readable, but also it did not require you to learn about a new class Mat to be able to operate on images. In a nutshell, for man using a mac os, python would be a better choice, and the deployment of environment is easier.

Training & Testing

The data used for training is so essential to decide whether the model would work and how well it would behave. So the choosing and marking of the data is really important. Whatever training we are doing, it is the first thing to deal with the data.

Since the forms are different, such as ".jpg", ".png". We need to provide the option to choose the from. At first, I did not realize it. So the training takes lots of time.