EigenFace Implementation

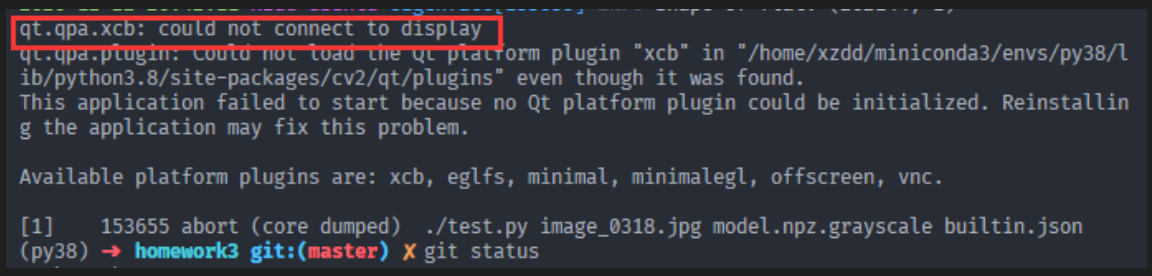
Basic Info

- Student ID: 3180105504
- Student Name: Xu Zhen
- Instructor Name: Song Mingli
- Course Name: Computer Vision
- Homework Name: `EigenFace Implementation`
- Basic Requirements:
  - Implement an EigenFace face recognition trainer/tester pair
  - Construct a face database of at least 40 persons (including the author)
  - `Don't` use OpenCV's API for eigenfaces
  - APIs relating to eigenvalues/eigenvectors `can` be used
  - No Qt for GUI, only HighGUI that comes with OpenCV can be used

    I really don't know why this requirements should exist

    OpenCV uses Qt as a backend to display stuff, at least it does so with modern Python interfaces

    This is a screenshot when I accidentally discovered this trying to open a window with an SSH connection to my server



Experiment Principles

Matrix & Eigenvalues & Eigenvectors

`Matrix is a form of transformation (square matrices)`

This is especially useful when we're trying to unveal the true nature of a lot of linear algebra concepts with the help of geometry (visualization).

Of course, one can always write abstract symbols and prove one's way throughout the linear algebra sea without having to draw even a single checkerboard, it's still gonna be extremely helpful, especially for us beginners, to understand some of the most basic concepts so deeply that, they're like imprinted into our heads.

Hereby I highly recommend the series brought to you by *3Blue1Brown*, a famous YouTuber, and a mathematician. Link below

Essence of linear algebra

I found these masterpiece series when trying to understand eigen stuff better (eigenvalues, eigenvectors, eigenbasis and of course, eigenfaces)

The most important takeways are:

- Matrices (square) can and should be viewed as a transform where each columns represents the new basis's location

  We can visualize this process by **dragging** [1, 0] to the first column of the transformation matrix, and [0, 1] to the second, assuming a 2 by 2 matrix
- Eigenvectors are vectors that doesn't get knocked off their span when performing the transfromation decribed in the previous term

  Typically, for a 2 by 2 matrix, there's two directions, which, during the **dragging** in the previous item, don't rotate around, but just scales up to a certain degree

  And that **certain degree**, are the eigenvalues

OpenCV gives us some API for computing eigenvectors and eigenvalues. Unfortunately, those direct methods are a bit outdated and don't exist anymore in modern interfaces (especially so if we're using Python as the programming language for performing these CV tasks).

But as we can probably guess from the importance of eigen stuff, there's tons of other resources for us to call. And they all got optimized the heck out of them because people use them sooooo often.

PCA: Principal Component Analysis

Intuitively, principal components are the axes, onto which if data points are projected, the projections' distances to the origin get the largest variance

Similar to eigen stuff, we find it much easier to understand PCA with a few visualization and a detailed illustration with a lot of examples.

This is the script that inspired me the most. Masterpiece indeed. As a techincal report, it got over 2500 citations on Google Scholar

A tutorial on Principal Components Analysis

The main takeways are:

- Principal Components are those axes that preserve information about the original data the most

  Take a 2-d checkerboard for example, there's a bunch of points on this plane and they roughly form a straight line

  For 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 eigenvectors sorted by their corresponding eigenvalues reversely of the covariance matrix of those data points

  A long sentence, right?

  A covariance matrix is a convenient way to write down all the covariance of the data points in question, where its index indicates the orginal data. For example, the value at `[i, j]` is the covariance of `data[i]` and `data[j]`

Core APIs

OpenCV API

```
def PCACompute(data, mean, eigenvectors= ... , maxComponents= ... ) -> typing.Any:
"""
PCACompute(data, mean[, eigenvectors[, maxComponents]]) -> mean, eigenvectors
.   wrap PCA::operator()


PCACompute(data, mean, retainedVariance[, eigenvectors]) -> mean, eigenvectors
.   wrap PCA::operator()
"""
    ...
```

In our implementation, we provided the user with the option of using `PCACompute` of OpenCV to compute the principal components efficiently.

Note that if we take the most basic approach, we'd have to construct a giant covariance matrix and compute the eigenvectors/eigenvalues of it.

For example, if we're to compute the covariance matrix of some 512 by 512 color image, meaning the covariance matrix would be 3*2^18: 786432 by 786432

Consider 64-bit double for the entries, we'd be needing 2^36 * 9 * 2^3 Bytes -> 4.5 TiB memory to just store the matrix.

And this would be totally unacceptable.

So we advise you to use the OpenCV's implementation for PCA, as long as you provide us with the nEigenFaces you want instead of trying to achieve some sort of **energy percentage**. This would be another issue will discuss later.

Well, but the instructions advise us to use our custom implementation of PCA, so we still implmented that version. You should use that version by tweaking the configuration file for our program.

Don't try computing covariance matrix for a 512*512 color image set, or we'll be asking you for 4.5TiB memory.

Instead, use a smaller image for that kind of computation, which can be set through the configuration file.

Like 128*128 gray image, 2^17 Bytes are pretty reasonable for matrix generation. And a small matrix like that CAN STILL REQUIRE TONS OF TIME TO COMPUTE EIGEN STUFF.

Scipy API

```
@array_function_dispatch(_cov_dispatcher)
def cov(m, y=None, rowvar=True, bias=False, ddof=None, fweights=None,
        aweights=None):
```

As mentioned above, to implement our own version of PCA computation, we'd first construct the covariance matrix for the image data using `numpy.cov`.

We're aware that there's covariance matrix API in OpenCV but the `numpy` one just comes in real handy

Then, with the covariance matrix, we're gonna have to compute its eigenvalues/eigenvectors. Here we used:

```
def eigsh(A, k=6, M=None, sigma=None, which='LM', v0=None,
          ncv=None, maxiter=None, tol=0, return_eigenvectors=True,
          Minv=None, OPinv=None, mode='normal'):
    """
    Find k eigenvalues and eigenvectors of the real symmetric square matrix
    or complex hermitian matrix A.
    """
```

to compute the eigenvalues and eigenvectors with more efficiency on a real symmetrix square matrix.

Full path of the API is

`scipy.sparse.linalg.eigen.eigsh`

If you're wondering why we're not using the eigenvalue/eigenvector interface of OpenCV, they don't come provided in the new interface `cv2`

Others

If we're to retain certain amount of energy when selecting the principal componenets from all eigenvectors, we'll have to firstly compute all the eigenvalues and sum them up. Eventually we'll have to decide how many eigenvalues' sum (descending order) can add up to take the right proportion we want.

Yeah, we do know **the trace of a matrix** is also the sum of all eigenvalues it has. But without the actual eigenvalues, we're not able to determine **how many** eigenvalues we need (when sorted) to take up a fixed proportion of the sum.

So in the previous two sets of APIs, we gave user the priviledge to compute only a fixed number of eigenvalues/eigenvectors, **which is hugely efficient compared to computing all eigenvalues for some matrix.**

```
def PCACompute2(data, mean, eigenvectors= ... , eigenvalues= ... , maxComponents= ... ) -> typing.Any:
"""
PCACompute2(data, mean[, eigenvectors[, eigenvalues[, maxComponents]]]) -> mean, eigenvectors, eigenvalues
.   wrap PCA::operator() and add eigenvalues output parameter



PCACompute2(data, mean, retainedVariance[, eigenvectors[, eigenvalues]]) -> mean, eigenvectors, eigenvalues
.   wrap PCA::operator() and add eigenvalues output parameter
"""
    ...
```

For OpenCV's implementation, we should use PCACompute2 to compute the eigenvalues/eigenvectors effieciently. By not providing the `maxComponents` parameter, we're to get all the eigenvalues/eigenvectors.

Note that `PCACompute`, the previous interface, doesn't compute eigenvalues

Other APIs

We used `cv2.warpAffine` to transform the input image (to match the eye with our mask)

```
def warpAffine(src, M, dsize, dst= ... , flags= ... , borderMode= ... , borderValue= ... ) -> typing.Any:
"""
warpAffine(src, M, dsize[, dst[, flags[, borderMode[, borderValue]]]]) -> dst
.   @brief Applies an affine transformation to an image.
```

```
.
.   The function warpAffine transforms the source image using the specified
matrix:
.
.   \f[\texttt{dst} (x,y) =  \texttt{src} ( \texttt{M} _{11} x +  \texttt{M}
_{12} y +  \texttt{M} _{13}, \texttt{M} _{21} x +  \texttt{M} _{22} y +
\texttt{M} _{23})\f]
.
.   when the flag #WARP_INVERSE_MAP is set. Otherwise, the transformation is
first inverted
.   with #invertAffineTransform and then put in the formula above instead of M.
The function cannot
.   operate in-place.
.
.   @param src input image.
.   @param dst output image that has the size dsize and the same type as src .
.   @param M \f$2\times 3\f$ transformation matrix.
.   @param dsize size of the output image.
.   @param flags combination of interpolation methods (see #InterpolationFlags)
and the optional
.   flag #WARP_INVERSE_MAP that means that M is the inverse transformation (
.   \f$\texttt{dst}\rightarrow\texttt{src}\f$ ).
.   @param borderMode pixel extrapolation method (see #BorderTypes); when
.   borderMode=#BORDER_TRANSPARENT, it means that the pixels in the destination
image corresponding to
.   the "outliers" in the source image are not modified by the function.
.   @param borderValue value used in case of a constant border; by default, it
is 0.
.
.   @sa  warpPerspective, resize, remap, getRectSubPix, transform
"""
    ...
```

And along the way we also used `cv2.getRotationMatrix2D`

```
def getRotationMatrix2D(center, angle, scale) → typing.Any:
"""
getRotationMatrix2D(center, angle, scale) → retval
.   @brief Calculates an affine matrix of 2D rotation.
.
.   The function calculates the following matrix:
.
.   \f[\begin{bmatrix} \alpha &  \beta & (1- \alpha )  \cdot \texttt{center.x}
-  \beta \cdot \texttt{center.y} \\ - \beta &  \alpha &  \beta \cdot
\texttt{center.x} + (1- \alpha )  \cdot \texttt{center.y} \end{bmatrix}\f]
.
.   where
.
.   \f[\begin{array}{l} \alpha =  \texttt{scale} \cdot \cos \texttt{angle} , \\
\beta =  \texttt{scale}
\cdot \sin \texttt{angle} \end{array}\f]
.
```

```
15    .   The transformation maps the rotation center to itself. If this is not the
      target, adjust the shift.
16    .
17    .   @param center Center of the rotation in the source image.
18    .   @param angle Rotation angle in degrees. Positive values mean counter-
      clockwise rotation (the
19    .   coordinate origin is assumed to be the top-left corner).
20    .   @param scale Isotropic scale factor.
21    .
22    .   @sa  getAffineTransform, warpAffine, transform
23    """
24        ...
```

We also used `cv2.equalizeHist` to equalize the histogram for grayscale image:

```
1     def equalizeHist(src, dst= ... ) → typing.Any:
2     """
3     equalizeHist(src[, dst]) → dst
4     .   @brief Equalizes the histogram of a grayscale image.
5     .
6     .   The function equalizes the histogram of the input image using the following
      algorithm:
7     .
8     .   - Calculate the histogram \f$H\f$ for src .
9     .   - Normalize the histogram so that the sum of histogram bins is 255.
10    .   - Compute the integral of the histogram:
11    .   \f[H'_i =  \sum _{0  \le j < i} H(j)\f]
12    .   - Transform the image using \f$H'\f$ as a look-up table: \f$\texttt{dst}
      (x,y) = H'(\texttt{src}(x,y))\f$
13    .
14    .   The algorithm normalizes the brightness and increases the contrast of the
      image.
15    .
16    .   @param src Source 8-bit single channel image.
17    .   @param dst Destination image of the same size and type as src
18    """
19        ...
```

For color images, we shouldn't just do the histogram equalization on all RGB channels, instead we converted the image to `YCbCr` color space, then do a `histEqulization` on the Y component

> YCbCr, Y′CbCr, or Y Pb/Cb Pr/Cr, also written as YCBCR or Y′CBCR, is a family of color spaces used as a part of the color image pipeline in video and digital photography systems. Y′ is the luma component and CB and CR are the blue-difference and red-difference chroma components. Y′ (with prime) is distinguished from Y, which is luminance, meaning that light intensity is nonlinearly encoded based on gamma corrected RGB primaries.
>
> Y′CbCr color spaces are defined by a mathematical coordinate transformation from an associated RGB color space. If the underlying RGB color space is absolute, the Y′CbCr color space is an absolute color space as well; conversely, if the RGB space is ill-defined, so is Y′CbCr.

> See Wikipedia for more.

