Homework 7

Sunday, October 25, 2020 7:58 PM

document

Chapter 20

Homework 7: PCA and Eigenfaces

Contents

Learning Objectives

Concepts

- Understand the connection between PCA and eigenvalues and eigenvectors.
- Understand how to use PCA to carry out data compression.
- Understand the idea of using Eigenfaces to do facial recognition.

MATLAB skills

- Use "eig" to carry out a PCA.
- Implement facial recognition using PCA.
- Determine the accuracy of PCA for different numbers of principal components.

20.1 Principal Component Analysis Revisited

As we in saw class, PCA is an algorithm in which we express our original data as a linear combination of the eigenvectors corresponding to the largest eigenvalues of the covariance matrix. We examined the property of PCA that it if we project our data onto these vectors, this will lead to maximizing the variance of the projected data. To refresh your memory further, here is the temperature plot for Boston, Sao Paolo, and Washington DC.

PCA = Convert data to combo of eigenvectors to maximize variance
↳ Dimensionality Reduction

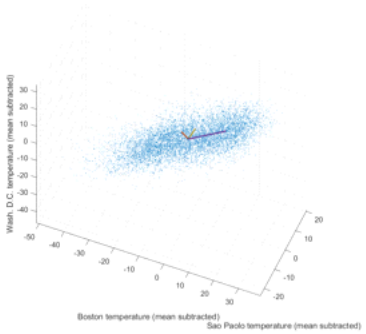


Figure 20.1: Temperatures in three cities and the eigenvectors of the covariance matrix.

We also briefly mentioned a second property of PCA, which is that it can be thought of as an optimal way to compress our data down to a smaller set of numbers. This idea, also known as dimensionality reduction, is going to be a view that we explore in this assignment. If you'd like, here are a few external resources on PCA:

- http://www.cs.otago.ac.nz/
- http://dai.fmph.uniba.sk/courses/ml/sl/PCA.pdf
- https://deeplearning4j.org/eigenvector/linear
- http://www.cerebralmastication.com/2010/09/principal-component-analysis-pca-vs-ordinary-least-squares-ols-a-visual-explination/
- http://stats.stackexchange.com/questions/2691/making-sense-of-principal-component-analysis-eigenvectors-eigenvalues/

PCA in two dimensions

In general, PCA is conducted on data that is mean-centered (i.e., the data has had the mean of each variable subtracted out). To refresh your memory of PCA and scaffold the introduction of the view of PCA as compressing a dataset, let's think about a simple example data set $\mathbf{D}$.

$$\mathbf{D} = \begin{bmatrix} -1 & 3 \\ 1 & 4 \\ 3 & 4 \\ 7 & 5 \\ 10 & 9 \end{bmatrix} \tag{20.1}$$

Exercise 20.1

1. Create a plot of $\mathbf{D}$ as a set of points in the xy-plane.
2. Define a matrix $\tilde{\mathbf{D}}$ which is the mean-centered version of $\mathbf{D}$ and plot $\tilde{\mathbf{D}}$ as a set of points in the xy-plane
3. The principal components ($\mathbf{p_1}$ and $\mathbf{p_2}$) are the eigenvectors of the covariance matrix of the

mean-centered $\tilde{\mathbf{D}}$. Compute $\mathbf{p}_1$ and $\mathbf{p}_2$ and plot them on top of the mean-centered data.

4. Compute the projection of your data onto the eigenvector which corresponds to the largest eignevalue, which in this case is $\mathbf{p}_2$. This is the "reduced dimenstionality" version of your data, called $\mathbf{B}$, which only include information about the projection along $\mathbf{p}_2$. (We reduced the 2-dimensional data to 1-dimensional data.) Plot the original data $\mathbf{D}$ and the reduced data $\mathbf{B}$.

Exercise 20.2

1. Can you recreate $\mathbf{D}$ perfectly from $\mathbf{B}$? NO, it's an approximation
2. What would have happened if you had created $\mathbf{B}$ using only information about the values along $\mathbf{p}_1$ instead of $\mathbf{p}_2$? All of the points would be very clustered around the origin, very inaccurately
3. How might you quantify how well you can represent $\mathbf{D}$ in this reduced dimensionality form? some sort of error calculation, maybe RMS
4. If you received a new piece of data, how would you go about representing this as a linear combination of $\mathbf{p}_1$ and $\mathbf{p}_2$? $\begin{bmatrix} coeff_1 \\ coeff_2 \end{bmatrix} = \begin{bmatrix} | & | \\ p_1 & p_2 \\ | & | \end{bmatrix} \backslash \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$

Data Compression via PCA

In this exercise you will perform a simple data compression exercise, similar to the one you did in a previous homework assignment. You will use temperature data from 3 cities over 10 years, as training data and use it to compress a year's worth of temperature data from 3 cities into a 2×365 matrix. In other words, you will represent 3×365 numbers (daily temperature data from 3 cities over 1 year), using 2×365 values. This compression is lossy, in that you will loose some information. However, by representing the data along the two most significant eigenvectors of the covariance matrix, you can reduce this data loss, because these two directions capture the bulk of the variation in the data set. Please note that while we have laid out the steps you need to take here quite explicitly, it is important for you to fully understand what each step does. You will be using very similar steps in your project.

