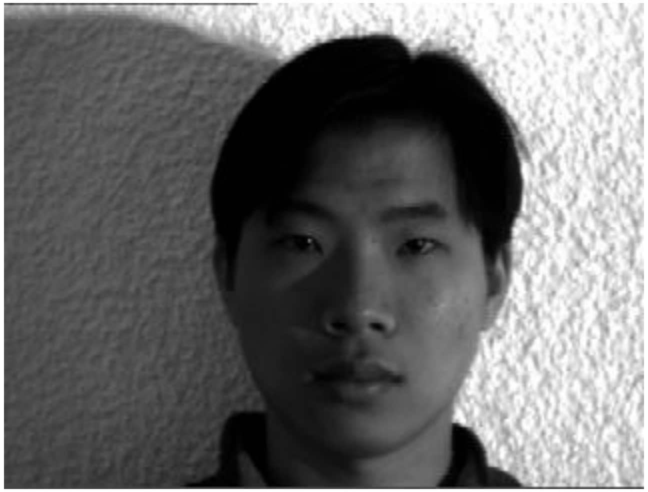
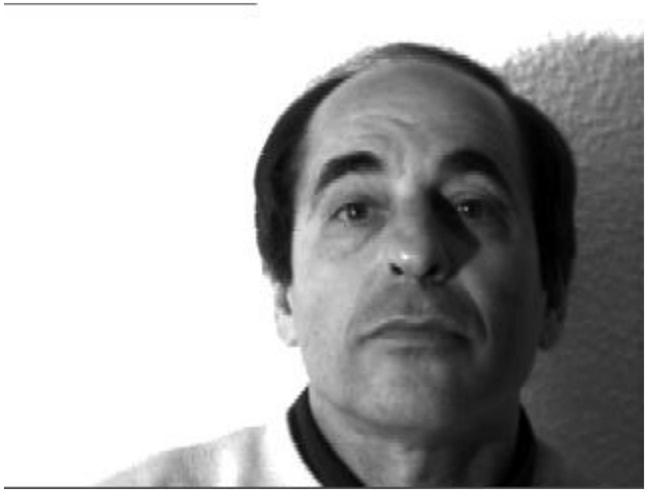
PA3: Face Recognition

Outline

- Preprocessing faces
- Nearest-neighbor on:
  - whole images
  - PCA of faces ("Eigenface" representation)
  - LDA of faces ("Fisherface" representation)
- Bonus: dilation/erosion

Raw data: problems?