We used `HighGUI` for image display and some other basic image IO functionality from OpenCV

We used `argparse` for constructing a user-friendly interface for our program

We used `JSON` format to store our configuration file. User can check the comment on `EigenFaceUtils.loadConfig` for a detailed illustration on the functionality of all the parameteres available

We'll put it here for convenience

```
    def loadConfig(self, filename):
        with open(filename, "rb") as f:
            data = json.load(f)
        log.info(f"Loading configuration from {filename}, with content: {data}")
        self.w = data["width"] # mask width
        self.h = data["height"] # mask height
        self.l = np.array(data["left"]) # left eye coordinates, x, y
        self.r = np.array(data["right"]) # right eye coordinates, x, y
        self.isColor = data["isColor"] # whether we should treat the model/input images as colored ones
        self.nEigenFaces = data["nEigenFaces"] # target eigenfaces to construct

        # setting this to null will reduce computation significantly
        # since we'll only have to compute nEigenFaces eigenvalues/eigenvectors
        # setting to a value would disable nEigenFaces parameter, computing from target information retain rate "targetPercentage"
        self.targetPercentage = data["targetPercentage"]

        # whether we should use cv2.PCACompute/cv2.PCACompute2 or our own PCA computation
        # note the builtin methods are much faster and memory efficient if we're only requiring
        # a small subset of all eigenvectors (don't have to allocate the covariance matrix)
        # enabling us to do large scale computation, even with colored image of 512 * 512
        self.useBuiltin = data["useBuiltin"]

        # ! depricated
        # Only HighGUI of OpenCV is supported.\nOther implementation removed due to regulation.
        # whether we should use matplotlib to draw results or HIGHGUI of opencv
        # I think HIGHGUI sucks at basic figure management, I'd prefer matplotlib for figure drawing
        # but if we're looking for a more general solution for GUI, it is a choice
        self.useHighgui = data["useHighgui"]
```

And we used `numpy` and `scipy` for matrix (arrays) manipulation heavily.

Like

- `matmul`
- `norm`
- `transpose`
- `flatten`
- `expand_dims`
- `copy`
- `squeeze`
- `zeros`
- `mean`
- `min`
- `max`
- `reshape`

The model is saved using compression API for numpy

```
@array_function_dispatch(_savez_compressed_dispatcher)
def savez_compressed(file, *args, **kwds):
    """
    Save several arrays into a single file in compressed ``.npz`` format.

    If keyword arguments are given, then filenames are taken from the keywords.
    If arguments are passed in with no keywords, then stored filenames are
    arr_0, arr_1, etc.
    """
     ...
```

We used `coloredlogs` to make our log info more intuitive for a reader and for us debugger.

Eigenfaces

There're some basic steps to take when constructing an eigenface-based face recognizor.

We'll have to:

1. Find the eye position of a given database image

   This can be done by hand (annotated by a human)

   Or, for convenience we can also use a haar cascade recognizor, which just comes with OpenCV

   `I'm aware of the fact that you asked us not to use face recognizor to get the face of the image`

   But I think we're performing this experiment to **learn more about eigenfaces, not to hone our own face recognition skills by labelling all eyes by hand** when already existing eye recognizor can to things much faster. Ain't computational machines created to free us human from doing those repetative work of computation?

   We can say we're rewriting some code to **learn more**

But I don't think we can learn more **by labelling more eyes**

2. Align all the eyes to the given position of our mask.

   In this process, we'll have to do some image transfor:

   - 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
3. 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.
4. Compute the **mean face** and subtract it from all the original image data
5. Construct the **covariance matrix**. Let's flatten the aligned, equalized face to batch and compute covariance matrix of all the possible pixel indices
6. Compute the **eigenvalues/eigenvectors** of the above mentioned covariance matrix.

   The eigenvectors with the largest eigenvalues are eigenfaces
7. To express the face in the form of eigenfaces, we'd take the dot product between the flattened image with all the above mentioned eigenvectors. The results that pop out are the compressed version of the image data with respect to the eigenfaces.
8. To reconstruct the face, simple to a matrix multiplication to sum up the weighted eigenvectors and add the mean face mentioned in 4. Unflatten it and we'll get the face
9. To recognize the face. Do a euclidean distance between the weights of the faces in the database and the weights computed in 8. The one with the smallest distance would be the most similar face in the database

Implementation

You'll notice that our implementation is pretty long, this is because we adopted detailed logging for a better debugging experience and we supported a tons of options:

1. You can choose to treat image as grayscale or color image (reconstruction is still done to a color image)
2. You can choose to set the number of eigenfaces beforehand or just set a target information retain rate
3. You can choose the size of the mask and the eye position of the left and the right one
4. You can choose to use OpenCV's implementation of PCA or ours
5. You were also able to choose to use HighGUI or just `matplotlib`, but this part is explicitly removed due to regulation

6. Theoretically, you can use whatever dataset you want, even the eyes/face are not annotated or they're of strange size.

   `You don't have to stick with the dataset provided by us`

   Because we've done carefully alignment

7. Our implementation doesn't have to be bounded with a certain dataset

   As long as it has some recognizable eye in it

   (And if you provide us with the eye position, we'll polite ask you to say the as:)

```
"""
We're assuming a <imageFileNameNoExt>.txt for eye position like
474 247 607 245
"""
```

8. The `EigenFaceUtils` class can be configured using a configuration file

   And the configuration can also be saved from a instantiated object

9. The configuration file doesn't have to have a model associated with it explicitly

   The only requirements are that the mask's size are matched and `isColor` field is set correctly

   So you can literally have multiple models linked to a configuration file

Training

```
    def train(self, path, imgext, txtext, modelName="model.npz"):
        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)
```

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

Reconstruction

```
    def reconstruct(self, img: np.ndarray) -> np.ndarray:
        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 = Infinity
        minName = ""
        for name in tqdm(self.faceDict.keys(), "Recognizing"):
            faceWeight = self.faceDict[name]
            dist = la.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)
```

```
33
34          # luckily, transpose of eigenvector is its inversion
35          # Eigenvectors of real symmetric matrices are orthogonal
36          # ! the magic happens here
37          # data has been lost because nEigenFaces is much smaller than the image
    dimension span
38          # which is width * height * color
39          # but because we're using PCA (principal components), most of the
    information will still be retained
40          flat = np.matmul(np.transpose(self.eigenVectors), weights)  # restored
41          log.info(f"Shape of flat: {flat.shape}")
42          flat = np.transpose(flat)
43          dst += self.unflatten(flat)
44          if self.isColor:
45              ori = np.zeros((self.h, self.w, 3))
46          else:
47              ori = np.zeros((self.h, self.w))
48          try:
49              ori = self.getImage(minName)
50              log.info(f"Successfully loaded the original image: {minName}")
51          except FileNotFoundError as e:
52              log.error(e)
53          dst = self.normalizeImg(dst)
54          eigen = self.normalizeImg(eigen)
55          face = self.normalizeImg(face)
56          return dst, eigen, face, ori, minName
```

The reconstruction procedure are also already illustrated.

Modules

Face Alignment

Align all the eyes to the given position of our mask.

In this process, we'll have to do some image transfor:

- 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

```
1       def alignFace2Mask(self, face: np.ndarray, left: np.ndarray, right:
    np.ndarray) -> np.ndarray:
2           faceVect = left - right
3           maskVect = self.l - self.r
4           log.info(f"Getting faceVect: {faceVect} and maskVect: {maskVect}")
5           faceNorm = np.linalg.norm(faceVect)
6           maskNorm = np.linalg.norm(maskVect)
7           log.info(f"Getting faceNorm: {faceNorm} and maskNorm: {maskNorm}")
8           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 scale > 1:
                # 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
                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))
            else:
                # 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
                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))
            return face
```

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.

```
    @staticmethod
    def equalizeHistColor(img):
        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

```
1
2       def getImage(self, name, manual_check=False) -> np.ndarray:
3           # the load the image accordingly
4           if self.isColor:
5               img = cv2.imread(name, cv2.IMREAD_COLOR)
6           else:
7               img = cv2.imread(name, cv2.IMREAD_GRAYSCALE)
8
9           # try getting eye position
10          eyes = self.getEyes(name, img)
11          log.info(f"Getting eyes: {eyes}")
12          if not len(eyes) == 2:
13              log.warning(f"Cannot get two eyes from this image: {name},
    {len(eyes)} eyes")
14              raise EigenFaceException("Bad image")
15
16          # align according to eye position
17          dst = self.alignFace2Mask(img, eyes[0], eyes[1])
18
19          # hist equalization
20          if self.isColor:
21              dst = self.equalizeHistColor(dst)
22          else:
23              dst = cv2.equalizeHist(dst)
24
25          # should we check every image before/after loading?
26          if manual_check:
27              cv2.imshow(name, dst)
28              cv2.waitKey()
29              cv2.destroyWindow(name)
30          return dst
31
32      def getImageFull(self, imgname) -> np.ndarray:
33          # Load the image specified and check for corresponding txt file to get
    the eye position from the file
34          txtname = f"{os.path.splitext(imgname)[0]}.txt"
35          if os.path.isfile(txtname):
36              self.updateEyeDictEntry(txtname)
37
38          log.info(f"Loading image: {imgname}")
39          return self.getImage(imgname)
40
41      def updateBatchData(self, path="./", ext=".jpg", manual_check=False,
    append=False) -> np.ndarray:
42          # get all image from a path with a specific extension
43          # align them, update histogram and add the to self.batch
44          # adjust logging level to be quite or not
45          prevLevel = coloredlogs.get_level()
46          if not manual_check:
```

```
47              coloredlogs.set_level("WARNING")
48
49          self.pathList = os.listdir(path)
50          self.pathList = [os.path.join(path, name) for name in self.pathList if
    name.endswith(ext)]
51          names = self.pathList
52          if not append:
53              if self.isColor:
54                  self.batch = np.ndarray((0, self.colorLen))  # assuming color
55              else:
56                  self.batch = np.ndarray((0, self.grayLen))
57          bads = []
58          for index, name in tqdm(enumerate(names), desc="Processing batch"):
59              try:
60                  dst = self.getImage(name, manual_check)
61                  flat = dst.flatten()
62                  flat = np.reshape(flat, (1, len(flat)))
63                  self.batch = np.concatenate([self.batch, flat])
64              except EigenFaceException as e:
65                  log.warning(e)
66                  bads.append(index)
67          for bad in bads[::-1]:
68              del names[bad]
69
70          coloredlogs.set_level(prevLevel)
71          log.info(f"Getting {len(names)} names and {self.batch.shape[0]}
    batch")
72          return self.batch
73
74      def updateEyeDict(self, path="./", ext=".eye", manual_check=False) →
    dict:
75          # get all possible eyes position from the files with a specific
    extension
76          # and add the to self.eyeDict
77          prevLevel = coloredlogs.get_level()
78          if not manual_check:
79              coloredlogs.set_level("WARNING")
80
81          names = os.listdir(path)
82          names = [os.path.join(path, name) for name in names if
    name.endswith(ext)]
83          log.info(f"Good names: {names}")
84          for name in names:
85              # iterate through all txt files
86              self.updateEyeDictEntry(name)
87
88          # restore the logging level
89          coloredlogs.set_level(prevLevel)
90          return self.eyeDict
91
92      def updateEyeDictEntry(self, name):
93          # update the dictionary but only one entry
```

```
94            with open(name, "r") as f:
95                lines = f.readlines()
96                log.info(f"Processing: {name}")
97                for line in lines:  # actually there should only be one line
98                    line = line.strip()  # get rid of starting/ending space \n
99                    # assuming # starting line to be comment
100                   if line.startswith("#"):  # get rid of comment file
101                       log.info(f"Getting comment line: {line}")
102                       continue
103                   coords = line.split()
104                   name = os.path.basename(name)  # get file name
105                   name = os.path.splitext(name)[0]  # without ext
106                   if len(coords) == 4:
107                       self.eyeDict[name] =
    np.reshape(np.array(coords).astype(int), [2, 2])
108                       order = np.argsort(self.eyeDict[name][:, 0])  # sort by
    first column, which is x
109                       self.eyeDict[name] = self.eyeDict[name][order]
110                   else:
111                       log.error(f"Wrong format for file: {name}, at line:
    {line}")
112           return self.eyeDict[name]
```

Get Mean Face

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

```
1       def updateMean(self):
2           assert self.batch is not None
3           # get the mean values of all the vectorized faces
4           self.mean = np.reshape(np.mean(self.batch, 0), (1, -1))
5           log.info(f"Getting mean vectorized face: {self.mean} with shape:
    {self.mean.shape}")
6           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`

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

A costom slow implementation:

```
    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)

    @staticmethod
    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

```
    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 = la.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 = sla.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}")
```

Face Dict

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

```
    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
        # 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 = 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
```

Utilities

Load/Save Configuration File

```
    @staticmethod
    def randcolor():
        '''
        Generate a random color, as list
        '''
        return [random.randint(0, 256) for _ in range(3)]

    def loadConfig(self, filename):
        with open(filename, "rb") as f:
            data = json.load(f)
        log.info(f"Loading configuration from {filename}, with content: {data}")
        self.w = data["width"] # mask width
        self.h = data["height"] # mask height
        self.l = np.array(data["left"]) # left eye coordinates, x, y
        self.r = np.array(data["right"]) # right eye coordinates, x, y
        self.isColor = data["isColor"] # whether we should treat the model/input images as colored ones
        self.nEigenFaces = data["nEigenFaces"] # target eigenfaces to construct

        # setting this to null will reduce computation significantly
        # since we'll only have to compute nEigenFaces eigenvalues/eigenvectors
        # setting to a value would disable nEigenFaces parameter, computing from target information retain rate "targetPercentage"
        self.targetPercentage = data["targetPercentage"]

        # whether we should use cv2.PCACompute/cv2.PCACompute2 or our own PCA computation
        # note the builtin methods are much faster and memory efficient if we're only requiring
        # a small subset of all eigenvectors (don't have to allocate the covariance matrix)
        # enabling us to do large scale computation, even with colored image of 512 * 512
        self.useBuiltin = data["useBuiltin"]