Exercise 20.3

1. Load the file `avg_temperatures_pt2.mat`. You will have 6 data vectors in your workspace. `b_tr, w_tr, s_tr` which represent 10 years of training data for the average daily temperatures in Boston, Washington DC, and Sao Paolo, respectively. The vectors `b_new, w_new, s_new` represent an additional year of data for the three cities – this is the data that you will compress using statistical knowledge of the previous 10 years of data. Create a covariance matrix $\mathbf{R}$ using the 10 years worth of temperature data from Boston, Washington DC and Sao Paolo (in that order).
2. Perform an eigendecomposition of the matrix $\mathbf{R}$, and make a new matrix $\mathbf{V}_p$ which has the 2 eigenvectors corresponding to the 2 largest eigenvalues of $\mathbf{R}$. You should use MATLAB's `eig` function. Let these eigenvectors be $\mathbf{v}_1$ and $\mathbf{v}_2$.
3. Create centered (i.e. subtract the mean), versions of the new temperature data vectors, and

create a 3×365 matrix $\mathbf{T}$ which has the centered temperatures of Boston, Washington DC and Sao Paolo as its rows (in that order). This matrix is a representation of the data you are now going to compress. Let the i-th column of $\mathbf{T}$ be $\mathbf{t}_i$.

4. Take the dot product of each column of the matrix $\mathbf{T}$ (which is a vector of the temperature of Boston, Washington DC and Sao Paolo for a given day) with the two eigenvectors in matrix $\mathbf{V}_p$, and save the values. Let this quantities be called α_{1i} and α_{2i}. In other words,

$$\alpha_{1i} = \mathbf{v}_1^T \mathbf{t}_i$$
$$\alpha_{2i} = \mathbf{v}_2^T \mathbf{t}_i$$

   You can do this using matrix multiplications.

   You should now have 365 different values for α_{1i} and α_{2i}, which are a compressed representation of 3×365 different temperature values. Moreover, these values are the components of the temperature data that lie in the directions of the two eigenvectors of the covariance matrix corresponding to the largest eigenvalues. From what we saw in the previous two classes, these vectors represent the two orthogonal directions in the data that have the most amount of variation, and hence the most "important" directions. Of course, there is a third direction (since the temperature vectors live in a 3-dimensional space), which we are discarding. But since this is the direction in which there is the least amount of variation in the data set, we do not lose too much information.

5. You can now check how well your compression worked, by using the values of α_{1i} and α_{2i} to reconstruct 365 different 3×1 vectors each representing the temperatures for the three cities over the 365 days. Let $\hat{\mathbf{t}}_i$ represent the reconstructed temperature vector on the i-th day. Using what you know about projections onto orthonormal vectors, reconstruct $\mathbf{t}_i$ using α_{1i}, α_{2i}, $\mathbf{v}_1$ and $\mathbf{v}_2$. Repeat this for all 365 days.

6. On the same axes, plot the original and reconstructed temperature for Boston. Repeat this for Washington DC and Sao Paolo. Observe how close the reconstructions are, for the different data sets.

7. How accurately do you think you can represent the data if you used 3 eigenvectors instead of 2? 100% Accurately

8. If you feel inspired, repeat the above with temperature data for four different cities, and 2 or 3 different eigenvectors.

While this example can be thought of as a "toy" example where we are representing 3 dimensional data using 2 dimensions, there are many applications for which there may be many more dimensions in the data for which accurate representations can be made using only a few dimensions. Additionally, you should note that such dimensionality reduction techniques are not just useful in compression, but they are also useful in speeding up computation. We can often get away with analyzing data over a small number of important dimensions, and this is an important technique when we deal with large amounts of data. Overall, these class of techniques is called Principal Component Analysis (PCA), since we are performing analysis along a few principal component directions of the data.

20.2 Face Data Compression via PCA

You are now ready to start applying PCA to face data. You have already seen this in a previous class assignment, except in that assignment you had the help of a genie. Load the MATLAB files `classdata_train.mat`

and `classdata_test.mat`. These are the training and test datasets with photos of your classmates. The file contains some images and as well as the identity of the person in each image (coded as an integer from 1 to 89).

Remember that the principal eigenvectors of the covariance matrix tell you the directions of greatest variation in a data set and also the directions that optimally compress our data. Your job is to use the training images to build a model for your faces such that you can compress a many-pixeled test face image (pick one from the test image array) using a small set of image vectors (e.g., 10, 20, or 50).

Before you dig into this problem, think through how you would formalize the face data compression as a problem that you can solve with the linear algebra techniques that you've learned so far. There is no exercise to answer, but we want you to think through these steps before going further in the assignment.

- How you will choose your set of image vectors?
- Come up with the steps needed to do the above and write some pseudo code (e.g., load the data, vectorize the images, etc.).
- How could you tell whether your compression algorithm works (these could either be quantitative metrics, like root-mean squared error or qualitative metrics).

20.3 Eigenfaces for Face Recognition

It's time to bring it all together and finally plunge into the prosopagnosia (look it up) problem (or at least build some facial recognition software, but alliteration is fun). You will implement the eigenfaces algorithm to identify photos of your classmates. While it sounds fancy, you have almost all of the pieces needed to understand and implement Eigenfaces (the last necessary piece you will pick up momentarily). Here are the major steps in the Eigenfaces algorithm.

