Intro

This repository implements EigenFaces paper. This is an algorithm that tries to recognize and detect faces using PCA.

Usage

The `src.py` file contains the class EigenFaces, whose usage is given in `demo.py` over the ATnT face dataset.
Run `demo.py` for a sample run.

Requirements

```
python 3.7.0
opencv-python 3.4.2.16
opencv-contrib-python 3.4.2.16
numpy 1.15.2
scipy 1.1.0
sklearn 0.19.2
```

Idea

This paper tries to use the idea of Principal Component Analysis to reduce the number of dimensions of the image and hopefully only measure those “Features” that vary to give different identities to people

faces.

Methodology

Training

- Suppose we have M images that are of the dimensions $N \times N$. We first convert all the images into vectors of dimention $N^2 \times 1$. Each of these vectors is denoted by Γ_i
- We next find the ‘mean-face’, i.e. the mean of all the M vectors as Ψ.
  $\Psi = \Sigma\Gamma_i$
- We find the offsets of the images from the mean face by substracting every image as,
  $\phi_i = \Gamma_i - \Psi$
- We next want to find the eigenvectors u_i of the Covariance matrix AA^T, where, $A = [\phi_1\phi_2 \cdots \phi_M]$
- Since AA^T is of the dimention $N^2 \times 1$ we don’t want to spend so much compute power to find out N^2 eigenvectors. We work around this by finding the eigenvectors v_i of $A^T A$.
- After we get all M of the v_i, we drop the all but the top K (according to thier eigenvalues) of them (PCA at work).
- We convert these v_i to u_i using the equation,
  $u_i = Av_i$
- We now normalize the u_i we got and then these u_i are exactly the ‘eigenfaces’ that the paper talks about.
- Next we find the weight vector $\in \mathbb{R}^{K \times 1}$ by the following equation,

$\mathbf{w}^j = [u_1 u_2 \cdots u_K]^T \times \phi_j$

- We store the resulting weight vectors $\mathbf{w}$ for each of the training images.

Testing

- During testing we find the $\mathbf{w}^{testing}$ of the testing image and try to minimize the L_2 from all the $\mathbf{w}$ of the training images. The $\mathbf{w}$ that is closest to $\mathbf{w}^{testing}$ set the label of the testing image, i.e. the label that the argmin $\mathbf{w}$ belonged to, is predicted the label of the testing image.
- There may be cases that we have an unknown face, in this case we set a threshold $t_{\in}$. The value of the minimum L_2 norm we got from subtracting $\mathbf{w}$ and $\mathbf{w}^{testing}$ should be less than this, else the testing image is considered to correspond to an unknownface, whose $\mathbf{w}$ we don't have.
- Other case is when we provide a non face image to the classifier, in this case we try to reconstruct the testing image $\phi^{testing}$ from $\mathbf{w}^{testing}$ by the following equation
  $\hat{\phi^{testing}} = [u_1 u_2 \cdots u_K] \times \mathbf{w}^{testing}$
- We want this reconstructed image to be withing t_f, threshold from the originial image $\phi^{testing}$, i.e.
  $L_2(\hat{\phi^{testing}} - \phi^{testing}) < t_f$
  To be considered a face image, else a non face image.

Testing

- We tested the class that we implemented by only letting the

algorithm learn 38 subject images in the dataset above. We then tested the algorithm on a collection of all the 40 subjects in the dataset.

- We set the values of the threshold by looking at the sample values we were getting of the L2 norms of the different differences in weights.
- To test images that are non-faces, we used a random numpy array generated by `np.random.rand` to test for non-face detection by our algorithm.
- We chose the values of the hyperparameters that resulted in a good accuracy over all the accuracy measures, one can obviously change the values of the hyperparameters to get a different distribution of the accuracies according to a particular use case.
- Also we took the `train_test_split` ratio of `80-20`.

Results

We used the ATnT face data set to train our algorithm and then we tested the implementation using some known faces, some unknown faces, and some non-faces, in each of the case we received accuracies of `Accuracy_knownFaces: 0.7157894736842105`, `Accuracy_unknownFaces: 0.8` and `Accuracy_nonFaces: 1.0`. The hyperparameters that we worked with were as below. One can change the hyperparameter at the start of demo.py to get different results.