        # ! depricated
        # Only HighGUI of OpenCV is supported.\nOther implementation removed due to regulation.
        # whether we should use matplotlib to draw results or HIGHGUI of opencv
        # I think HIGHGUI sucks at basic figure management, I'd prefer matplotlib for figure drawing
        # but if we're looking for a more general solution for GUI, it is a choice
        self.useHighgui = data["useHighgui"]

    def saveConfig(self, filename):
```

```
38          data = {}
39          data["width"] = self.w
40          data["height"] = self.h
41          data["left"] = self.l.tolist()
42          data["right"] = self.r.tolist()
43          data["isColor"] = self.isColor
44          data["nEigenFaces"] = self.nEigenFaces
45          data["targetPercentage"] = self.targetPercentage
46          data["useBuiltin"] = self.useBuiltin
47          data["useHighgui"] = self.useHighgui
48
49          log.info(f"Dumping configuration to {filename}, with content: {data}")
50          with open(filename, "wb") as f:
51              json.dump(data, f)
```

Load/Save Eigenmodel

```
1       def loadModel(self, modelName):
2           # load previous eigenvectors/mean value
3           log.info(f"Loading model: {modelName}")
4           data = np.load(modelName, allow_pickle=True)
5           try:
6               self.eigenVectors = data["arr_0"]
7           except KeyError as e:
8               log.error(f"Cannot load eigenvectors, {e}")
9           try:
10              self.mean = data["arr_1"]
11          except KeyError as e:
12              log.error(f"Cannot load mean face data, {e}")
13          try:
14              self.faceDict = data["arr_2"].item()
15          except KeyError as e:
16              log.error(f"Cannot load face dict, {e}")
17          log.info(f"Getting mean vectorized face: {self.mean} with shape:
    {self.mean.shape}")
18          log.info(f"Getting sorted eigenvectors:\n{self.eigenVectors}\nof shape:
    {self.eigenVectors.shape}")
19          log.info(f"Getting face dict of length: {len(self.faceDict)}")
20
21      def saveModel(self, modelName):
22          log.info(f"Saving model: {modelName}")
23          np.savez_compressed(modelName, self.eigenVectors, self.mean,
    self.faceDict)
24          log.info(f"Model: {modelName} saved")
```

Others

And, there also some other utility functions

```

    def unflatten(self, flat: np.ndarray) -> np.ndarray:
        # rubust method for reverting a flat matrix
        if len(flat.shape) == 2:
            length = flat.shape[1]
        else:
            length = flat.shape[0]
        if length == self.grayLen:
            if self.isColor:
                log.warning("You're reshaping a grayscale image when color is wanted")
            return np.reshape(flat, (self.h, self.w))
        elif length == self.colorLen:
            if not self.isColor:
                log.warning("You're reshaping a color image when grayscale is wanted")
            return np.reshape(flat, (self.h, self.w, 3))
        else:
            raise EigenFaceException(f"Unsupported flat array of length: {length}, should provide {self.grayLen} or {self.colorLen}")

    def uint8unflatten(self, flat):
        # for displaying
        img = self.unflatten(flat)
        return img.astype("uint8")

    def updatenEigenFaces(self) -> int:
        assert self.eigenValues is not None
        # get energy
        self.nEigenFaces = len(self.eigenValues)
        targetValue = np.sum(self.eigenValues) * self.targetPercentage
        accumulation = 0
        for index, value in enumerate(self.eigenValues):
            accumulation += value
            if accumulation > targetValue:
                self.nEigenFaces = index + 1
                log.info(f"For a energy percentage of {self.targetPercentage}, we need {self.nEigenFaces} vectors from {len(self.eigenValues)}")
                break  # current index should be nEigenFaces

        return self.nEigenFaces

    # ! unused
    def updateEigenFaces(self) -> np.ndarray:
        assert self.eigenVectors is not None
        log.info(f"Computing eigenfaces")
```

```
43          self.eigenFaces = np.array([self.unflatten(vector) for vector in
    self.eigenVectors])
44          log.info(f"Getting eigenfaces of shape {self.eigenFaces.shape}")
45          return self.eigenFaces
46
47      @staticmethod
48      def normalizeImg(mean: np.ndarray) -> np.ndarray:
49          return ((mean-np.min(mean)) * 255 / (np.max(mean)-
    np.min(mean))).astype("uint8")
50
51      def drawEigenFaces(self, rows=None, cols=None) -> np.ndarray:
52          assert self.eigenFaces is not None
53          # get a canvas for previewing the eigenfaces
54          faces = self.eigenFaces
55          if rows is None or cols is None:
56              faceCount = faces.shape[0]
57              rows = int(math.sqrt(faceCount))  # truncating
58              cols = faceCount // rows
59              while rows*cols < faceCount:
60                  cols += 1
61              # has to be enough
62          else:
63              faceCount = rows * cols
64          log.info(f"Getting canvas for rows: {rows}, cols: {cols}, faceCount:
    {faceCount}")
65
66          if self.isColor:
67              canvas = np.zeros((rows * faces.shape[1], cols * faces.shape[2],
    3), dtype="uint8")
68          else:
69              canvas = np.zeros((rows * faces.shape[1], cols * faces.shape[2]),
    dtype="uint8")
70
71          for index in range(faceCount):
72              i = index // cols
73              j = index % cols
74              canvas[i * faces.shape[1]:(i+1)*faces.shape[1], j * faces.shape[2]:
    (j+1)*faces.shape[2]] = self.normalizeImg(faces[index])
75              log.info(f"Filling EigenFace of {index} at {i}, {j}")
76
77          return canvas
78
79      def getMeanFace(self) -> np.ndarray:
80          assert self.eigenFaces is not None
81          faces = self.eigenFaces
82          mean = np.mean(faces, 0)
83          mean = np.squeeze(mean)
84          mean = self.normalizeImg(mean)
85          log.info(f"Getting mean eigenface\n{mean}\nof shape: {mean.shape}")
86          return mean
```

Usage and Experiments

Check out the `train.py` and `test.py` file for more detailed implementation

And there's another list of options:

You'll notice that our implementation is pretty long, this is because we adopted detailed logging for a better debugging experience and we supported a tons of options:

1. You can choose to treat image as grayscale or color image (reconstruction is still done to a color image)
2. You can choose to set the number of eigenfaces beforehand or just set a target information retain rate
3. You can choose the size of the mask and the eye position of the left and the right one
4. You can choose to use OpenCV's implementation of PCA or ours
5. You were also able to choose to use HighGUI or just `matplotlib`, but this part is explicitly removed due to regulation
6. Theoretically, you can use whatever dataset you want, even the eyes/face are not annotated or they're of strange size.

   `You don't have to stick with the dataset provided by us`

   Because we've done carefully alignment
7. Our implementation doesn't have to be bounded with a certain dataset

   As long as it has some recognizable eye in it

   (And if you provide us with the eye position, we'll polite ask you to say the as:)

   ```
   """
   We're assuming a <imageFileNameNoExt>.txt for eye position like
   474 247 607 245
   """
   ```
8. The `EigenFaceUtils` class can be configured using a configuration file

   And the configuration can also be saved from a instantiated object
9. The configuration file doesn't have to have a model associated with it explicitly

   The only requirements are that the mask's size are matched and `isColor` field is set correctly

   So you can literally have multiple models linked to a configuration file

Basic Usage

Basically, you set your configuration file like this:

```
{
    "width": 512,
    "height": 512,
```

```
    "left": [
        188,
        188
    ],
    "right": [
        324,
        188
    ],
    "isColor": true,
    "nEigenFaces": 1000,
    "targetPercentage": null,
    "useBuiltin": true,
    "useHighgui": true
}
```

Eye positions would be pretty straight forward.

`isColor` defines how our program will treat the input images and generate output image

`nEigenFaces` defines the number of eigenfaces you want to generate, this option can be overwritten by `targetPercentage`, which is the target information retain rate you want to achieve.

If `targetPercentage` is set to `null` here (`None` in python implementation), we'll use `nEigenFaces`. Else we'll try computing `nEigenFaces` manually.

`useBuiltIn` means you want to use the `PCACompute` implementation of OpenCV, which is way more faster and memory efficient than our own version.

`useHighgui` is depricated. Leave it to `true`

Then you can run the `train.py` or `test.py` with the correct arguments provided:

You can print the help message using `python train.py -h` or `python test.py -h`

```
usage: train.py [-h] [-p PATH] [-i IMGEXT] [-t TXTEXT] [-c CONFIG] [-m MODEL]


We're assuming a <imageFileNameNoExt>.txt for eye position like
474 247 607 245
Comment line starting with # will be omitted
Or we'll use OpenCV's haarcascade detector to try locating eyes' positions

optional arguments:
  -h, --help            show this help message and exit
  -p PATH, --path PATH  The path of the data, we're assuming that all files are

                        under this directory
  -i IMGEXT, --imgext IMGEXT
                        The extension of the image file like .jpg, .png or even

                        .pgm
  -t TXTEXT, --txtext TXTEXT
```

```
                        The text file extension we want to read eye positions
off
                        from, usually .txt. But others will work too
  -c CONFIG, --config CONFIG
                        The configuration file for the eigenface utility
instance
  -m MODEL, --model MODEL
                        The model trained with this eigenface utility
```

```
usage: test.py [-h] [-i INPUT] [-m MODEL] [-c CONFIG] [-o OUTPUT]

We're assuming a <imageFileNameNoExt>.txt for eye position like
474 247 607 245
Comment line starting with # will be omitted
Or we'll use OpenCV's haarcascade detector to try locating eyes' positions

Note that if you want to save the recognition result
pass -o argument to specify the output file name
It's highly recommended to do so since OpenCV can't even draw large window
properly ...

optional arguments:
  -h, --help            show this help message and exit
  -i INPUT, --input INPUT
                        The image we want to reconstruct and recognize on
  -m MODEL, --model MODEL
                        The model trained with this eigenface utility
  -c CONFIG, --config CONFIG
                        The configuration file for the eigenface utility
instance
  -o OUTPUT, --output OUTPUT
                        The output file to save the reconstruction/recognition

                        result. If not specified, the program WILL NOT SAVE THE

                        RESULT
```

An example run would be:

```
python train.py -p "MyDataSet" -i .jpg -t .txt -c builtin.json -m model.color.npz

python test.py -i image_0318.jpg -m model.color.npz -c builtin.json -o
similar.png
```

Dataset

Our `EigenFaceUtils` can operate on the given dataset and also our own.

We constructed our dataset by labelling the eye position of *Caltec Database* and adding a few of the developer's face.

It was pretty awkward...

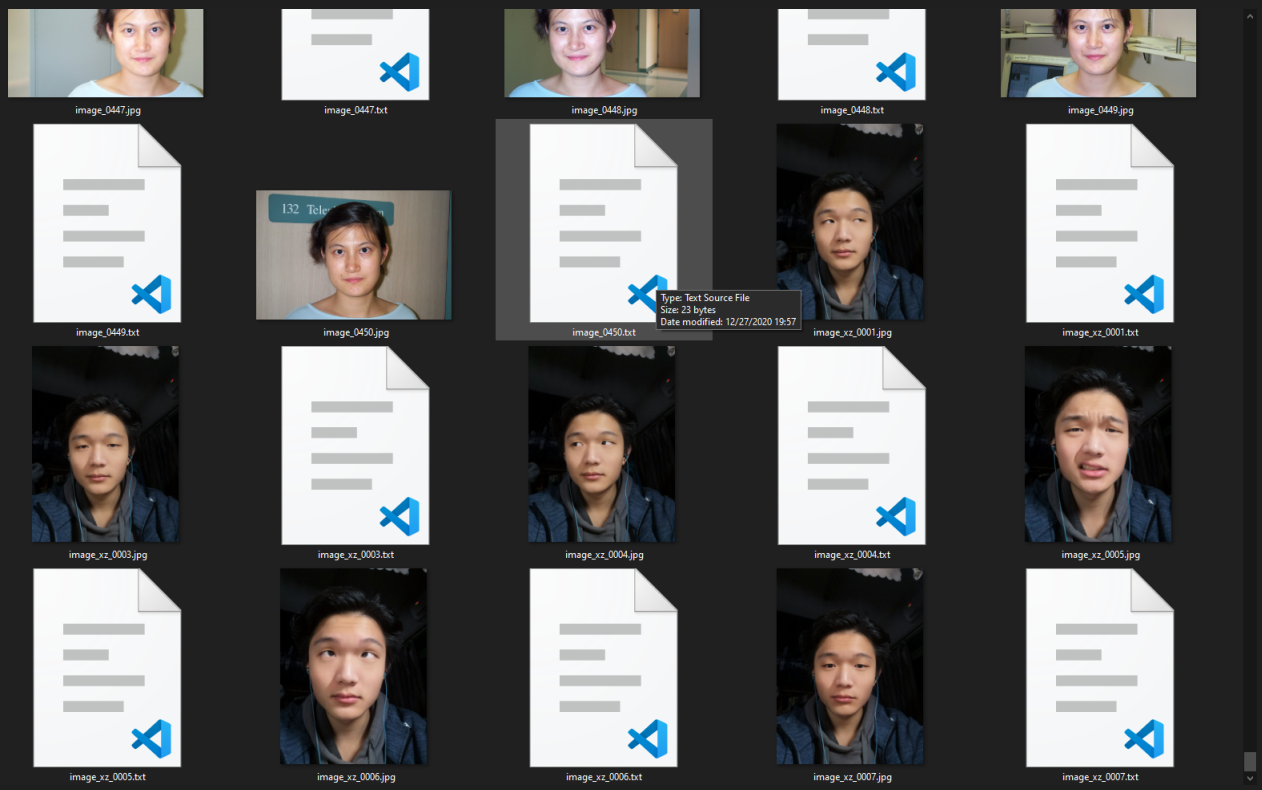
To make our life simplier, we only selected those images in *Caltec Database* that has a recognizable eyes to OpenCV's haar cascade recognizer.

And hand labelled the developer's eyes.

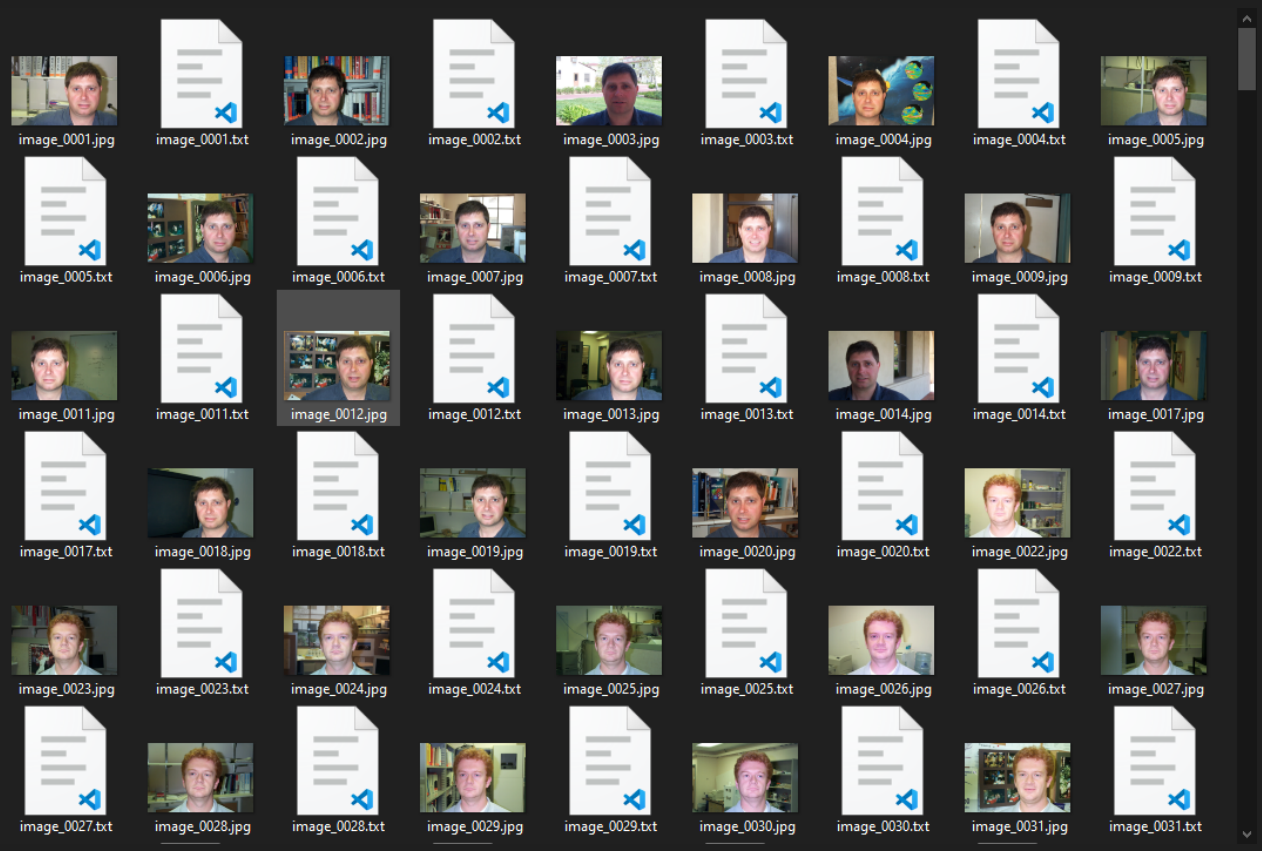
We provided 7 images of the developer, with label:

```
"image_xz_0001.jpg", "image_xz_0001.txt"
"image_xz_0002.jpg", "image_xz_0002.txt"
"image_xz_0003.jpg", "image_xz_0003.txt"
"image_xz_0004.jpg", "image_xz_0004.txt"
"image_xz_0005.jpg", "image_xz_0005.txt"
"image_xz_0006.jpg", "image_xz_0006.txt"
"image_xz_0007.jpg", "image_xz_0007.txt"
```

You can find the dataset in `MyDataSet` directory under the source directory



All other images in this directory are also labelled.



Experiments And Results

Training

We trained our eigenface model using this setup:

```
{
    "width": 512,
    "height": 512,
    "left": [
        188,
        188
    ],
    "right": [
        324,
        188
    ],
    "isColor": true,
    "nEigenFaces": 1000,
    "targetPercentage": null,
    "useBuiltin": true,
    "useHighgui": true
}
```

And we used this command on a Ubuntu machine:

```
./train.py -p "MyDataSet" -i .jpg -t .txt -c builtin.json -m
model.MyDataSet512×512×3.npz
```

with hardware specification:

```
                           ./+o+-       xzdd@xzdd-ubuntu
                   yyyyy- -yyyyyy+      OS: Ubuntu 20.04 focal
                ://+//////-yyyyyyo      Kernel: x86_64 Linux 5.4.0-58-generic
            .++ .:/++++++/-.+sss/`      Uptime: 3d 13h 49m
          .:++o:  /++++++++/:--:/-      Packages: 1909
         o:+o+:++.`..```.-/oo+++++/     Shell: zsh 5.8
        .:+o:+o/.          `+sssoo+/    Disk: 338G / 670G (54%)
   .++/+:+oo+o:`             /sssooo.   CPU: Intel Core i7-9700 @ 8x 4.7GHz
  [34.0°C]
  /+++//+:`oo+o               /::--:.   GPU: GeForce RTX 2080 Ti
  \+/+o+++`o++o               ++////.   RAM: 951MiB / 15921MiB
   .++.o+++oo+:`             /dddhhh.
        .+.o+oo:.           `oddhhhh+
         \+.++o+o``-````.:ohdhhhhh+
          `:o+++ `ohhhhhhhhyo++os:
            .o:`.syhhhhhhh/.oo++o`
                /osyyyyyyo++ooo+++/
                    ````` +oo+++o\:
                           `oo++.
```

This is the program loading images and aligning them to our mask.

```
(py38) → homework3 git:(master) ./train.py -p "MyDataSet" -i .jpg -t .txt -c builtin.json -m model.MyDataSet512×512×3.npz
2020-12-28 11:03:51 xzdd-ubuntu eigenface[122502] INFO Loading configuration from builtin.json, with content: {'width': 512, 'height': 512, 'left': [188, 188], 'right': [324, 188], 'i
sColor': True, 'nEigenFaces': 1000, 'targetPercentage': None, 'useBuiltin': True, 'useHighgui': True}
Processing batch: 113it [00:08,  8.14it/s]
```

This is the program computing eigenvectors/eigenvalues using `PCA`

```
(py38) → homework3 git:(master) ./train.py -p "MyDataSet" -i .jpg -t .txt -c builtin.json -m model.MyDataSet512×512×3.npz
2020-12-28 11:03:51 xzdd-ubuntu eigenface[122502] INFO Loading configuration from builtin.json, with content: {'width': 512, 'height': 512, 'left': [188, 188], 'right': [324, 188], 'i
sColor': True, 'nEigenFaces': 1000, 'targetPercentage': None, 'useBuiltin': True, 'useHighgui': True}
Processing batch: 345it [01:06,  5.20it/s]
2020-12-28 11:04:58 xzdd-ubuntu eigenface[122502] INFO Getting 345 names and 345 batch
2020-12-28 11:04:58 xzdd-ubuntu eigenface[122502] INFO Beginning builtin PCACompute for 1000 eigenvalues/eigenvectors
```

Note that we set `targetPercentage` to `null` so we'll use `nEigenFaces: 1000` to try computing 1000 eigenfaces (possibly 512*512*3=786432)

Eventually we can only get 345 eigenfaces:

```
(py38) → homework3 git:(master) ./train.py -p "MyDataSet" -i .jpg -t .txt -c builtin.json -m model.MyDataSet512×512×3.npz
2020-12-28 11:03:51 xzdd-ubuntu eigenface[122502] INFO Loading configuration from builtin.json, with content: {'width': 512, 'height': 512, 'left': [188, 188], 'right': [324, 188], 'i
sColor': True, 'nEigenFaces': 1000, 'targetPercentage': None, 'useBuiltin': True, 'useHighgui': True}
Processing batch: 345it [01:06,  5.20it/s]
2020-12-28 11:04:58 xzdd-ubuntu eigenface[122502] INFO Getting 345 names and 345 batch
2020-12-28 11:04:58 xzdd-ubuntu eigenface[122502] INFO Beginning builtin PCACompute for 1000 eigenvalues/eigenvectors
2020-12-28 11:06:08 xzdd-ubuntu eigenface[122502] INFO Getting mean vectorized face: [[104.96231884 115.71594203 113.35072464 ... 123.13913043 117.17101449
  106.61449275]] with shape: (1, 786432)
2020-12-28 11:06:08 xzdd-ubuntu eigenface[122502] INFO Getting sorted eigenvectors:
[[ 1.51637383e-03  1.52720111e-03  1.65011839e-03 ... -1.59191082e-06
   1.19530430e-04  2.64387095e-04]
 [-6.25459418e-04 -6.74333024e-04 -7.32235834e-04 ...  3.66228402e-03
   3.86894434e-03  3.52330871e-03]
 [-1.14571354e-03 -1.15501789e-03 -1.01845025e-03 ...  3.27172565e-04
   1.00252147e-04 -2.35212609e-05]
 ...
 [-4.36742887e-04 -2.96260371e-04 -1.22413086e-03 ... -1.02809826e-03
  -8.18479782e-04 -9.07450186e-04]
 [ 1.93892789e-04  1.39758695e-04  1.53280597e-04 ... -1.36009244e-03
  -1.30934660e-03 -2.01625493e-03]
 [-1.76967576e-03 -9.32994634e-04 -6.18863852e-04 ... -1.55962110e-03
  -1.26697964e-03 -1.51802196e-03]]
of shape: (345, 786432)
FaceDict:  97%|                                                                                | 334/345 [01:30<00:03,  3.64it/s]
```

And this is the program compressing image data using eigenfaces

The compressed image data, from ((512x512x3)^2)x8 bits to 345 x 512x512x3x64 bit (or we could even use 8 bit values), gets a compression ratio of 345x8/(512x512x3) = 0.0035

Then it tries to save the model to the file we specified in the commandline arguments
```

```
(py38) → homework3 git:(master) ./train.py -p "MyDataSet" -i .jpg -t .txt -c builtin.json -m model.MyDataSet512×512×3.npz
2020-12-28 11:03:51 xzdd-ubuntu eigenface[122502] INFO Loading configuration from builtin.json, with content: {'width': 512, 'height': 512, 'left': [188, 188], 'right': [324, 188], 'i
sColor': True, 'nEigenFaces': 1000, 'targetPercentage': None, 'useBuiltin': True, 'useHighgui': True}
Processing batch: 345it [01:06,  5.20it/s]
2020-12-28 11:04:58 xzdd-ubuntu eigenface[122502] INFO Getting 345 names and 345 batch
2020-12-28 11:04:58 xzdd-ubuntu eigenface[122502] INFO Beginning builtin PCACompute for 1000 eigenvalues/eigenvectors
2020-12-28 11:06:08 xzdd-ubuntu eigenface[122502] INFO Getting mean vectorized face: [[104.96231884 115.71594203 113.35072464 ... 123.13913043 117.17101449
  106.61449275]] with shape: (1, 786432)
2020-12-28 11:06:08 xzdd-ubuntu eigenface[122502] INFO Getting sorted eigenvectors:
[[ 1.51637383e-03  1.52720111e-03  1.65011839e-03 ... -1.59191082e-06
   1.19530430e-04  2.64387095e-04]
 [-6.25459418e-04 -6.74333024e-04 -7.32235834e-04 ...  3.66228402e-03
   3.86894434e-03  3.52330871e-03]
 [-1.14571354e-03 -1.15501789e-03 -1.01845025e-03 ...  3.27172565e-04
   1.00252147e-04 -2.35212609e-05]
 ...
 [-4.36742887e-04 -2.96260371e-04 -1.22413086e-03 ... -1.02809826e-03
  -8.18479782e-04 -9.07450186e-04]
 [ 1.93892789e-04  1.39758695e-04  1.53280597e-04 ... -1.36009244e-03
  -1.30934660e-03 -2.01625493e-03]
 [-1.76967576e-03 -9.32994634e-04 -6.18863852e-04 ... -1.55962110e-03
  -1.26697964e-03 -1.51802196e-03]]
of shape: (345, 786432)
FaceDict: 100%|██████████| 345/345 [01:33<00:00,  3.68it/s]
2020-12-28 11:07:41 xzdd-ubuntu eigenface[122502] INFO Got face dict of length 345
2020-12-28 11:07:41 xzdd-ubuntu eigenface[122502] INFO Saving model: model.MyDataSet512×512×3.npz
```

And this is what happens when no GUI is booted up and we tried to show windows using `cv2.imshow`

```
screen
  [128 131 124]
  ...
  [ 86  75  75]
  [ 87  73  77]
  [ 87  76  81]]

 ...

 [[141 128 112]
  [152 140 127]
  [150 141 127]
  ...
  [139 146 144]
  [154 163 161]
  [156 167 161]]

 [[153 140 122]
  [170 154 139]
  [177 165 150]
  ...
  [137 148 144]
  [151 163 158]
  [155 168 163]]

 [[159 146 133]
  [160 147 136]
  [166 155 146]
  ...
  [138 149 146]
  [139 153 148]
  [144 159 155]]]
of shape: (512, 512, 3)
2020-12-28 11:08:50 xzdd-ubuntu eigenface[122502] INFO Getting canvas for rows: 3, cols: 4, faceCount: 12
2020-12-28 11:08:50 xzdd-ubuntu eigenface[122502] INFO Filling EigenFace of 0 at 0, 0
2020-12-28 11:08:50 xzdd-ubuntu eigenface[122502] INFO Filling EigenFace of 1 at 0, 1
2020-12-28 11:08:50 xzdd-ubuntu eigenface[122502] INFO Filling EigenFace of 2 at 0, 2
2020-12-28 11:08:50 xzdd-ubuntu eigenface[122502] INFO Filling EigenFace of 3 at 0, 3
2020-12-28 11:08:50 xzdd-ubuntu eigenface[122502] INFO Filling EigenFace of 4 at 1, 0
2020-12-28 11:08:50 xzdd-ubuntu eigenface[122502] INFO Filling EigenFace of 5 at 1, 1
2020-12-28 11:08:50 xzdd-ubuntu eigenface[122502] INFO Filling EigenFace of 6 at 1, 2
2020-12-28 11:08:50 xzdd-ubuntu eigenface[122502] INFO Filling EigenFace of 7 at 1, 3
2020-12-28 11:08:50 xzdd-ubuntu eigenface[122502] INFO Filling EigenFace of 8 at 2, 0
2020-12-28 11:08:50 xzdd-ubuntu eigenface[122502] INFO Filling EigenFace of 9 at 2, 1
2020-12-28 11:08:50 xzdd-ubuntu eigenface[122502] INFO Filling EigenFace of 10 at 2, 2
2020-12-28 11:08:50 xzdd-ubuntu eigenface[122502] INFO Filling EigenFace of 11 at 2, 3
qt.qpa.xcb: could not connect to display
qt.qpa.plugin: Could not load the Qt platform plugin "xcb" in "/home/xzdd/miniconda3/envs/py38/lib/python3.8/site-packages/cv2/qt/plugins" even though it was found.
This application failed to start because no Qt platform plugin could be initialized. Reinstalling the application may fix this problem.

Available platform plugins are: xcb, eglfs, minimal, minimalegl, offscreen, vnc.

[1]    122502 abort (core dumped)  ./train.py -p "MyDataSet" -i .jpg -t .txt -c builtin.json -m
(py38) → homework3 git:(master)
```

Fortunately, we'll save the images to files before loading all of them to the screen.