1. Use PCA to compute the k principal components of the training face images (the k eigenvectors with largest eigenvalues).
2. Project the training and test face images onto the k principal components. We'll call this the *facespace* representation of our original images.
3. For each of the test images, compute the closest match between the test image (represented as a k-dimensional vector in facespace) and the training images (again, in facespace). The notion of "closest match" here can be described in a few different ways, but the easiest thing to do is to use the the Euclidean distance to define how far apart two points are. In this way, you would look for the training point that has the smallest Euclidean distance for a particular test point and predict the identity of the test point to be the same as the identity of this closest training point. This method of classification is known as nearest neighbor classification and it is the one new concept you need to implement Eigenfaces.

Exercise 20.4

Earlier in this assignment you wrote some pseudo code for face compression, which as you can see from the description of Eigenfaces above, gets you most of the way there. Before you actually implement Eigenfaces, we'd like you to extend your pseudocode to cover the whole Eigenfaces algorithm. In addition to the steps of Eigenfaces describe above, you should also think about the steps needed to calculate the accuracy of your system (i.e., how often does it get the person's identity correct).

Exercise 20.5

Implement the eigenfaces algorithm.

1. Your code, which can be a script, function, or livescript, should use the training and test sets of images provided: `classdata_train.mat` and `classdata_test.mat`, respectively.
2. You may want to start by identifying one face from the test set, but by the time you are done, your code should run through all of the test images and report the fraction it guesses correctly.
3. Test different numbers of eigenvectors. How many does it take to guess right most of the time?
4. If you'd like, time how long it takes your code to run. Can you do anything more efficiently to make it run faster? (use `tic` and `toc` in MATLAB for timing)
5. Visualize the first few eigenfaces. Can you interpret what they mean?
6. Generate a figure that depicts the success rate (accuracy at determining the identify of a person in an image) versus the number of eigenfaces used.
7. Generate a figure that depicts the success rate (accuracy at determining the identify of a person in an image) versus the number of eigenfaces used where the training data consists of only images of people not smiling and the test data consists of only images of people smiling (these are located in `classdata_non_smiles.mat` and `classdata_smiles.mat` respectively. Comment on the difference in performance on when using these files versus the data from the previous parts of this problem.

Guidelines:

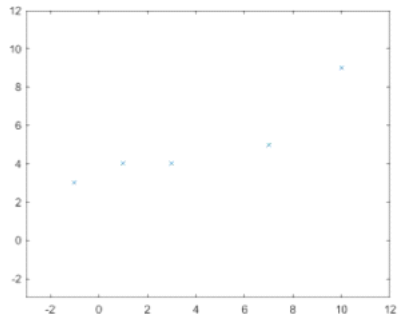
- You should comment your code (use %) so others could read and understand it.
- Don't use the command `pca`, but instead build your algorithm using either the `eig` or `eigs` command (`eigs` computes just a few eigenvectors, which can be faster when you only care about the eigenvectors with large eigenvalues). We want you to think through all the steps involved in your facial recognition program, and that means doing the math "yourself".

(Optional) Extensions (These are not spelled out in much detail. We recommend you talk to a member of the teaching team before trying these (especially the second two).

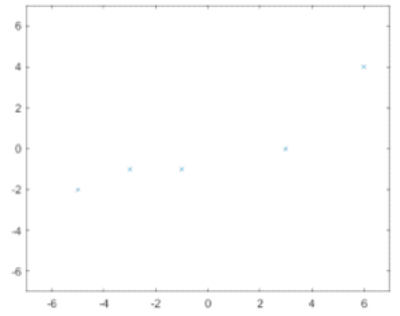
- Analyze the mistakes your algorithm makes (particularly when training on non-smiles and testing on smiles).
- Use Eigenfaces to do smile detection instead of identity recognition.
- Combine Eigenfaces with a classifier other than nearest neighbors (e.g., formulate an LSAE to create a series of one person versus everyone else detectors).
- Get your system working on live video.

Solution 20.1

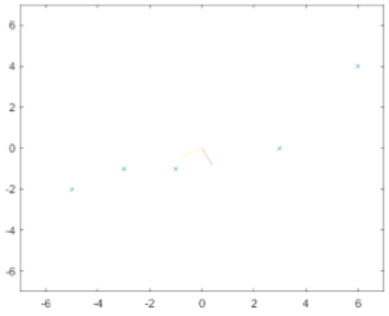
1.
```
>> D = [-1 3; 1 4; 3 4; 7 5; 10 9]
>> plot(D(:,1),D(:,2),'x')
>> axis ([-3 12 -3 12])
```



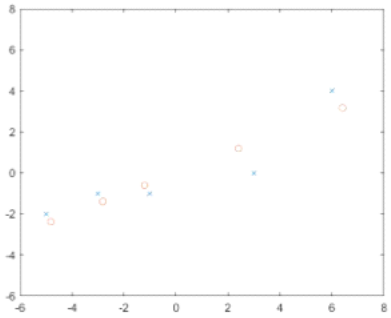
2.
```
>> mumatrix=[mean(D(:,1))*ones(5,1) mean(D(:,2))*ones(5,1)]
>> tildeD=D-mumatrix
>> plot(tildeD(:,1),tildeD(:,2),'x')
>> axis ([-7 7 -7 7])
```



3.
```
>> [Vec,Diam]=eig(tildeD'*tildeD)
>> hold on
>> quiver(0,0,Vec(1,1),Vec(2,1))
>> quiver(0,0,Vec(1,2),Vec(2,2))
```



4.
```
>> proj=tildeD*Vec(:,2)
>> B=proj*Vec(:,2)'
>> hold on
>> plot(tildeD(:,1),tildeD(:,2),'x')
>> plot(B(:,1),B(:,2),'o')
>> axis ([-6 8 -6 8])
```