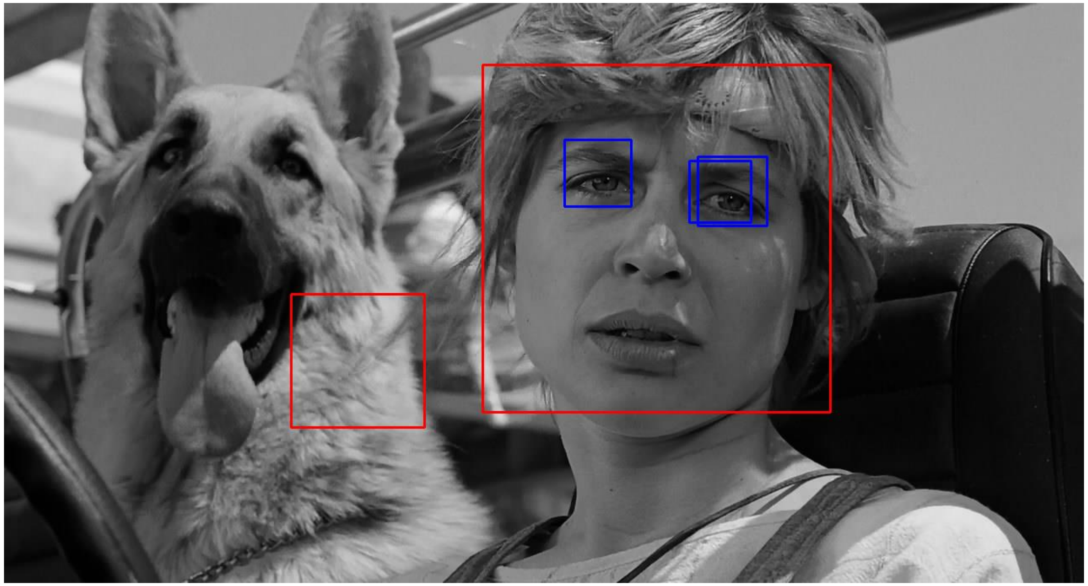


Raw data

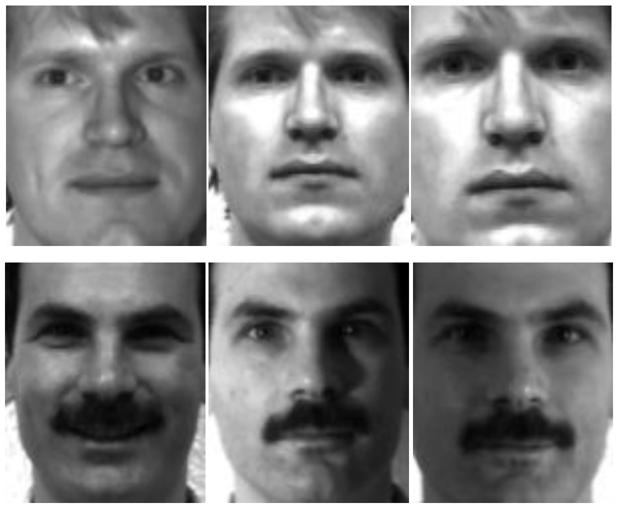
- If we plan to do a simple pixel-by-pixel comparison (and we do), then the faces must be in the exact same position in each image
  - So we compare eye pixels to eye pixels, nose pixels to nose pixels, etc.
- Computers can do this, using the Viola-Jones (a.k.a. Haar Cascade) face-detection algorithm

Viola-Jones algorithm

- We don't cover it in this class, but Viola-Jones face detection basically uses a bunch of linear filters, which were arrived at through machine learning, to detect faces, eyes, or whatever object it's trained on
- Great for detecting faces and other very consistent-looking objects
- We have applied it for you, to cut out and rotate/scale faces



Preprocessed Data

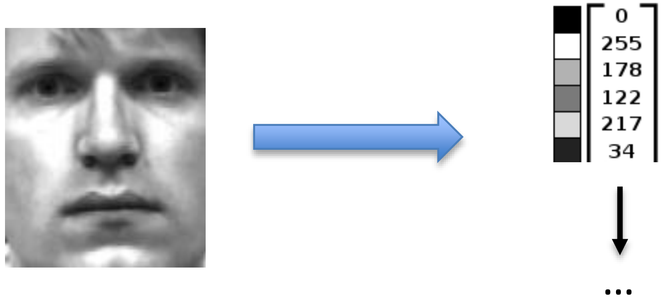


…

- We give you a big database, with multiple faces per test subject
- Faces are well-aligned
- You will compare new faces to this database, and label them as belonging to the closest test subject (K-NN with K=1)

Comparing faces

- Simplest method: “unroll” each grayscale face image, columnwise, into a single long vector



- Compare faces by taking Euclidean distance between new face-vector and each one in the database
- You’ll do this in `compareFaces.m`

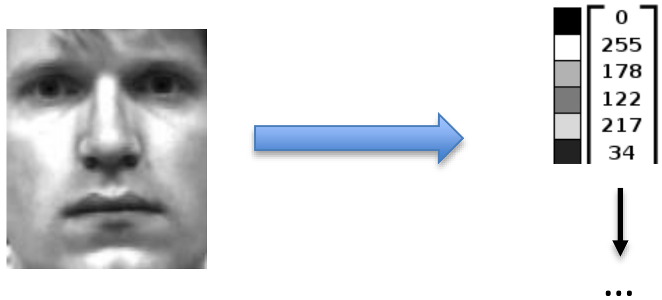
Format of provided database

```
% load our face database into a matrix.
[rawFaceMatrix, imageOwner, imgHeight, imgWidth] = readInFaces();
% This give us: faceMatrix - column 1 of this matrix is image 1,
%       converted to grayscale, and unrolled columnwise into a vector.
%       So if image 1 is 120x100, column 1 will be length 12000. Column
%       2 is the same for image 2.
% imageOwner - a vector of size 1 x numImages, where imageOwner(i)
%       holds the integer label of image (i). Images from the same
%       person have the same label.
% imgHeight - the height of an original image (they are all the same
%       size)
% imgWidth - the width of an original image (they are all the same
%       size)
```

- Database faces are unrolled for you
- You unroll test images yourself, with

```
testImgVector = testImg(:)
```

Comparing faces



- Even a small image size of 120x100 pixels produces a vector with 12,000 numbers
  - If we do lots of comparisons, it will get slow
  - Not great for storage space either
- Do we truly need 12,000 separate numbers to compare faces? **NO!**

PCA for lean representation

- Principal Component Analysis is a technique to reduce the dimensionality of data
- Key insight is that most types of raw data (e.g. faces) can be represented as a combination of simple patterns
- PCA finds a set of patterns that can be linearly combined to reproduce the data:
  - e.g. **faceImage1** = 2*****pattern1** - 0.5*****pattern3**
- We store the patterns once, and then we can represent each face just in terms of its weights on the patterns (e.g. 2 and -.5, in the example above)