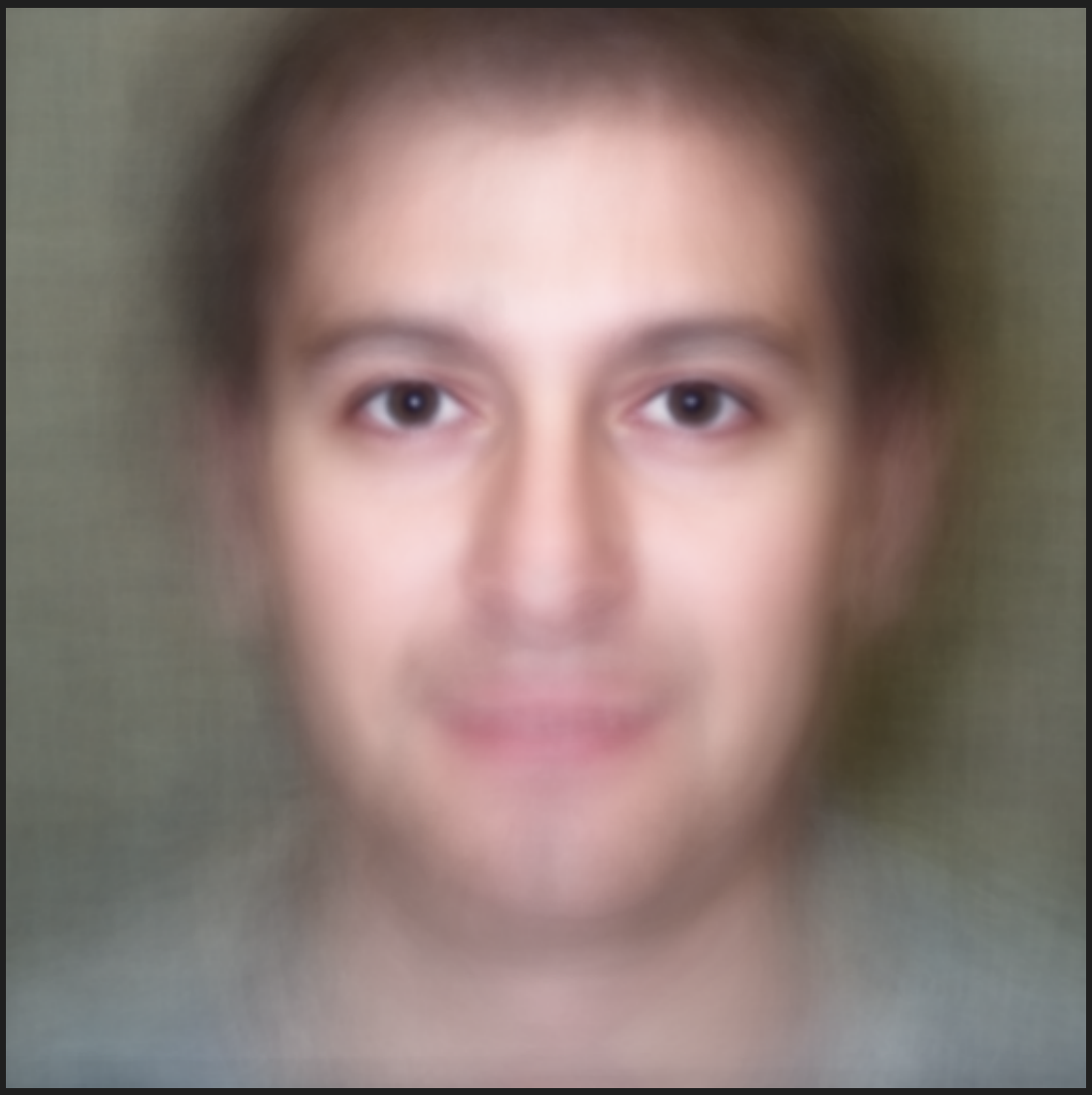
So if your running on a server without a display, this is the `success` message for our program

Let's checkout the result:

1. We specified a target number of eigenfaces of 1000

   But only 345 valid eigenvalues are computed, this is pretty reasonable

2. After alignment and color histogram equlization, we can get a mean face:

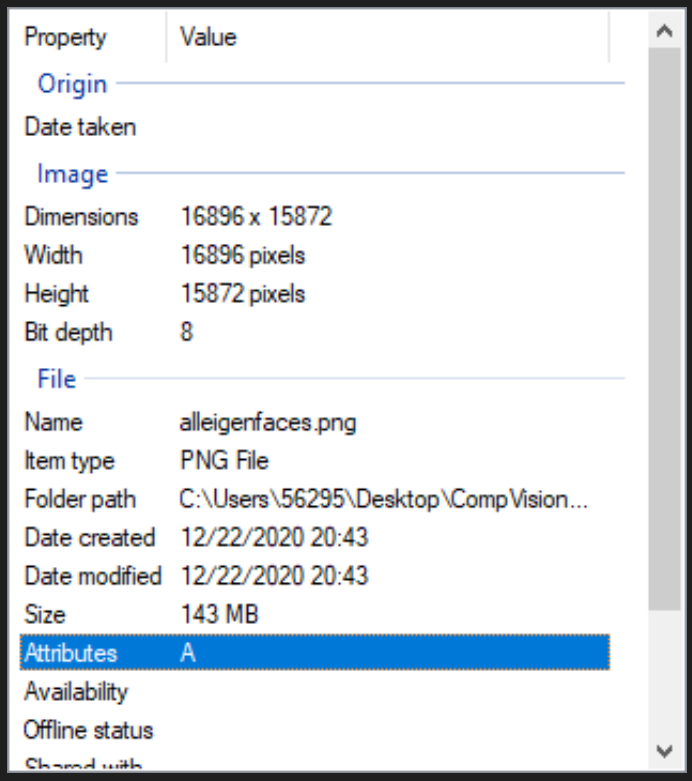


3. This is the first few eigenfaces:

The mean eigenface:

And we'd also generate a file with all the eigenfaces for you to exam:

But it's a little bit too large to be displayed here:



4. We've got a model of size: `-rw-rw-r-- 1 xzdd xzdd 2.0G Dec 28 11:08 model.MyDataSet512×512×3.npz`

Testing

Then, with the model and corresponding configuration file, we can use the eigenfaces we computed to compress our own image and get the reconstruction.

Then by comparing the Euclidean Distance between the projected test image with that of the ones in the database (saved to model), we can get the most similar face from the database and display (or render to file)

This is what we're going to do:

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:

```
1   img = mean
2   img += np.mul(np.transpose(vectors), weights)
```

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 (me):

```
(py38) ➜ homework3 git:(master) ./test.py -i image_xz_0002.jpg -m model.MyDataSet512x512x3.npz -c builtin.json -o similarxz0002.png
2020-12-28 15:20:27 xzdd-ubuntu eigenface[147319] INFO Loading configuration from builtin.json, with content: {'width': 512, 'height'
: 512, 'left': [188, 188], 'right': [324, 188], 'isColor': True, 'nEigenFaces': 1000, 'targetPercentage': None, 'useBuiltin': True, '
useHighgui': True}
2020-12-28 15:20:27 xzdd-ubuntu eigenface[147319] INFO Loading model: model.MyDataSet512x512x3.npz
2020-12-28 15:20:36 xzdd-ubuntu eigenface[147319] INFO Getting mean vectorized face: [[104.96231884 115.71594203 113.35072464 ... 123
.13913043 117.17101449
  106.61449275]] with shape: (1, 786432)
2020-12-28 15:20:36 xzdd-ubuntu eigenface[147319] INFO Getting sorted eigenvectors:
[[ 1.51637383e-03  1.52720111e-03  1.65011839e-03 ... -1.59191082e-06
   1.19530430e-04  2.64387095e-04]
 [-6.25459418e-04 -6.74333024e-04 -7.32235834e-04 ...  3.66228402e-03
   3.86894434e-03  3.52330871e-03]
 [-1.14571354e-03 -1.15501789e-03 -1.01845025e-03 ...  3.27172565e-04
   1.00252147e-04 -2.35212609e-05]
 ...
 [-4.36742887e-04 -2.96260371e-04 -1.22413086e-03 ... -1.02809826e-03
  -8.18479782e-04 -9.07450186e-04]
 [ 1.93892789e-04  1.39758695e-04  1.53280597e-04 ... -1.36009244e-03
  -1.30934660e-03 -2.01625493e-03]
 [-1.76967576e-03 -9.32994634e-04 -6.18863852e-04 ... -1.55962110e-03
  -1.26697964e-03 -1.51802196e-03]]
of shape: (345, 786432)
2020-12-28 15:20:36 xzdd-ubuntu eigenface[147319] INFO Getting face dict of length: 345
2020-12-28 15:20:36 xzdd-ubuntu __main__[147319] INFO Found text file
2020-12-28 15:20:36 xzdd-ubuntu eigenface[147319] INFO Processing: image_xz_0002.txt
2020-12-28 15:20:36 xzdd-ubuntu __main__[147319] INFO Loading image: image_xz_0002.jpg
2020-12-28 15:20:36 xzdd-ubuntu eigenface[147319] INFO Getting eyes: [[1514. 2780.]
 [2387. 2840.]]
2020-12-28 15:20:36 xzdd-ubuntu eigenface[147319] INFO Getting faceVect: [-873.  -60.] and maskVect: [-136    0]
2020-12-28 15:20:36 xzdd-ubuntu eigenface[147319] INFO Getting faceNorm: 875.0594265534198 and maskNorm: 136.0
2020-12-28 15:20:36 xzdd-ubuntu eigenface[147319] INFO Should scale the image to: 0.155418016049105
2020-12-28 15:20:36 xzdd-ubuntu eigenface[147319] INFO Should rotate the image: -90.0 - -93.93167148002225 = 3.9316714800222456 degre
es
2020-12-28 15:20:36 xzdd-ubuntu eigenface[147319] INFO Getting faceCenter: [1950.5 2810. ] and maskCenter: [256. 188.]
2020-12-28 15:20:36 xzdd-ubuntu eigenface[147319] INFO Should translate the image using: [-1694.5 -2622. ]
2020-12-28 15:20:36 xzdd-ubuntu eigenface[147319] INFO Getting # of channels: 3
2020-12-28 15:20:36 xzdd-ubuntu eigenface[147319] INFO Shape of eigenvectors and flat: (345, 786432), (786432, 1)
Recognizing: 100%|                                                                  | 345/345 [00:00<00:00, 95779.38it/s]
2020-12-28 15:20:37 xzdd-ubuntu eigenface[147319] INFO MOST SIMILAR FACE: MyDataSet/image_xz_0007.jpg WITH RESULT 22112.468775792233
2020-12-28 15:20:37 xzdd-ubuntu eigenface[147319] INFO Shape of flat: (786432, 1)
2020-12-28 15:20:37 xzdd-ubuntu eigenface[147319] INFO Processing: MyDataSet/image_xz_0007.txt
2020-12-28 15:20:37 xzdd-ubuntu eigenface[147319] INFO Getting eyes: [[1971. 2782.]
 [2422. 2797.]]
2020-12-28 15:20:37 xzdd-ubuntu eigenface[147319] INFO Getting faceVect: [-901.  -29.] and maskVect: [-136    0]
2020-12-28 15:20:37 xzdd-ubuntu eigenface[147319] INFO Getting faceNorm: 901.4665828526313 and maskNorm: 136.0
2020-12-28 15:20:37 xzdd-ubuntu eigenface[147319] INFO Should scale the image to: 0.1508652706455707
2020-12-28 15:20:37 xzdd-ubuntu eigenface[147319] INFO Should rotate the image: -90.0 - -91.84351185544986 = 1.8435118554498615 degre
es
2020-12-28 15:20:37 xzdd-ubuntu eigenface[147319] INFO Getting faceCenter: [1971.5 2782.5] and maskCenter: [256. 188.]
2020-12-28 15:20:37 xzdd-ubuntu eigenface[147319] INFO Should translate the image using: [-1715.5 -2594.5]
2020-12-28 15:20:37 xzdd-ubuntu eigenface[147319] INFO Getting # of channels: 3
2020-12-28 15:20:37 xzdd-ubuntu eigenface[147319] INFO Successfully loaded the original image: MyDataSet/image_xz_0007.jpg
2020-12-28 15:20:37 xzdd-ubuntu __main__[147319] INFO Saving output to similarxz0002.png
qt.qpa.xcb: could not connect to display
qt.qpa.plugin: Could not load the Qt platform plugin "xcb" in "/home/xzdd/miniconda3/envs/py38/lib/python3.8/site-packages/cv2/qt/plu
gins" even though it was found.
This application failed to start because no Qt platform plugin could be initialized. Reinstalling the application may fix this proble
m.

Available platform plugins are: xcb, eglfs, minimal, minimalegl, offscreen, vnc.

[1]    147319 abort (core dumped)  ./test.py -i image_xz_0002.jpg -m model.MyDataSet512x512x3.npz -c builtin.jso
(py38) ➜ homework3 git:(master)
```

loading model

loading from database

The command we used were:

```
./test.py -i image_xz_0002.jpg -m model.MyDataSet512×512×3.npz -c builtin.json -o similarxz0002.png
```

You can see that we were using a server so no windows could be opened.