Solution 20.2

1. No. If we write a data point $a\mathbf{p}_1 + b\mathbf{p}_2$, then the reduced dimension version is $b\mathbf{p}_2$. It's impossible to recover a, which is the information in the perpendicular direction.
2. We would get the information in the perpendicular direction, which we can interpret as the "error" in reducing the dimension from $\mathbf{D}$ to $\mathbf{B}$.
3. You can use the error $\mathbf{B} - \mathbf{D}$.
4. We can write a new data point $\mathbf{d}$ as

$$(\mathbf{d} \cdot \mathbf{p}_1)\mathbf{p}_1 + (\mathbf{d} \cdot \mathbf{p}_2)\mathbf{p}_2.$$

Solution 20.3

1.
```
>> A = (1/sqrt(7304))*[b_tr-mean(b_tr) w_tr-mean(w_tr) s_tr-mean(s_tr)];
>> R=A'*A
```

2.
```
>> [V,D]=eig(R)
>> Vp=[V(:,2) V(:,3)]
```

3.
```
>> T = [b_new-mean(b_tr) w_new-mean(w_tr) s_new-mean(s_tr)]'
```

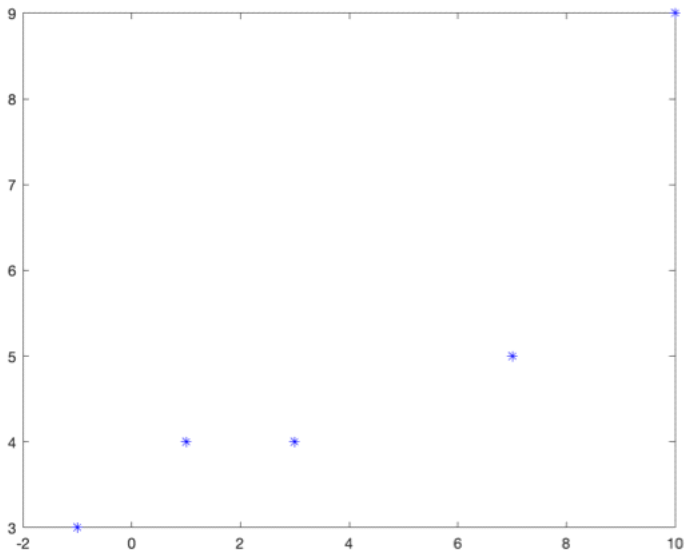
4.
```
>> alpha=Vp'*T;
```

Solution 20.5

A reference implementation of Eigenfaces is linked from the Canvas assignment page.

HW7-1

```
% Excercise 20.1
D = [-1 3; 1 4; 3 4; 7 5; 10 9];

figure; plot(D(:, 1), D(:, 2), "b*");
```




```
D2 = D - mean(D);

figure; hold on;

plot(D2(:, 1), D2(:, 2), "r*");
D_cov = D2' * D2;
[V, D] = eig(D_cov);
quiver(0, 0, V(1, 1), V(2, 1));
quiver(0, 0, V(1, 2), V(2, 2));

projection = D2 * V(:, 2)
```

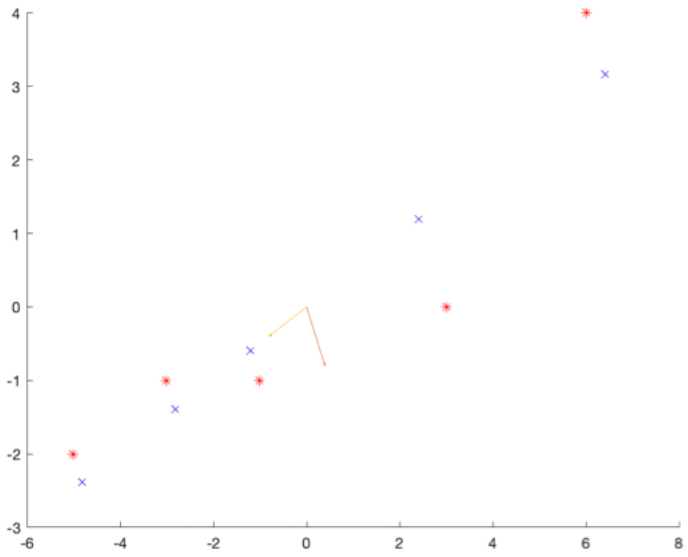
```
projection = 5×1
    5.3684
    3.1323
    1.3398
   -2.6889
   -7.1516
```

```
B = projection * V(:, 2)'
```