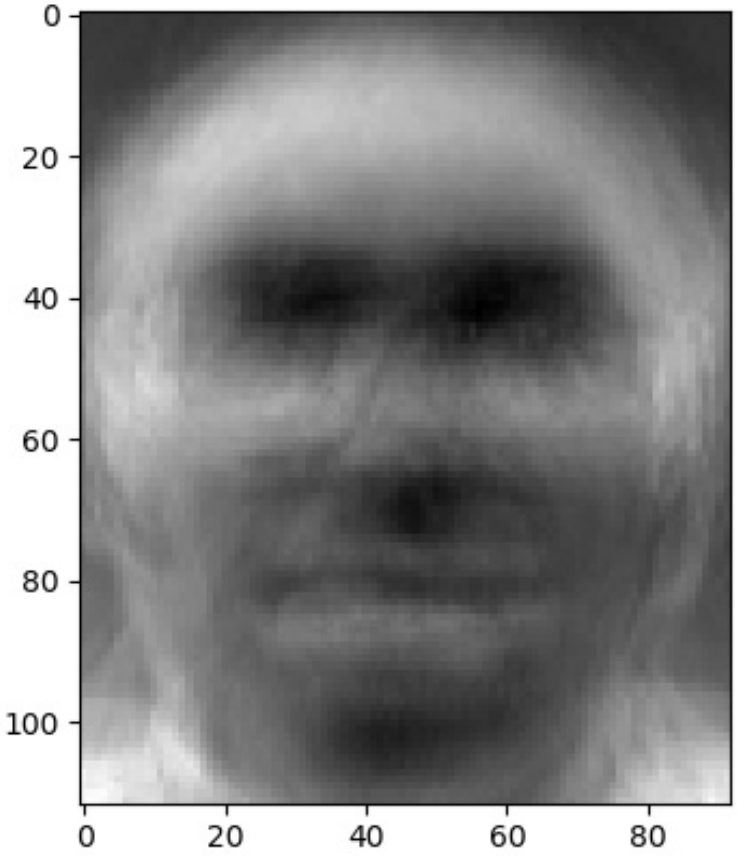
We print the accuracies to the console and the main class that has been implemented resides in `src.py`. Have a look at that file to know the

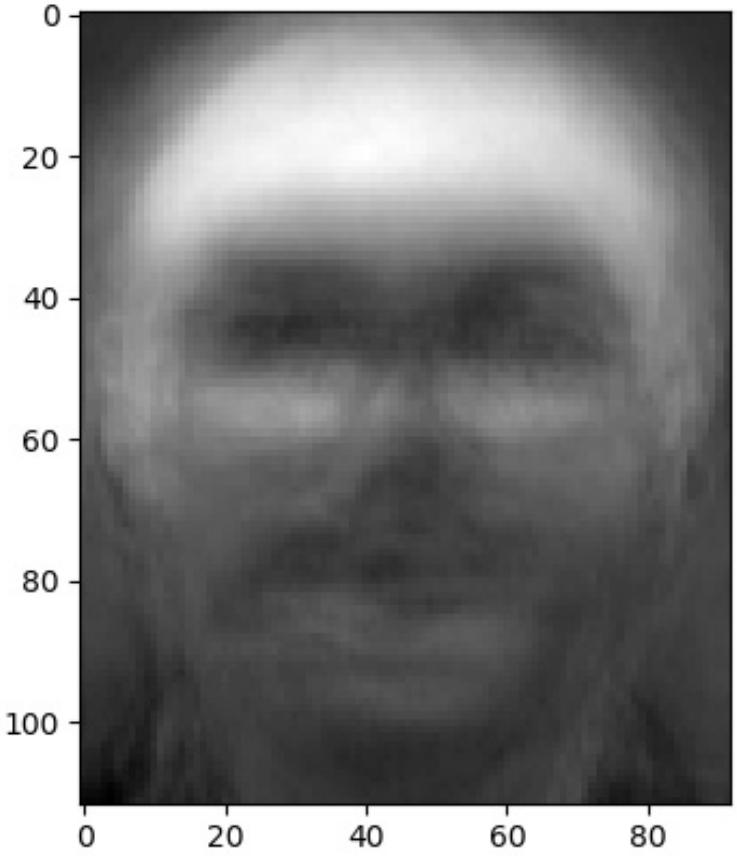
implementation details.