But luckily the images are saved to the directory, so you can check them via either some service on your server or `scp` them back to your local directory

We rendered the images at full size, so this should be a 512 by 5x512 image

You can see the reconstructed test image do resembles me in some way.

And it corrected my closed eyes (due to the fact that most of the images in the dataset are open-eyed) and tilted head (due to unaligned labelling)

Moreover, even my clothes are being reconstructed

On the right side is the most similar database image to the one being tested and the person on that is indeed me.

The most astonishing part is that: the reconstructed database image is almost identical to the orignal one, meaning the compression loses almost no data at all, even though the compression ratio is already 0.0035 (without the whole model)

During the testing, you can see the speed of recognizing the face is indeed pretty fast

(Even though, due to the fact that we saved our eigenvectors using 64-bit float values with compression, the model file itself is pretty large and chunky to load from dist)

Another test result:

Testing with an existing image:

```
(py38) ➜ homework3 git:(master) ./test.py -i image_xz_0006.jpg -m model.MyDataSet512x512x3.npz -c builtin.json -o similarxz0006.png
2020-12-28 15:18:25 xzdd-ubuntu eigenface[146892] INFO Loading configuration from builtin.json, with content: {'width': 512, 'height'
: 512, 'left': [188, 188], 'right': [324, 188], 'isColor': True, 'nEigenFaces': 1000, 'targetPercentage': None, 'useBuiltin': True, '
useHighgui': True}
2020-12-28 15:18:25 xzdd-ubuntu eigenface[146892] INFO Loading model: model.MyDataSet512x512x3.npz
2020-12-28 15:18:34 xzdd-ubuntu eigenface[146892] INFO Getting mean vectorized face: [[104.96231884 115.71594203 113.35072464 ... 123
.13913043 117.17101449
  106.61449275]] with shape: (1, 786432)
2020-12-28 15:18:34 xzdd-ubuntu eigenface[146892] INFO Getting sorted eigenvectors:
[[ 1.51637383e-03  1.52720111e-03  1.65011839e-03 ... -1.59191082e-06
   1.19530430e-04  2.64387095e-04]
 [-6.25459418e-04 -6.74333024e-04 -7.32235834e-04 ...  3.66228402e-03
   3.86894434e-03  3.52330871e-03]
 [-1.14571354e-03 -1.15501789e-03 -1.01845025e-03 ...  3.27172565e-04
   1.00252147e-04 -2.35212609e-05]
 ...
 [-4.36742887e-04 -2.96260371e-04 -1.22413086e-03 ... -1.02809826e-03
  -8.18479782e-04 -9.07450186e-04]
 [ 1.93892789e-04  1.39758695e-04  1.53280597e-04 ... -1.36009244e-03
  -1.30934660e-03 -2.01625493e-03]
 [-1.76967576e-03 -9.32994634e-04 -6.18863852e-04 ... -1.55962110e-03
  -1.26697964e-03 -1.51802196e-03]]
of shape: (345, 786432)
2020-12-28 15:18:34 xzdd-ubuntu eigenface[146892] INFO Getting face dict of length: 345
2020-12-28 15:18:34 xzdd-ubuntu __main__[146892] INFO Found text file
2020-12-28 15:18:34 xzdd-ubuntu eigenface[146892] INFO Processing: image_xz_0006.txt
2020-12-28 15:18:34 xzdd-ubuntu __main__[146892] INFO Loading image: image_xz_0006.jpg
2020-12-28 15:18:34 xzdd-ubuntu eigenface[146892] INFO Getting eyes: [[1364. 2439.]
 [2551. 2497.]]
2020-12-28 15:18:34 xzdd-ubuntu eigenface[146892] INFO Getting faceVect: [-1187.   -58.] and maskVect: [-136    0]
2020-12-28 15:18:34 xzdd-ubuntu eigenface[146892] INFO Getting faceNorm: 1188.4161728956738 and maskNorm: 136.0
2020-12-28 15:18:34 xzdd-ubuntu eigenface[146892] INFO Should scale the image to: 0.11443802524886952
2020-12-28 15:18:34 xzdd-ubuntu eigenface[146892] INFO Should rotate the image: -90.0 - -92.79740037767522 = 2.797400377675217 degree
s
2020-12-28 15:18:34 xzdd-ubuntu eigenface[146892] INFO Getting faceCenter: [1957.5 2468. ] and maskCenter: [256. 188.]
2020-12-28 15:18:34 xzdd-ubuntu eigenface[146892] INFO Should translate the image using: [-1701.5 -2280. ]
2020-12-28 15:18:34 xzdd-ubuntu eigenface[146892] INFO Getting # of channels: 3
2020-12-28 15:18:34 xzdd-ubuntu eigenface[146892] INFO Shape of eigenvectors and flat: (345, 786432), (786432, 1)
Recognizing: 100%|                                                                                | 345/345 [00:00<00:00, 93098.81it/s]
2020-12-28 15:18:34 xzdd-ubuntu eigenface[146892] INFO MOST SIMILAR FACE: MyDataSet/image_xz_0006.jpg WITH RESULT 0.0
2020-12-28 15:18:35 xzdd-ubuntu eigenface[146892] INFO Shape of flat: (786432, 1)
2020-12-28 15:18:35 xzdd-ubuntu eigenface[146892] INFO Processing: MyDataSet/image_xz_0006.txt
2020-12-28 15:18:35 xzdd-ubuntu eigenface[146892] INFO Getting eyes: [[1364. 2439.]
 [2551. 2497.]]
2020-12-28 15:18:35 xzdd-ubuntu eigenface[146892] INFO Getting faceVect: [-1187.   -58.] and maskVect: [-136    0]
2020-12-28 15:18:35 xzdd-ubuntu eigenface[146892] INFO Getting faceNorm: 1188.4161728956738 and maskNorm: 136.0
2020-12-28 15:18:35 xzdd-ubuntu eigenface[146892] INFO Should scale the image to: 0.11443802524886952
2020-12-28 15:18:35 xzdd-ubuntu eigenface[146892] INFO Should rotate the image: -90.0 - -92.79740037767522 = 2.797400377675217 degree
s
2020-12-28 15:18:35 xzdd-ubuntu eigenface[146892] INFO Getting faceCenter: [1957.5 2468. ] and maskCenter: [256. 188.]
2020-12-28 15:18:35 xzdd-ubuntu eigenface[146892] INFO Should translate the image using: [-1701.5 -2280. ]
2020-12-28 15:18:35 xzdd-ubuntu eigenface[146892] INFO Getting # of channels: 3
2020-12-28 15:18:35 xzdd-ubuntu eigenface[146892] INFO Successfully loaded the original image: MyDataSet/image_xz_0006.jpg
2020-12-28 15:18:35 xzdd-ubuntu __main__[146892] INFO Saving output to similarxz0006.png
qt.qpa.xcb: could not connect to display
qt.qpa.plugin: Could not load the Qt platform plugin "xcb" in "/home/xzdd/miniconda3/envs/py38/lib/python3.8/site-packages/cv2/qt/plu
gins" even though it was found.
This application failed to start because no Qt platform plugin could be initialized. Reinstalling the application may fix this proble
m.

Available platform plugins are: xcb, eglfs, minimal, minimalegl, offscreen, vnc.

[1]    146892 abort (core dumped)  ./test.py -i image_xz_0006.jpg -m model.MyDataSet512x512x3.npz -c builtin.jso
```

loading model

image already in database

From the existing images' result, we can also confirm that the compression is indeed enough for accuracy and reducing dimension can significantly reduce the time required to do the computation of other kinds (utilizing change of basis)

Additional Experiments

Additional Training

Training with `isColor=false`, `targetPercentage=null` and `useBuiltn=false`:

The configuration we used:

```
{
    "width": 128,
    "height": 128,
    "left": [
        48,
        48
    ],
    "right": [
        80,
        48
    ],
    "isColor": false,
    "nEigenFaces": 50,
    "targetPercentage": null,
    "useBuiltin": false,
    "useHighgui": true
}
```

Here we're using grayscale training and our own implementation of PCA

We also constructed a small dataset to speed things up: `MySmallDataSet`

Everyone has only three faces here.

Here the `targetPercentage` is set to null, so we **won't have to compute all eigenvalues** to try reaching the target information retain rate

```
(local38) ➜ homework3 git:(master) X python ./train.py -p "MySmallDataSet" -i .jpg -t .txt -c default.json -m model.MySm
allDataSet128x128x1.npz
2020-12-28 16:13:17 DESKTOP-XZDDSPC eigenface[22492] INFO Loading configuration from default.json, with content: {'width'
: 128, 'height': 128, 'left': [48, 48], 'right': [80, 48], 'isColor': False, 'nEigenFaces': 32, 'targetPercentage': None,
 'useBuiltin': False, 'useHighgui': True}
Processing batch: 45it [00:00, 59.37it/s]
2020-12-28 16:13:18 DESKTOP-XZDDSPC eigenface[22492] INFO Getting 45 names and 45 batch
2020-12-28 16:13:18 DESKTOP-XZDDSPC eigenface[22492] INFO Getting mean vectorized face: [[105.2        106.         104.7
3333333 ... 115.8        116.71111111
  115.44444444]] with shape: (1, 16384)
2020-12-28 16:13:18 DESKTOP-XZDDSPC eigenface[22492] INFO Trying to compute the covariance matrix
2020-12-28 16:13:22 DESKTOP-XZDDSPC eigenface[22492] INFO Getting covar of shape: (16384, 16384)
2020-12-28 16:13:22 DESKTOP-XZDDSPC eigenface[22492] INFO Getting covariance matrix:
[[ 5559.84545455  5422.97727273  4975.94090909 ... -1017.50454545
  -1084.89545455 -1220.18181818]
 [ 5422.97727273  5483.54545455  4942.95454545 ...  -995.36363636
  -1061.95454545 -1221.18181818]
 [ 4975.94090909  4942.95454545  5227.38181818 ...  -405.6
   -709.96515152  -997.74242424]
 ...
 [-1017.50454545  -995.36363636  -405.6        ...  8662.39090909
   8570.55454545  8123.86363636]
 [-1084.89545455 -1061.95454545  -709.96515152 ...  8570.55454545
   8832.48282828  8476.54040404]
 [-1220.18181818 -1221.18181818  -997.74242424 ...  8123.86363636
   8476.54040404  8362.11616162]]
2020-12-28 16:13:22 DESKTOP-XZDDSPC eigenface[22492] INFO Begin computing 32 eigenvalues/eigenvectors
2020-12-28 16:13:30 DESKTOP-XZDDSPC eigenface[22492] INFO Getting 32 eigenvalues and eigenvectors with shape (16384, 32)
2020-12-28 16:13:30 DESKTOP-XZDDSPC eigenface[22492] INFO Getting sorted eigenvalues:
[13692077.82648995  8065424.82527987  4945670.55159279  3883130.78841176
  3030309.58099939  2835500.38371136  2427842.1930707   1911457.71950977
  1779447.25859033  1573182.62766051  1456963.11799481  1359633.86157442
  1155014.95786098  1056813.40248439   951043.12834829   872667.24346596
   860681.71336261   767367.56220602   743076.28065638   666788.53745266
   650917.3351369    594323.2315434    549146.13607777   522088.96079639
   503104.61461041   475584.82595126   466131.64726078   438166.47782317
   413767.387414     380024.56284485   360908.19535676   333803.59797119]
of shape: (32,)
2020-12-28 16:13:30 DESKTOP-XZDDSPC eigenface[22492] INFO Getting sorted eigenvectors:
[[-1.22082472e-02 -1.22170198e-02 -1.15940454e-02 ...  3.55293140e-03
   4.01347596e-03  5.11466732e-03]
 [-4.20471112e-03 -2.84525086e-03 -2.12906552e-03 ...  2.21606187e-02
   2.15921188e-02  1.99257795e-02]
 [ 1.20820334e-02  1.39370645e-02  9.93585603e-03 ... -2.05471490e-03
  -2.12010685e-03 -2.91506808e-04]
 ...
 [-2.58951293e-03  6.66316446e-04  3.51098408e-03 ...  2.07559640e-03
   1.43094726e-03 -3.75370247e-05]
 [ 3.16011767e-03  1.78737093e-03 -1.93208445e-03 ... -1.01141184e-02
  -7.52251863e-03 -3.59788604e-03]
 [-2.65106330e-02 -2.55080570e-02 -1.91679145e-02 ... -9.51263106e-03
  -5.91214268e-03 -4.91148062e-03]]
of shape: (32, 16384)
FaceDict: 100%|████████████████████████████████████████████████████████████| 45/45 [00:00<00:00, 4101.70it/s]
2020-12-28 16:13:31 DESKTOP-XZDDSPC eigenface[22492] INFO Got face dict of length 45
2020-12-28 16:13:31 DESKTOP-XZDDSPC eigenface[22492] INFO Saving model: model.MySmallDataSet128x128x1.npz
2020-12-28 16:13:31 DESKTOP-XZDDSPC eigenface[22492] INFO Model: model.MySmallDataSet128x128x1.npz saved
2020-12-28 16:13:31 DESKTOP-XZDDSPC eigenface[22492] INFO Computing eigenfaces
2020-12-28 16:13:31 DESKTOP-XZDDSPC eigenface[22492] INFO Getting eigenfaces of shape (32, 128, 128)
2020-12-28 16:13:31 DESKTOP-XZDDSPC eigenface[22492] INFO Getting mean eigenface
[[112 117 105 ... 111 111  89]
 [ 83  91  94 ... 113 107 114]
 [ 89 101 111 ... 129 123 112]
 ...
 [158 131  99 ... 166 156 140]
 [146 115 104 ... 168 180 163]
```