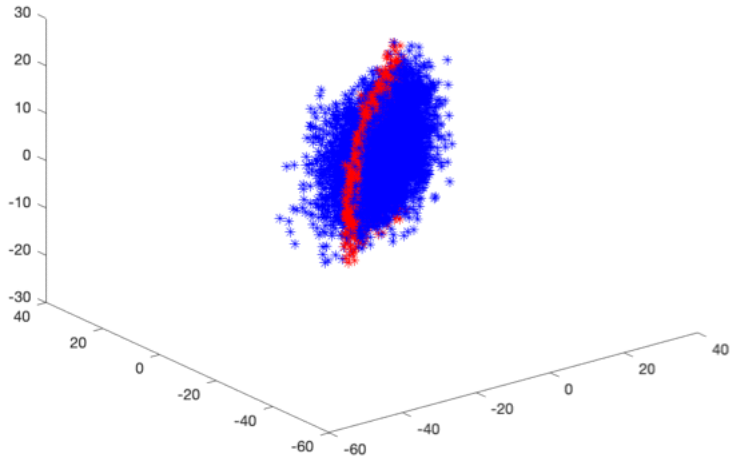
```
B = 5×2
```

```
   -4.8116   -2.3807
   -2.8075   -1.3891
   -1.2008   -0.5941
    2.4100    1.1924
    6.4099    3.1715
```

```
plot(B(:, 1), B(:, 2), 'xb')
```




```
% Excercise 20.3
load("avg_temperatures_pt2.mat");
D = [b_tr w_tr s_tr];
R = D' * D;

[V, M] = eig(R);
Vp = V(:, 2:3);
V1 = Vp(:, 1);
V2 = Vp(:, 2);

Vmagic = [V1'; V2'];

T = [b_tr' - mean(b_tr); w_tr' - mean(w_tr); s_tr' - mean(s_tr)];

alpha = Vmagic * T;

T_hat = Vmagic' * alpha;

hold on; clf;
plot3(T(1, :), T(2, :), T(3, :), "b*"); hold on;
plot3(T_hat(1, :), T_hat(2, :), T_hat(3, :), "r*");
```



Faces

```
  Name                  Size                 Bytes  Class

  grayfaces_test       64x64x356            11665408  double
  smile_test           356x1                    356  logical
  subject_test         356x1                   2848  double
```

Initial Training Sample Face:

Pre-Compressed Mean-Adjusted Reshaped Training Sample Face:

Decompressed Training Face Sample:

test_faces_in_facespace = 356×50

10^3 ×

```
   -0.7902   -0.9978   -0.6244    0.3187   -0.8285    0.9024   -0.6078   -0
   -0.5061   -0.8324   -0.0959    0.2594   -0.6006    0.8093   -0.4340   -0
   -0.8056   -0.9280   -0.5789    1.1013   -0.8719    0.7953   -0.0357   -0
   -0.8074   -0.8201   -0.4869    1.3223   -0.8583    0.6008    0.2503    0
    2.1030   -0.6553    0.4159    0.7487   -1.0638   -0.2990    0.8728   -0
    2.0634   -0.6455    0.4400    0.7754   -0.9873   -0.3180    0.8992   -0
    2.4972   -0.6805    0.5553   -0.0592   -1.1831   -0.1022    0.2087   -0
    2.4625   -0.6628    0.5575    0.2237   -1.1506   -0.1182    0.5055   -0
    2.6487   -0.5396    0.9551    0.6305   -0.0982    0.4625   -0.0898   -0
    2.5724   -0.7453    1.0006    0.5746   -0.3044    0.5583   -0.1255   -0


nearest_neighbors = 356×1
     3
     4
     7
     8
    11
    12
    15
    16
    19
    20


n =

     []
foo = 1×356
     1     1     1     1     2     2     2     2     3     3     3     3
bar = 356×1
     1
     1
     1
     1
     2
     2
     2
     2
     3
     3


ans = 356×2
     1     1
     1     1
     1     1
     1     1
     2     2
     2     2
     2     2
     2     2
     3     3
     3     3
```

HW7-2

```
% `dbstack` returns the call stack. For a live script, this is always at
% least three items long (in my testing). Here, we detect if it's longer
% than that, which means that we're being called by another script
is_main = length(dbstack) <= 3
```

```
is_main = logical
   1
```

```
% We allow the global variable of load_smiles to be set to "pass in"
% if we should train with only non-smiling images but test with only
% similing images.
global load_smiles_only
if ~exist("num_eigenvectors", "var") || isempty(load_smiles_only) || (load_smiles_only ~= 1 && load_smiles_only ~= 0)
    load_smiles_only = true;
end

if load_smiles_only
    load("classdata_smile.mat");
    load("classdata_no_smile.mat");
else
    load("classdata_train.mat");
    load("classdata_test.mat");
end

% We allow the global variable of num_eigenvectors to be set to "pass in" a
% precision level.
global num_eigenvectors
if ~exist("num_eigenvectors", "var") || isempty(num_eigenvectors) || num_eigenvectors < 1
    num_eigenvectors = 10;
end

disp("Using Eigenvectors: " + num_eigenvectors);
```

```
Using Eigenvectors: 7
```

```
train_sample_idx = 42; % index of a sample image to use for validation of the training process
test_sample_idx = 42; % index of a sample image to use for validation of the testing process
```

Training

```
% First, we convert all the images to row vectors
train_faces = imgs_to_row(grayfaces_train);

% Then make sure we haven't corrupted the images yet
figure; disp("Training Sample Face, Reshaped, Unmodified:");
```

```
Training Sample Face, Reshaped, Unmodified:
```

```
imshow(imgs_to_grid(train_faces(train_sample_idx, :)), []);
```

```
% Now, we need to mean-center all the faces. We'll store the mean so we can
% easily un-do this later to display processed images.
train_face_mean = mean(train_faces);
train_faces_centered = train_faces - train_face_mean;

% Double-check that we can still successfully read and display one of the
% images.
disp("Training Sample Face, Mean-Adjusted, Uncompressed:");
```

```
Training Sample Face, Mean-Adjusted, Uncompressed:
```

```
imshow(imgs_to_grid(train_faces_centered(train_sample_idx, :), train_face_mean), []);
```

```
% Now, we can actually do the Principal Component Analysis!
train_faces_covariance = train_faces_centered' * train_faces_centered;
[eigenvectors, ~] = eigs(train_faces_covariance, num_eigenvectors); % ignoring second result, which is eigenvalues
```

```
% Now, let's project all the training data onto our eigenvectors, therefore
% compressing it
train_faces_compressed = train_faces_centered * eigenvectors;
```

```
% For validation purposes, let's decompress the image (project it back to
% the standard basis), and draw it
decompressed_train_faces = train_faces_compressed * eigenvectors';

disp("Training Sample Face, Decompressed:");
```

```
Training Sample Face, Decompressed:
```

```
imshow(imgs_to_grid(decompressed_train_faces(train_sample_idx, :), train_face_mean), []);
```