PCA review: getting PCA from SVD

$$\overset{U\Sigma}{\begin{bmatrix} -3.67 & -.71 & 0 \\ -8.8 & .30 & 0 \end{bmatrix}} \times \overset{V^T}{\begin{bmatrix} -.42 & -.57 & -.70 \\ .81 & .11 & -.58 \\ .41 & -.82 & .41 \end{bmatrix}} \qquad \overset{A_{partial}}{\begin{bmatrix} 1.6 & 2.1 & 2.6 \\ 3.8 & 5.0 & 6.2 \end{bmatrix}}$$

- Construct a matrix where each column is a separate data sample (e.g. each column is a face vector)
- Run SVD on that matrix, and look at the first few columns of ***U*** to see patterns that are common among the columns
- Columns of ***U*** are called *Principal Components* of the data samples.
- (Note: the above image combines U and Σ. We'll actually combine Σ and V^T, so that our principal components are the columns of U, and are unit vectors.)

PCA review: getting PCA from SVD

$$\overset{U\Sigma}{\begin{bmatrix} -3.67 & -.71 & 0 \\ -8.8 & .30 & 0 \end{bmatrix}} \times \overset{V^T}{\begin{bmatrix} -.42 & -.57 & -.70 \\ .81 & .11 & -.58 \\ .41 & -.82 & .41 \end{bmatrix}} \qquad \overset{A_{partial}}{\begin{bmatrix} 1.6 & 2.1 & 2.6 \\ 3.8 & 5.0 & 6.2 \end{bmatrix}}$$

- Often, raw data samples have a lot of redundancy and patterns
- PCA can allow you to represent data samples as weights on the principal components, rather than using the original raw form of the data
- By representing each sample as just those weights, you can represent just the “meat” of what’s different between samples.
- This minimal representation makes machine learning and other algorithms much more efficient

PCA for lean representation

- The PCA principal components are also known as “basis vectors” that can be linearly combined, with some weighting, to produce each face vector.
- The weights for the training faces can be read off from V^T
- When we see a new face, we can easily get its weights:
  - PCA basis vectors are unit vectors and are orthogonal (mutually perpendicular)
  - So, dot product of a PCA basis vector with a face produces the weight on that vector
- Before we do PCA to get the patterns, we calculate a “mean face” and subtract it from all samples. (There’s no benefit to representing patterns that are identical for all faces)
  - So, remember to also subtract that mean face from the test sample.

PCA for lean representation

- PCA basis vectors are column vectors. But we can roll them up into an image and view them to see what patterns they're representing:

PCA for lean representation

- We can now represent images as weights on PCA basis vectors (the vector of weights for an image is sometimes called its "PCA space" representation)
- Those components represent most of the variation between images
  - So, distance measurements in PCA space are just as good!

- If we use weights on the top 20 principal components to represent images of size 120x100, we have compressed to 0.17% of the original size
  - We do need to store those top 20 principal component vectors for the dataset, but the savings is still massive for large datasets!

Fisherfaces

- PCA compresses data, which is great
  - Its basis vectors capture the most variance possible
- But what if we could get basis vectors that actually help us with our task? They would:
  - Include variations in data that are important to distinguish faces
  - Intentionally leave out variations that are not helpful, such as lighting changes
- Fisher Linear Discriminant Analysis (a.k.a. Fisherfaces) can do that

Fisherfaces

- Fisherfaces needs a training set that includes multiple examples (face images) for each class (test subject)
  - Each examples is labeled with its class
  - Fisherfaces finds basis vectors that capture the most variation between classes, and the **least** variation within classes
  - If your training data includes multiple lighting situations, it will tend to produce vectors that ignore lighting changes

Fisherfaces

- We have implemented Fisherfaces for you, and you'll just experiment with it.
  - You'll need to know what it does, but not the math behind it