8 seconds to get 32 eigenvalues vectors

Note that computing **32 eigenvalues/eigenvectors** took **8 seconds**

Of course we'd get the basic results:

Mean Eigenfaces:

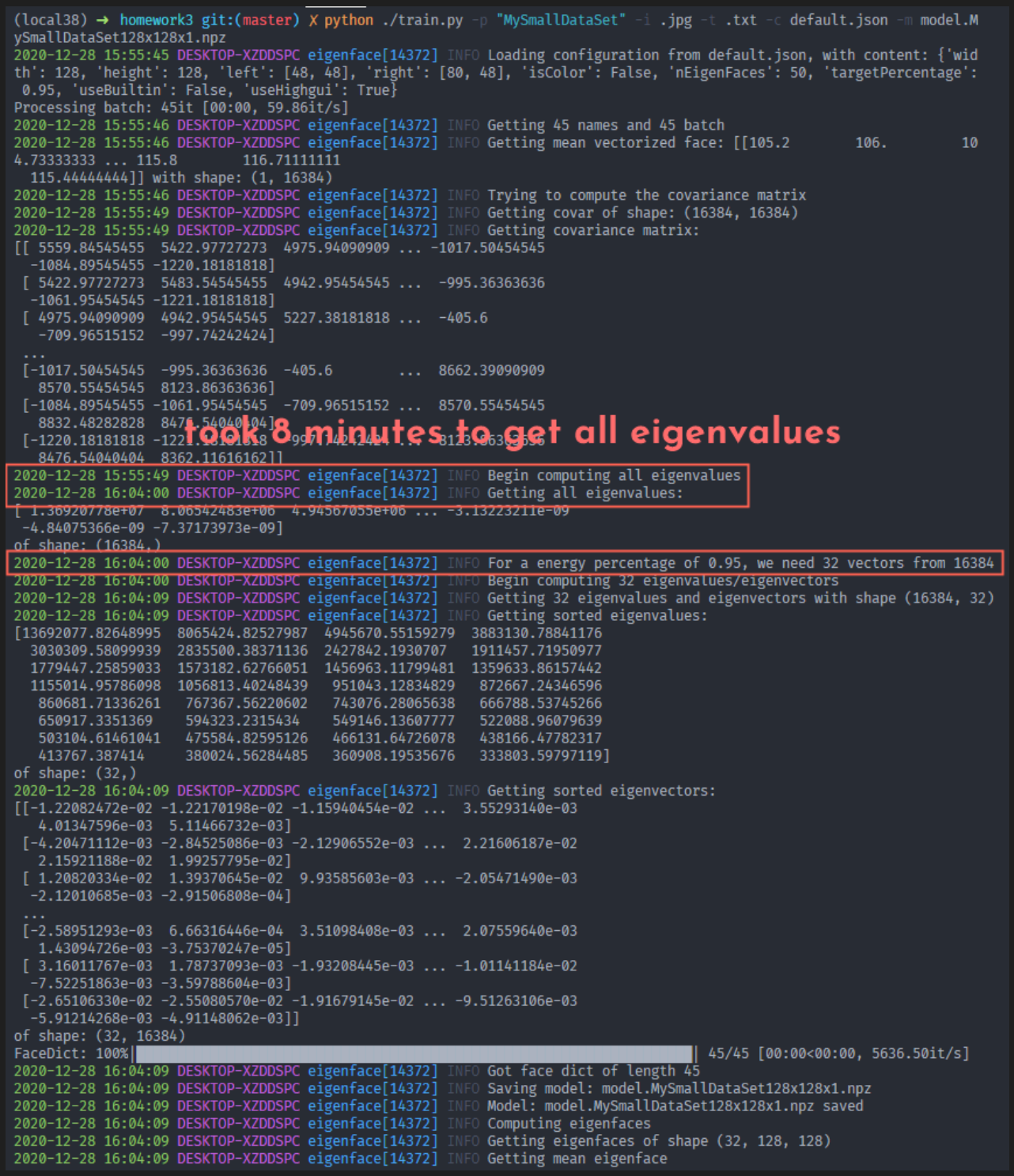
First 12 Eigenfaces:

Then we performed another training using `targetPercentage=0.95` to retain 0.95 of all the information

Training with `targetPercentage=0.95`


```
{
    "width": 128,
    "height": 128,
    "left": [
        48,
        48
    ],
    "right": [
        80,
        48
    ],
    "isColor": false,
    "nEigenFaces": 50,
    "targetPercentage": 0.95,
    "useBuiltin": false,
    "useHighgui": true
}
```


Other procedures are purely identical




```
(local38) ➜ homework3 git:(master) X python ./train.py -p "MySmallDataSet" -i .jpg -t .txt -c default.json -m model.M
ySmallDataSet128x128x1.npz
2020-12-28 15:55:45 DESKTOP-XZDDSPC eigenface[14372] INFO Loading configuration from default.json, with content: {'wid
th': 128, 'height': 128, 'left': [48, 48], 'right': [80, 48], 'isColor': False, 'nEigenFaces': 50, 'targetPercentage':
 0.95, 'useBuiltin': False, 'useHighgui': True}
Processing batch: 45it [00:00, 59.86it/s]
2020-12-28 15:55:46 DESKTOP-XZDDSPC eigenface[14372] INFO Getting 45 names and 45 batch
2020-12-28 15:55:46 DESKTOP-XZDDSPC eigenface[14372] INFO Getting mean vectorized face: [[105.2        106.         10
4.73333333 ... 115.8        116.71111111
  115.44444444]] with shape: (1, 16384)
2020-12-28 15:55:46 DESKTOP-XZDDSPC eigenface[14372] INFO Trying to compute the covariance matrix
2020-12-28 15:55:49 DESKTOP-XZDDSPC eigenface[14372] INFO Getting covar of shape: (16384, 16384)
2020-12-28 15:55:49 DESKTOP-XZDDSPC eigenface[14372] INFO Getting covariance matrix:
[[ 5559.84545455  5422.97727273  4975.94090909 ... -1017.50454545
  -1084.89545455 -1220.18181818]
 [ 5422.97727273  5483.54545455  4942.95454545 ...  -995.36363636
  -1061.95454545 -1221.18181818]
 [ 4975.94090909  4942.95454545  5227.38181818 ...  -405.6
   -709.96515152  -997.74242424]
 ...
 [-1017.50454545  -995.36363636  -405.6        ...  8662.39090909
   8570.55454545  8123.86363636]
 [-1084.89545455 -1061.95454545  -709.96515152 ...  8570.55454545
   8832.48282828  8476.54040404]
 [-1220.18181818 -1221.18181818  -997.74242424 ...  8123.86363636
   8476.54040404  8362.11616162]]
2020-12-28 15:55:49 DESKTOP-XZDDSPC eigenface[14372] INFO Begin computing all eigenvalues
2020-12-28 16:04:00 DESKTOP-XZDDSPC eigenface[14372] INFO Getting all eigenvalues:
[ 1.36920778e+07  8.06542483e+06  4.94567055e+06 ... -3.13223211e-09
 -4.84075366e-09 -7.37173973e-09]
of shape: (16384,)
2020-12-28 16:04:00 DESKTOP-XZDDSPC eigenface[14372] INFO For a energy percentage of 0.95, we need 32 vectors from 16384
2020-12-28 16:04:00 DESKTOP-XZDDSPC eigenface[14372] INFO Begin computing 32 eigenvalues/eigenvectors
2020-12-28 16:04:09 DESKTOP-XZDDSPC eigenface[14372] INFO Getting 32 eigenvalues and eigenvectors with shape (16384, 32)
2020-12-28 16:04:09 DESKTOP-XZDDSPC eigenface[14372] INFO Getting sorted eigenvalues:
[13692077.82648995  8065424.82527987  4945670.55159279  3883130.78841176
  3030309.58099939  2835500.38371136  2427842.1930707   1911457.71950977
  1779447.25859033  1573182.62766051  1456963.11799481  1359633.86157442
  1155014.95786098  1056813.40248439   951043.12834829   872667.24346596
   860681.71336261   767367.56220602   743076.28065638   666788.53745266
   650917.3351369    594323.2315434    549146.13607777   522088.96079639
   503104.61461041   475584.82595126   466131.64726078   438166.47782317
   413767.387414     380024.56284485   360908.19535676   333803.59797119]
of shape: (32,)
2020-12-28 16:04:09 DESKTOP-XZDDSPC eigenface[14372] INFO Getting sorted eigenvectors:
[[-1.22082472e-02 -1.22170198e-02 -1.15940454e-02 ...  3.55293140e-03
   4.01347596e-03  5.11466732e-03]
 [-4.20471112e-03 -2.84525086e-03 -2.12906552e-03 ...  2.21606187e-02
   2.15921188e-02  1.99257795e-02]
 [ 1.20820334e-02  1.39370645e-02  9.93585603e-03 ... -2.05471490e-03
  -2.12010685e-03 -2.91506808e-04]
 ...
 [-2.58951293e-03  6.66316446e-04  3.51098408e-03 ...  2.07559640e-03
   1.43094726e-03 -3.75370247e-05]
 [ 3.16011767e-03  1.78737093e-03 -1.93208445e-03 ... -1.01141184e-02
  -7.52251863e-03 -3.59788604e-03]
 [-2.65106330e-02 -2.55080570e-02 -1.91679145e-02 ... -9.51263106e-03
  -5.91214268e-03 -4.91148062e-03]]
of shape: (32, 16384)
FaceDict: 100%|                                                                                | 45/45 [00:00<00:00, 5636.50it/s]
2020-12-28 16:04:09 DESKTOP-XZDDSPC eigenface[14372] INFO Got face dict of length 45
2020-12-28 16:04:09 DESKTOP-XZDDSPC eigenface[14372] INFO Saving model: model.MySmallDataSet128x128x1.npz
2020-12-28 16:04:09 DESKTOP-XZDDSPC eigenface[14372] INFO Model: model.MySmallDataSet128x128x1.npz saved
2020-12-28 16:04:09 DESKTOP-XZDDSPC eigenface[14372] INFO Computing eigenfaces
2020-12-28 16:04:09 DESKTOP-XZDDSPC eigenface[14372] INFO Getting eigenfaces of shape (32, 128, 128)
2020-12-28 16:04:09 DESKTOP-XZDDSPC eigenface[14372] INFO Getting mean eigenface
```

And this mean to compute that 0.95 ratio without providing the function with the actual number of eigenvectors we want, we'd wait an additional **8 minutes** for this small dataset with only 45 images and a processed size of 128x128x1

It's easy to notice that the results are the same

eigenmean.png

eigenmean32.png

eigenfaces.png

eigenfaces32.png

Training with target eigenface number = 50

```
(local38) ➜ homework3 git:(master) X python ./train.py -p "MySmallDataSet" -i .jpg -t .txt -c default.json -m model.MySm
allDataSet128x128x1x50.npz
2020-12-28 16:26:38 DESKTOP-XZDDSPC eigenface[20856] INFO Loading configuration from default.json, with content: {'width'
: 128, 'height': 128, 'left': [48, 48], 'right': [80, 48], 'isColor': False, 'nEigenFaces': 50, 'targetPercentage': None,
 'useBuiltin': False, 'useHighgui': True}
Processing batch: 45it [00:00, 59.44it/s]
2020-12-28 16:26:39 DESKTOP-XZDDSPC eigenface[20856] INFO Getting 45 names and 45 batch
2020-12-28 16:26:39 DESKTOP-XZDDSPC eigenface[20856] INFO Getting mean vectorized face: [[105.2        106.         104.7
3333333 ... 115.8        116.71111111
  115.44444444]] with shape: (1, 16384)
2020-12-28 16:26:39 DESKTOP-XZDDSPC eigenface[20856] INFO Trying to compute the covariance matrix
2020-12-28 16:26:42 DESKTOP-XZDDSPC eigenface[20856] INFO Getting covar of shape: (16384, 16384)
2020-12-28 16:26:42 DESKTOP-XZDDSPC eigenface[20856] INFO Getting covariance matrix:
[[ 5559.84545455  5422.97727273  4975.94090909 ... -1017.50454545
  -1084.89545455 -1220.18181818]
 [ 5422.97727273  5483.54545455  4942.95454545 ...  -995.36363636
  -1061.95454545 -1221.18181818]
 [ 4975.94090909  4942.95454545  5227.38181818 ...  -405.6
    -709.96515152  -997.74242424]
 ...
 [-1017.50454545  -995.36363636  -405.6        ...  8662.39090909
   8570.55454545  8123.86363636]
 [-1084.89545455 -1061.95454545  -709.96515152 ...  8570.55454545
   8832.48282828  8476.54040404]
 [-1220.18181818 -1221.18181818  -997.74242424 ...  8123.86363636
   8476.54040404  8362.11616162]]
2020-12-28 16:26:42 DESKTOP-XZDDSPC eigenface[20856] INFO Begin computing 50 eigenvalues/eigenvectors
2020-12-28 16:27:12 DESKTOP-XZDDSPC eigenface[20856] INFO Getting 50 eigenvalues and eigenvectors with shape (16384, 50)
2020-12-28 16:27:12 DESKTOP-XZDDSPC eigenface[20856] INFO Getting sorted eigenvalues:
[ 1.36920778e+07  8.06542483e+06  4.94567055e+06  3.88313079e+06
  3.03030958e+06  2.83550038e+06  2.42784219e+06  1.91145772e+06
  1.77944726e+06  1.57318263e+06  1.45696312e+06  1.35963386e+06
  1.15501496e+06  1.05681340e+06  9.51043128e+05  8.72667243e+05
  8.60681713e+05  7.67367562e+05  7.43076281e+05  6.66788537e+05
  6.50917335e+05  5.94323232e+05  5.49146136e+05  5.22088961e+05
  5.03104615e+05  4.75584826e+05  4.66131647e+05  4.38166478e+05
  4.13767387e+05  3.80024563e+05  3.60908195e+05  3.33803598e+05
  3.21100430e+05  3.10183150e+05  2.98025904e+05  2.74127350e+05
  2.48076632e+05  2.42620441e+05  2.30272108e+05  2.19841886e+05
  2.03463411e+05  1.96540753e+05  1.82455940e+05  1.71103940e+05
  5.92915720e-10  5.84871586e-10  5.50996692e-10 -5.57130828e-10
 -5.59378057e-10 -5.86506010e-10]
of shape: (50,)
2020-12-28 16:27:12 DESKTOP-XZDDSPC eigenface[20856] INFO Getting sorted eigenvectors:
[[ 0.01220825  0.01221702  0.01159405 ... -0.00355293 -0.00401348
  -0.00511467]
 [ 0.00420471  0.00284525  0.00212907 ... -0.02216062 -0.02159212
  -0.01992578]
 [-0.01208203 -0.01393706 -0.00993586 ...  0.00205471  0.00212011
   0.00029151]
 ...
 [-0.00622595 -0.00240548 -0.00289496 ...  0.00330015  0.00455527
   0.00284277]
 [ 0.00318609  0.00860076 -0.00335022 ...  0.00025306 -0.003702
   0.01103413]
 [ 0.03488897 -0.00829044  0.00933665 ...  0.00791596  0.00914027
  -0.01174183]]
of shape: (50, 16384)
FaceDict: 100%|                                                                                | 45/45 [00:00<00:00, 5012.58it/s]
2020-12-28 16:27:12 DESKTOP-XZDDSPC eigenface[20856] INFO Got face dict of length 45
```