Testing

```
% Let's load the data
load("classdata_test.mat");
test_faces = imgs_to_row(grayfaces_test);

% Then make sure we haven't corrupted the images yet
disp("Testing Sample Face, Reshaped, Unmodified:");
```

```
Testing Sample Face, Reshaped, Unmodified:
```

```
imshow(imgs_to_grid(test_faces(test_sample_idx, :)), []);
```

```
% Now, we need to mean-center all the faces.
% NB: We're centering it on the mean of the testing faces, not the training
% faces, meaning that the testing and training images are being centered
% around a different point, which is questionable.
test_face_mean = mean(test_faces);
test_faces_centered = test_faces - train_face_mean;

% Double-check that we can still successfully read and display one of the
% images.
disp("Training Sample Face, Mean-Adjusted, Uncompressed:");
```

```
Training Sample Face, Mean-Adjusted, Uncompressed:
```

```
imshow(imgs_to_grid(test_faces_centered(test_sample_idx, :), test_face_mean), []);
```

```
% Now, we can actually do the Principal Component Analysis!
%test_faces_covariance = test_faces_centered' * test_faces_centered;
%[eigenvectors, ~] = eigs(test_faces_covariance, num_eigenvectors); % ignoring second result, which is eigenvalues
```

```
% Now, let's project all the testing data onto our eigenvectors, therefore
% compressing it
test_faces_compressed = test_faces_centered * eigenvectors;
```

```
% For validation purposes, let's decompress an image (project it back to
% the standard basis), and draw it.
decompressed_test_faces = test_faces_compressed * eigenvectors';

disp("Training Sample Face, Decompressed:");
```

```
Training Sample Face, Decompressed:
```

```
imshow(imgs_to_grid(decompressed_test_faces(test_sample_idx, :), test_face_mean), []);
```

```
% Now, that we've projected both the training faces and the testing faces
% onto our eigenvectors, let's figure out the nearest neighbor in the
% training dataset of each of the testing faces. We'll assume that the
% nearest neighbor is the same person.
nearest_neighbors = knnsearch(train_faces_compressed, test_faces_compressed);
```

```
% Finally, we can compare the results of our algorithm with the correct
% answers in subject_tests
% Mean here is a shortcut, since the mean of 0 for wrong or 1 for right
% will be percent correct.
percent_correct = mean(subject_train(nearest_neighbors) == subject_test)
```

```
percent_correct = 0.9607
```

Helper Functions

```
function out = imgs_to_grid(imgs, mean)
    %%
    % Convert a 2D matrix of images as row vectors (img, pikel) to a 3D array of images as
    % grids (x, y, img). If mean is provided, also add it to each pixel.
    %%
    if nargin < 2
        mean = 0;
    end

    out = reshape(imgs + mean, [sqrt(size(imgs, 2)), sqrt(size(imgs, 2)), size(imgs, 1)]);
end

function out = imgs_to_row(imgs)
    %%
    % Convert a 3D array of images (x, y, img) to a 2D matrix of images as
    % row vectors (img, pixel).
    %%
    out = reshape(imgs, [size(imgs, 1) * size(imgs, 2) size(imgs, 3)])';
end

function debuglog(text)
    if ~is_main
        return
    end
    disp(text);
end

function imshow_row(img, mean)
    if ~is_main
        return
    end
    imshow_row(imgs_to_grid(img, mean), [])
end
```

