CS131 Computer Vision: Foundations and Applications (Fall 2015)
Programming Assignment 3: Face Recognition

Due December 4th, 2015

1 Introduction

In this assignment, you'll train a computer to recognize Sarah Connor, the protagonist of the 1984 movie *The Terminator*.

SCHWARZENEGGER
THE TERMINATOR
Hemdale Presents A Pacific Western Production of a James Cameron Film
Arnold Schwarzenegger "The Terminator" Michael Biehn, Linda Hamilton and Paul Winfield
Make-up Effects By Stan Winston - Director of Photography Adam Greenberg
Executive Producers John Daly and Derek Gibson - Written by James Cameron with Gale Anne Hurd
ORION Produced by Gale Anne Hurd - Directed by James Cameron

Figure 1: Your End User

We have provided you with a training set that includes labeled images of Sarah Connor and other humans' faces under multiple lighting conditions and with multiple facial expressions, but with the faces perfectly aligned. In other words, the subject's eyes are always at the same pixel locations, so that faces can be compared to each other easily.

These faces were identified and aligned automatically, via the Viola-Jones algorithm. We won't go over that algorithm in this class, but it's basically a way to effectively do a series of cross-correlations to locate face features such as eyes.

1. We'll begin with a simple pixel-by-pixel method of face comparison. In *compareFaces.m*, we have provided code that reads in all the labeled training images as a single matrix. Each column of the matrix is an image which has been unrolled column-wise (first column, second column, etc) to form one long column vector. You will complete the "Part 1" sections of code to compare a provided test image to each of these column vectors. If the most-similar column vector is an image of Sarah Connor, we will decide that the test image is Sarah Connor.

   Edit the main *compareFaces()* function to take the given test image and unroll it into a vector. Edit the *indexOfClosestColumn()* function to perform a Euclidean distance comparison to each column from the training data. Finally, edit the main *compareFaces()* function to return the label of the training example which was the closest match to the test face. Read and run *compareFacesTest.m* with the first few test faces to confirm that this simple method works.

2. What you did above is called a "nearest-neighbor" algorithm. Note that you didn't use the training data to build a computer model of Sarah Connor. Instead, you just kept all the training data, compared the new image to every training example, and gave it the label of the closest one. **In your writeup**: what are the advantages and drawbacks of the nearest-neighbor approach to classification, in general? (Be sure to discuss computational difficulties if you have a very large databases of faces.)

2 Eigenfaces

The "Eigenfaces" algorithm is a more efficient version of what you did above. It is still a nearest-neighbor algorithm, but it uses Principal Component Analysis (PCA) to represent each face with just a few numbers. You may want to review our Linear Algebra slides on PCA before continuing. You will complete the "Part 2" code in compareFaces.m and doPCA.m to perform the same comparisons as above, but using the PCA form of the face vectors.

1. We will use the same matrix of training examples as before, and we will reuse the code which finds the mean column of this matrix and subtracts it from every column. This will prevent PCA from representing patterns that are the same in every image.

2. In *doPCA.m*, use the SVD command to perform PCA, and retain the top *numComponentsToKeep* principal components (also known as the "basis vectors.") You should use the *SVD(A,'econ')* form of the command, as it is much more efficient when we only need the top few components, as we do here. Also, retain the matrix of component weights:

$$W = \Sigma_{short} V_{short}^T \tag{1}$$

   where "short" indicates a matrix which only contains entries that affect the top *numComponentsToKeep* principal components. (E.g. Σ_{short} is size *numComponentsToKeep* x *numComponentsToKeep*.)

3. **In your writeup**: What do the weights represent? (Be specific about what each entry in W represents.) How could you use the weights to reproduce the faces that they represent? (You may use MATLAB to verify your answer if desired – the provided *viewFace* function will reshape an unrolled face column back into an image and display it.)

4. In *compareFaces.m*, call your *doPCA()* function to produce basis vectors and weights for the training images. Then, convert your column-vector test image to "PCA space." In other words, convert it to a vector of weights on the PCA principal components which could be used to reproduce the original image. *Hint*: the PCA basis vectors are unit vectors. The dot product of a vector x with a unit vector gives the component of x which lies is in the direction of that unit vector.

5. Use your *indexOfClosestColumn()* function from before to compare the weight vector of the test image (i.e. its representation as a "PCA space" vector) to the PCA space vectors of each training example, to choose the nearest neighbor. Use *compareFacesTest.m* to verify that your algorithm works as before. **In your writeup**: If we wanted to build a mobile robot which can recognize Sarah Connor, what are the advantages of PCA-space representation of face images? Consider the effects on storage and on computation.

3 Fisherfaces

Above, PCA found the basis vectors which were best to reconstruct the faces. (That is, they capture the most possible variance across all images.) However, our purpose here is not to reconstruct the faces from their PCA-space representations, it's to compare their PCA-space representations. The Fisherfaces algorithm chooses basis vectors which are more optimal for our comparison task. Given images and a class label for each image, it will choose basis vectors which capture the most between-class variance, and the least within-class variance. Thus, the resulting weight vectors will be better for distinguishing between classes.

1. In *compareFacesTest.m*, uncomment the "Sarah Connor, higher brightness" test image. Run it with the Eigenfaces comparison method and again with the simple pixel-by-pixel comparison method, and observe that both fail.