30s for 50 eigenvalues/eigenvectors

Training with target eigenface number = 100

```
(local38) ➜ homework3 git:(master) X python ./train.py -p "MySmallDataSet" -i .jpg -t .txt -c default.json -m model.MySm
allDataSet128x128x1x100.npz
2020-12-28 16:31:36 DESKTOP-XZDDSPC eigenface[23712] INFO Loading configuration from default.json, with content: {'width'
: 128, 'height': 128, 'left': [48, 48], 'right': [80, 48], 'isColor': False, 'nEigenFaces': 100, 'targetPercentage': None
, 'useBuiltin': False, 'useHighgui': True}
Processing batch: 45it [00:00, 59.64it/s]
2020-12-28 16:31:37 DESKTOP-XZDDSPC eigenface[23712] INFO Getting 45 names and 45 batch
2020-12-28 16:31:37 DESKTOP-XZDDSPC eigenface[23712] INFO Getting mean vectorized face: [[105.2        106.         104.7
3333333 ... 115.8        116.71111111
  115.44444444]] with shape: (1, 16384)
2020-12-28 16:31:37 DESKTOP-XZDDSPC eigenface[23712] INFO Trying to compute the covariance matrix
2020-12-28 16:31:40 DESKTOP-XZDDSPC eigenface[23712] INFO Getting covar of shape: (16384, 16384)
2020-12-28 16:31:40 DESKTOP-XZDDSPC eigenface[23712] INFO Getting covariance matrix:
[[ 5559.84545455  5422.97727273  4975.94090909 ... -1017.50454545
  -1084.89545455 -1220.18181818]
 [ 5422.97727273  5483.54545455  4942.95454545 ...  -995.36363636
  -1061.95454545 -1221.18181818]
 [ 4975.94090909  4942.95454545  5227.38181818 ...  -405.6
   -709.96515152  -997.74242424]
 ...
 [-1017.50454545  -995.36363636  -405.6        ...  8662.39090909
   8570.55454545  8123.86363636]
 [-1084.89545455 -1061.95454545  -709.96515152 ...  8570.55454545
   8832.48282828  8476.54040404]
 [-1220.18181818 -1221.18181818  -997.74242424 ...  8123.86363636
   8476.54040404  8362.11616162]]
2020-12-28 16:31:40 DESKTOP-XZDDSPC eigenface[23712] INFO Begin computing 100 eigenvalues/eigenvectors
2020-12-28 16:34:15 DESKTOP-XZDDSPC eigenface[23712] INFO Getting 100 eigenvalues and eigenvectors with shape (16384, 100
)
2020-12-28 16:34:15 DESKTOP-XZDDSPC eigenface[23712] INFO Getting sorted eigenvalues:
[ 1.36920778e+07  8.06542483e+06  4.94567055e+06  3.88313079e+06
  3.03030958e+06  2.83550038e+06  2.42784219e+06  1.91145772e+06
  1.77944726e+06  1.57318263e+06  1.45696312e+06  1.35963386e+06
  1.15501496e+06  1.05681340e+06  9.51043128e+05  8.72667243e+05
  8.60681713e+05  7.67367562e+05  7.43076281e+05  6.66788537e+05
  6.50917335e+05  5.94323232e+05  5.49146136e+05  5.22088961e+05
  5.03104615e+05  4.75584826e+05  4.66131647e+05  4.38166478e+05
  4.13767387e+05  3.80024563e+05  3.60908195e+05  3.33803598e+05
  3.21100430e+05  3.10183150e+05  2.98025904e+05  2.74127350e+05
  2.48076632e+05  2.42620441e+05  2.30272108e+05  2.19841886e+05
  2.03463411e+05  1.96540753e+05  1.82455940e+05  1.71103940e+05
  6.31881861e-10  6.10329092e-10  5.84871586e-10  5.83418051e-10
  5.82367694e-10  5.74516477e-10  5.64402973e-10  5.58294802e-10
  5.53794041e-10  5.52210686e-10  5.47673919e-10  5.42134368e-10
  5.39260513e-10  5.37138540e-10  5.34487323e-10  5.32825193e-10
  5.31187904e-10  5.29370739e-10  5.29056431e-10  5.25258509e-10
  5.18375149e-10  5.10876335e-10  5.10628239e-10  5.10518172e-10
  5.09951279e-10  5.06413436e-10  5.05659206e-10  5.05565110e-10
 -5.04237028e-10 -5.04512979e-10 -5.04940374e-10 -5.05032066e-10
 -5.10600410e-10 -5.10711723e-10 -5.11709239e-10 -5.14101278e-10
 -5.15957769e-10 -5.27134817e-10 -5.27475672e-10 -5.29417688e-10
 -5.29775130e-10 -5.30181806e-10 -5.36656609e-10 -5.37412488e-10
 -5.38496639e-10 -5.40407823e-10 -5.47553819e-10 -5.49078537e-10
 -5.53804308e-10 -5.54943987e-10 -5.77271024e-10 -5.78746313e-10
 -5.86506010e-10 -5.93412114e-10 -6.08456431e-10 -6.26581087e-10]
of shape: (100,)
2020-12-28 16:34:15 DESKTOP-XZDDSPC eigenface[23712] INFO Getting sorted eigenvectors:
[[-0.01220825 -0.01221702 -0.01159405 ...  0.00355293  0.00401348
   0.00511467]
 [-0.00420471 -0.00284525 -0.00212907 ...  0.02216062  0.02159212
   0.01992578]
 [ 0.01208203  0.01393706  0.00993586 ... -0.00205471 -0.00212011
  -0.00029151]
 ...
```

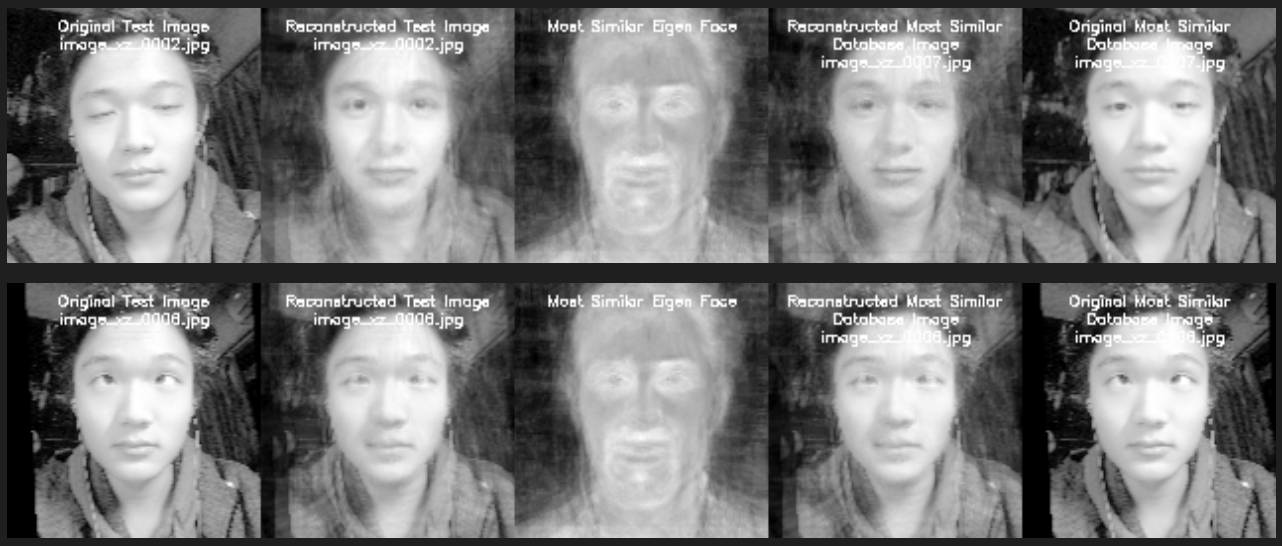
2.5 minutes for 100 eigenvalues/vectors

You'll notice even getting more usable eigenvectors is faster than computing all of the eigenvalues

Additional Testing

Then we performed the test decribed in the colored image section

32:

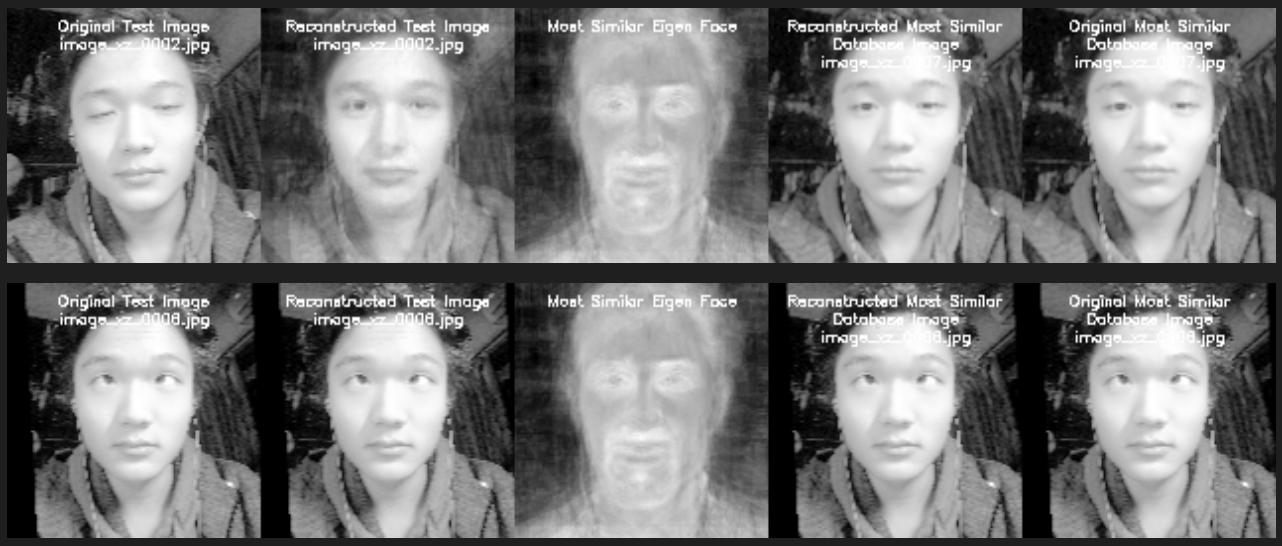

Original Test Image
image_xx_0002.jpg
Reconstructed Test Image
image_xx_0002.jpg
Most Similar Eigen Face
Reconstructed Most Similar
Database Image
image_xx_0007.jpg
Original Most Similar
Database Image
Original Test Image
image_xx_0008.jpg
Reconstructed Test Image
image_xx_0008.jpg
Most Similar Eigen Face
Reconstructed Most Similar
Database Image
Original Most Similar
Database Image

50:


Original Test Image
image_0318.jpg
Reconstructed Test Image
image_0318.jpg
Most Similar Eigen Face
Reconstructed Most Similar
Database Image
image_0309.jpg
Original Most Similar
Database Image
image_0309.jpg
Original Test Image
image_xx_0002.jpg
Reconstructed Test Image
image_xx_0002.jpg
Most Similar Eigen Face
Reconstructed Most Similar
Database Image
Original Most Similar
Database Image
Original Test Image
image_xx_0008.jpg
Reconstructed Test Image
image_xx_0008.jpg
Most Similar Eigen Face
Reconstructed Most Similar
Database Image
Original Most Similar
Database Image

100:


Original Test Image
image_0318.jpg
Reconstructed Test Image
image_0318.jpg
Most Similar Eigen Face
Reconstructed Most Similar
Database Image
image_0309.jpg
Original Most Similar
Database Image
image_0309.jpg


Original Test Image
image_xx_0002.jpg
Reconstructed Test Image
image_xx_0002.jpg
Most Similar Eigen Face
Reconstructed Most Similar
Database Image
Original Most Similar
Database Image
Original Test Image
image_xx_0008.jpg
Reconstructed Test Image
image_xx_0008.jpg
Most Similar Eigen Face
Reconstructed Most Similar
Database Image
Original Most Similar
Database Image