Fisherface basis vectors

Eigenface basis vectors

Design problem: classification as face/nonface

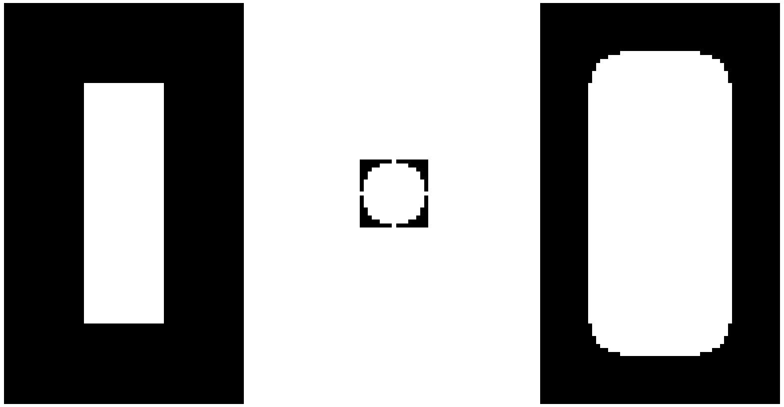
- You will code `isFace.m`, which decides if a given image is a face
- Many possible methods
- Good approaches involve checking for face-like patterns. Some options:
  - How much of the image is represented by the basis vectors (which we know are good at representing faces)
  - How similar to "mean face"?
  - Other options too (faces tend to have edges in certain locations, etc.)

Erosion/Dilation

- Erosion and dilation are a pixel-level filtering technique
- Slide a “structuring element” across an image (just like a linear filter)

Dilation

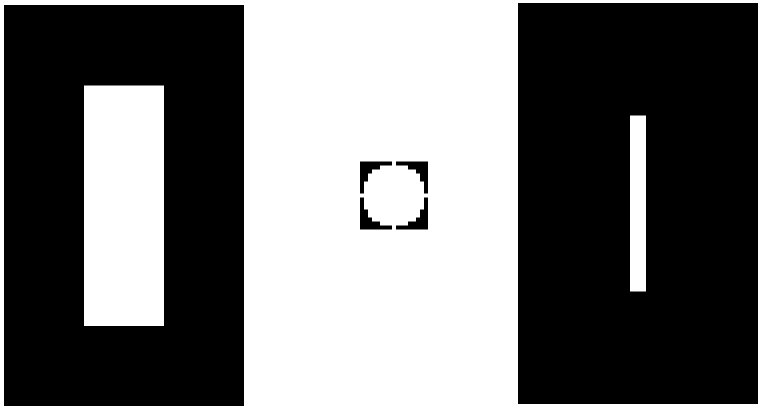
- In dilation, the pixel at the center of the structuring element is replaced with the **max** of everything under the structuring element



- Typically use a circular structuring element, as above, but other shapes can have other effects

Erosion

- In erosion, the pixel at the center of the structuring element is replaced with the **min** of everything under the structuring element

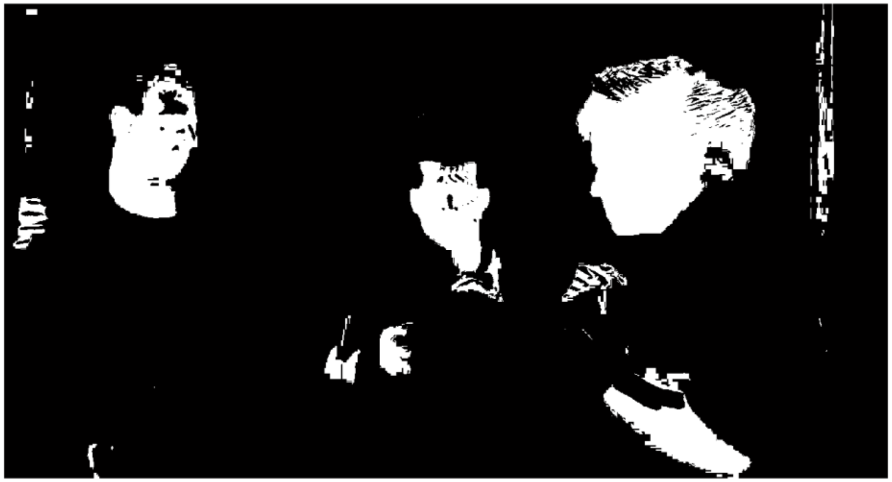


- In binary images, erosion shrinks blobs and dilation grows blobs.
- They can be used to get clustering-type effects

Design problem: cleaning up skin segmentation

- In `findHeads.m`, we give you code which makes a binary image, where 1 means the pixel is close to skin color
- With dilation/erosion, you can get round blobs (connected regions of 1's) where there are heads
  - Will require a lot of tweaking while looking at results
- Then, MATLAB's `regionprops` function can give you the center, area, eccentricity, and other characteristics for a blob
- You must return the centers of all heads

Design problem: cleaning up skin segmentation



Writeup

- Answer the given questions about how and why things work
- Our grading process is:
  - We answer the questions ourselves and come up with important “bullet-points” that a complete answer contains
  - Grade for a question is based on whether you include the important points
    - No need to tell us other stuff, or repeat info we’ve given you, unless you want to