2. Complete the "Part3" code in *compareFaces.m* to call the provided *fisherfaces()* function in *fisherfaces.m.* Use the basis vectors and weights which it produces to do the comparison instead. It should successfully match the "Sarah Connor, higher brightness" test image.

3. **In your writeup**: Explain why Fisherfaces worked better in this specific case. It may help to look at the training images and test images.

4 Identification of humans

In the intro, we mentioned that fast algorithms exist to locate faces in an image. But these algorithms occasionally make a mistake, and grab an image region which is not a face. If we are given such a non-face region, we would like to reject it, rather than blindly try to match it to a face.

Design your own method to decide if a given image is a human face or not. The function in *isFace.m* will be given an image, and it must return true if it is a human face, and false if it is not. (If the image is a human face, you can assume that it will be aligned like the faces in the training set. Also, all images will be the same size as the ones in the training set.) You may reuse any code from this homework or previous ones. You can use *isFaceTest.m* to test your method.

Some tips to get you started:

1. You have a lot of useful code in *compareFaces.m*: the basis vectors, the mean face, etc. You can also use other things like MATLAB's edge function.

2. If you wish, your code may use the provided faces in the "facedatabase" and the provided non-face images in "nonfacedatabase".

3. If you make use of basis vectors, be aware that the top few PCA vectors tend to capture general brightness patterns, rather than more face-specific patterns. So if you use basis vectors, you may want to experiment with discarding the first 3 to 5 basis vectors.

4. Note that the images are grayscale, so the color-segmentation technique of problem 5, below, will not be useful here.

Grading: For full credit, your method must achieve at least 65% accuracy on the *isFaceTest.m* test set. (Your code may not use the images in the *facetestset* directory, and it must not do any form of memorization or hard-coding of the right answers.) We'll also run your code on a hidden test set, and the top 3 classifiers will receive extra credit.

In your writeup: Make sure to explain in detail exactly how you implemented your function. What design decisions did you make while implementing this function? Also report your accuracy score on the *facetestset*.

5 Erosion, Dilation, and Blobs

Erosion and dilation are useful pixel-level image processing techniques. When applied to a binary image (composed of 1's and 0's), dilation will expand all '1' areas, and erosion will shrink all '1' areas.

Mathematically, it works like this: to perform a dilation, a structuring element (such as a circle) is applied to an image at all possible offsets, just like a linear filter. But unlike a linear filter, the output is the max of all points under the structuring element. So, if you use a circle-shaped structuring element with radius R, any point which is within R of a '1' will turn into a '1'. Erosion works the same way, but it takes the min of all points under the structuring element.

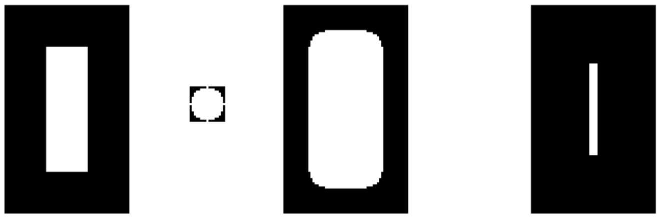

Figure 2: From left to right: The original binary image; a structuring element; dilation; erosion

In *findHeads.m*, we apply color segmentation to identify skin-colored pixels in a given photograph. The result is a binary image, where '1' shows pixels that are skin-colored.

We would like to use this to find the heads of humans in the image. Heads tend to be roundish and skin-colored. MATLAB has an ability to identify "blobs", i.e. connected regions of '1's. But the '1' pixels for each head may not be connected due to noise. We can use dilation to connect them. Erosion might also be helpful for eliminating spurious '1' pixels.

Read the comments in *findHeads.m*, and use MATLAB's *imdilate*, *imerode*, and *regionprops* functions to produce blobs for the heads and then find the centers of blobs which are head-like. You can run the function with no arguments to test it.

For full credit, your code should accurately find the three heads in *heads.jpg*, without labeling any non-head regions as a head.

(And, obviously, your code should not have the right answers hard-coded. E.g. if we give it the same image with the humans translated or added/removed, it should still work. It does not need to work reliably for completely different images.)

In your writeup: How robust is your head-finding method? Discuss the effects of the scale of the heads, rotation of heads, and image features that could cause false positives.

6 Writeup

In your writeup, answer all writeup questions from above: 1.2), 2.3), 2.5), 3.3), 4) and 5). Also answer these questions:

1. Why did we have to align the face images? Discuss the effect on the PCA basis vectors if the labeled training faces were not aligned.
2. How would the nearest-neighbor algorithm behave if you had a very small database of labeled faces? (For example, suppose your database only had images of Sarah Connor and one other human.) Explain your answer.
3. Describe how your *isFace()* algorithm works and why it responds to faces.

Make sure to include all the code you write in the writeup as well. Also, include in the writeup any images that you generate in this assignment.

7 References

1. Joao P. Hespanha Peter N. Belhumeur and David J. Kriegman. Eigenfaces vs. fisherfaces: Recog- nition using class specific linear projection. http://www.cs.columbia.edu/ belhumeur/journal/ fisherface-pami97.pdf, 1997.
2. L. Sirovich and M. Kirby. Low-dimensional procedure for the characterization of human faces. http: //goo.gl/xdb2Ln, 1987.