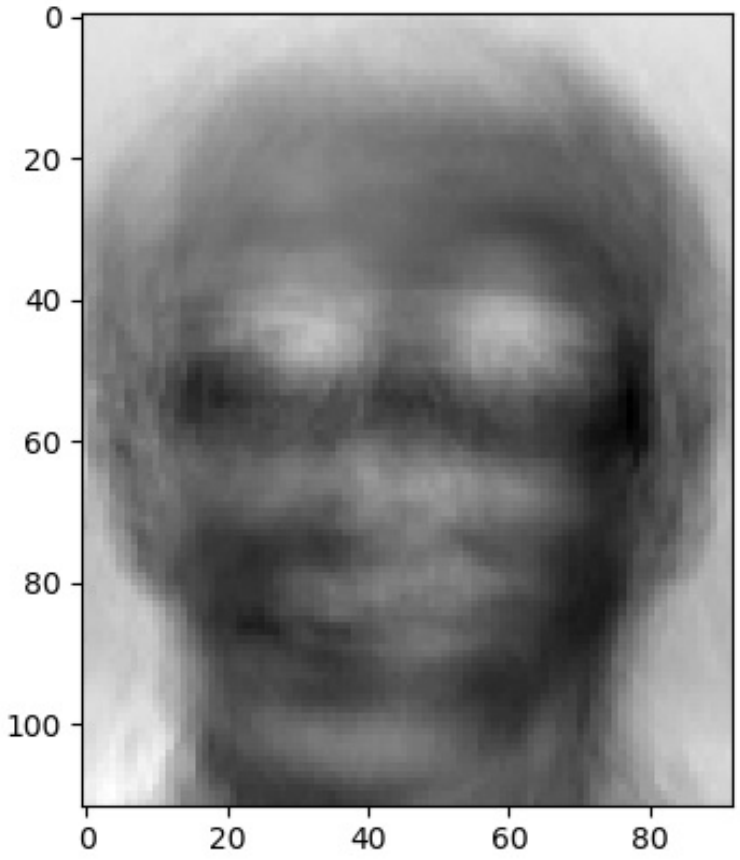
Hyperparameters used:

```
K = 30 # dimension of face_space
b = 2 # number of classes to keep unseen
te = 2100 # Threshold for the L2 distance for training wei
ght vectors
tf = 13000 # Threshold for the L2 distance from the face s
pace
unknownface = -1 # label to denote an unknownface
nonface = -2 # label to denote an nonface
```

Top 3 eigenfaces:


0
20
40
60
80
100
0
20
40
60
80


0
20
40
60
80
100
0
20
40
60
80



Conclusions

- Setting the hyperparameter of threshold `te` to a big value ensures that we get good recognition when the faces are known but keeping it low will increase the accuracy of detecting an unknown face at the cost of accuracy of recognizing a known face.
- Keeping `K` bigger ensures better recognition, at the cost of detecting an unknown face.
- One needs to tune the values of `te`, `tf`, `K` according to a particular use case. For example, taking `K=120` and `te = 3500` in our dataset testing results in `Accuracy_knownFaces:`

`0.9263157894736842`, `Accuracy_unknownFaces: 0.0` and `Accuracy_nonFaces: 1.0`.

Reference

- F. Samaria and A. Harter

  “Parameterisation of a stochastic model for human face identification”

  2nd IEEE Workshop on Applications of Computer Vision

  December 1994, Sarasota (Florida).
- http://www.vision.jhu.edu/teaching/vision08/Handouts/case_study_
- http://www.face-rec.org/algorithms/pca/jcn.pdf