HW7-3

```
[nums_eigs_normal, percents_correct_normal] = spread_algorithm(false);
```

```
is_main = logical
   0
Using Eigenvectors: 1
Training Sample Face, Reshaped, Unmodified:
Training Sample Face, Mean-Adjusted, Uncompressed:
Training Sample Face, Decompressed:
Testing Sample Face, Reshaped, Unmodified:
Training Sample Face, Mean-Adjusted, Uncompressed:
Training Sample Face, Decompressed:
```

```
percent_correct = 0.1489
is_main = logical
   0
Using Eigenvectors: 2
Training Sample Face, Reshaped, Unmodified:
Training Sample Face, Mean-Adjusted, Uncompressed:
Training Sample Face, Decompressed:
Testing Sample Face, Reshaped, Unmodified:
Training Sample Face, Mean-Adjusted, Uncompressed:
Training Sample Face, Decompressed:
```

```
percent_correct = 0.7584
is_main = logical
   0
Using Eigenvectors: 3
Training Sample Face, Reshaped, Unmodified:
Training Sample Face, Mean-Adjusted, Uncompressed:
Training Sample Face, Decompressed:
Testing Sample Face, Reshaped, Unmodified:
Training Sample Face, Mean-Adjusted, Uncompressed:
Training Sample Face, Decompressed:
```

```
percent_correct = 0.8652
is_main = logical
   0
Using Eigenvectors: 4
Training Sample Face, Reshaped, Unmodified:
Training Sample Face, Mean-Adjusted, Uncompressed:
Training Sample Face, Decompressed:
Testing Sample Face, Reshaped, Unmodified:
Training Sample Face, Mean-Adjusted, Uncompressed:
```

```
Training Sample Face, Decompressed:
```

```
percent_correct = 0.9129
is_main = logical
   0
Using Eigenvectors: 5
Training Sample Face, Reshaped, Unmodified:
Training Sample Face, Mean-Adjusted, Uncompressed:
Training Sample Face, Decompressed:
Testing Sample Face, Reshaped, Unmodified:
Training Sample Face, Mean-Adjusted, Uncompressed:
Training Sample Face, Decompressed:
```

```
percent_correct = 0.9382
is_main = logical
   0
Using Eigenvectors: 6
Training Sample Face, Reshaped, Unmodified:
Training Sample Face, Mean-Adjusted, Uncompressed:
Training Sample Face, Decompressed:
Testing Sample Face, Reshaped, Unmodified:
Training Sample Face, Mean-Adjusted, Uncompressed:
Training Sample Face, Decompressed:
```

```
percent_correct = 0.9466
is_main = logical
   0
Using Eigenvectors: 7
Training Sample Face, Reshaped, Unmodified:
Training Sample Face, Mean-Adjusted, Uncompressed:
Training Sample Face, Decompressed:
Testing Sample Face, Reshaped, Unmodified:
Training Sample Face, Mean-Adjusted, Uncompressed:
Training Sample Face, Decompressed:
```

```
percent_correct = 0.9607
is_main = logical
   0
Using Eigenvectors: 8
Training Sample Face, Reshaped, Unmodified:
Training Sample Face, Mean-Adjusted, Uncompressed:
Training Sample Face, Decompressed:
Testing Sample Face, Reshaped, Unmodified:
```

```
Training Sample Face, Mean-Adjusted, Uncompressed:
Training Sample Face, Decompressed:
```

```
percent_correct = 0.9663
is_main = logical
   0
Using Eigenvectors: 9
Training Sample Face, Reshaped, Unmodified:
Training Sample Face, Mean-Adjusted, Uncompressed:
Training Sample Face, Decompressed:
Testing Sample Face, Reshaped, Unmodified:
Training Sample Face, Mean-Adjusted, Uncompressed:
Training Sample Face, Decompressed:
```

```
percent_correct = 0.9747
is_main = logical
   0
Using Eigenvectors: 10
Training Sample Face, Reshaped, Unmodified:
Training Sample Face, Mean-Adjusted, Uncompressed:
Training Sample Face, Decompressed:
Testing Sample Face, Reshaped, Unmodified:
Training Sample Face, Mean-Adjusted, Uncompressed:
Training Sample Face, Decompressed:
```

```
percent_correct = 0.9747
is_main = logical
   0
Using Eigenvectors: 15
Training Sample Face, Reshaped, Unmodified:
Training Sample Face, Mean-Adjusted, Uncompressed:
Training Sample Face, Decompressed:
Testing Sample Face, Reshaped, Unmodified:
Training Sample Face, Mean-Adjusted, Uncompressed:
Training Sample Face, Decompressed:
```

```
percent_correct = 0.9860
is_main = logical
   0
Using Eigenvectors: 20
Training Sample Face, Reshaped, Unmodified:
Training Sample Face, Mean-Adjusted, Uncompressed:
Training Sample Face, Decompressed:
```

```
Testing Sample Face, Reshaped, Unmodified:
Training Sample Face, Mean-Adjusted, Uncompressed:
Training Sample Face, Decompressed:
```

```
percent_correct = 0.9860
is_main = logical
   0
Using Eigenvectors: 25
Training Sample Face, Reshaped, Unmodified:
Training Sample Face, Mean-Adjusted, Uncompressed:
Training Sample Face, Decompressed:
Testing Sample Face, Reshaped, Unmodified:
Training Sample Face, Mean-Adjusted, Uncompressed:
Training Sample Face, Decompressed:
```

```
percent_correct = 0.9916
is_main = logical
   0
Using Eigenvectors: 30
Training Sample Face, Reshaped, Unmodified:
Training Sample Face, Mean-Adjusted, Uncompressed:
Training Sample Face, Decompressed:
Testing Sample Face, Reshaped, Unmodified:
Training Sample Face, Mean-Adjusted, Uncompressed:
Training Sample Face, Decompressed:
```

```
percent_correct = 0.9916
is_main = logical
   0
Using Eigenvectors: 40
Training Sample Face, Reshaped, Unmodified:
Training Sample Face, Mean-Adjusted, Uncompressed:
Training Sample Face, Decompressed:
Testing Sample Face, Reshaped, Unmodified:
Training Sample Face, Mean-Adjusted, Uncompressed:
Training Sample Face, Decompressed:
```

```
percent_correct = 0.9916
is_main = logical
   0
Using Eigenvectors: 50
Training Sample Face, Reshaped, Unmodified:
Training Sample Face, Mean-Adjusted, Uncompressed:
```

```
Training Sample Face, Decompressed:
Testing Sample Face, Reshaped, Unmodified:
Training Sample Face, Mean-Adjusted, Uncompressed:
Training Sample Face, Decompressed:
```

```
percent_correct = 0.9916
is_main = logical
   0
Using Eigenvectors: 75
Training Sample Face, Reshaped, Unmodified:
Training Sample Face, Mean-Adjusted, Uncompressed:
Training Sample Face, Decompressed:
Testing Sample Face, Reshaped, Unmodified:
Training Sample Face, Mean-Adjusted, Uncompressed:
Training Sample Face, Decompressed:
```

```
percent_correct = 0.9944
is_main = logical
   0
Using Eigenvectors: 100
Training Sample Face, Reshaped, Unmodified:
Training Sample Face, Mean-Adjusted, Uncompressed:
Training Sample Face, Decompressed:
Testing Sample Face, Reshaped, Unmodified:
Training Sample Face, Mean-Adjusted, Uncompressed:
Training Sample Face, Decompressed:
```

```
percent_correct = 0.9944
```

```
[nums_eigs_smiles, percents_correct_smiles] = spread_algorithm(true);
```

```
is_main = logical
   0
Using Eigenvectors: 1
Training Sample Face, Reshaped, Unmodified:
Training Sample Face, Mean-Adjusted, Uncompressed:
Training Sample Face, Decompressed:
Testing Sample Face, Reshaped, Unmodified:
Training Sample Face, Mean-Adjusted, Uncompressed:
Training Sample Face, Decompressed:
```

```
percent_correct = 0.5478
is_main = logical
```

```
   0
Using Eigenvectors: 2
Training Sample Face, Reshaped, Unmodified:
Training Sample Face, Mean-Adjusted, Uncompressed:
Training Sample Face, Decompressed:
Testing Sample Face, Reshaped, Unmodified:
Training Sample Face, Mean-Adjusted, Uncompressed:
Training Sample Face, Decompressed:
```

```
percent_correct = 0.7135
is_main = logical
   0
Using Eigenvectors: 3
Training Sample Face, Reshaped, Unmodified:
Training Sample Face, Mean-Adjusted, Uncompressed:
Training Sample Face, Decompressed:
Testing Sample Face, Reshaped, Unmodified:
Training Sample Face, Mean-Adjusted, Uncompressed:
Training Sample Face, Decompressed:
```

```
percent_correct = 0.8202
is_main = logical
   0
Using Eigenvectors: 4
Training Sample Face, Reshaped, Unmodified:
Training Sample Face, Mean-Adjusted, Uncompressed:
Training Sample Face, Decompressed:
Testing Sample Face, Reshaped, Unmodified:
Training Sample Face, Mean-Adjusted, Uncompressed:
Training Sample Face, Decompressed:
```

```
percent_correct = 0.8371
is_main = logical
   0
Using Eigenvectors: 5
Training Sample Face, Reshaped, Unmodified:
Training Sample Face, Mean-Adjusted, Uncompressed:
Training Sample Face, Decompressed:
Testing Sample Face, Reshaped, Unmodified:
Training Sample Face, Mean-Adjusted, Uncompressed:
Training Sample Face, Decompressed:
```

```
percent_correct = 0.8820
```

```
is_main = logical
   0
Using Eigenvectors: 6
Training Sample Face, Reshaped, Unmodified:
Training Sample Face, Mean-Adjusted, Uncompressed:
Training Sample Face, Decompressed:
Testing Sample Face, Reshaped, Unmodified:
Training Sample Face, Mean-Adjusted, Uncompressed:
Training Sample Face, Decompressed:
```

```
percent_correct = 0.8989
is_main = logical
   0
Using Eigenvectors: 7
Training Sample Face, Reshaped, Unmodified:
Training Sample Face, Mean-Adjusted, Uncompressed:
Training Sample Face, Decompressed:
Testing Sample Face, Reshaped, Unmodified:
Training Sample Face, Mean-Adjusted, Uncompressed:
Training Sample Face, Decompressed:
```

```
percent_correct = 0.9101
is_main = logical
   0
Using Eigenvectors: 8
Training Sample Face, Reshaped, Unmodified:
Training Sample Face, Mean-Adjusted, Uncompressed:
Training Sample Face, Decompressed:
Testing Sample Face, Reshaped, Unmodified:
Training Sample Face, Mean-Adjusted, Uncompressed:
Training Sample Face, Decompressed:
```

```
percent_correct = 0.9213
is_main = logical
   0
Using Eigenvectors: 9
Training Sample Face, Reshaped, Unmodified:
Training Sample Face, Mean-Adjusted, Uncompressed:
Training Sample Face, Decompressed:
Testing Sample Face, Reshaped, Unmodified:
Training Sample Face, Mean-Adjusted, Uncompressed:
Training Sample Face, Decompressed:
```

```
percent_correct = 0.9270
is_main = logical
   0
Using Eigenvectors: 10
Training Sample Face, Reshaped, Unmodified:
Training Sample Face, Mean-Adjusted, Uncompressed:
Training Sample Face, Decompressed:
Testing Sample Face, Reshaped, Unmodified:
Training Sample Face, Mean-Adjusted, Uncompressed:
Training Sample Face, Decompressed:
```

```
percent_correct = 0.9242
is_main = logical
   0
Using Eigenvectors: 15
Training Sample Face, Reshaped, Unmodified:
Training Sample Face, Mean-Adjusted, Uncompressed:
Training Sample Face, Decompressed:
Testing Sample Face, Reshaped, Unmodified:
Training Sample Face, Mean-Adjusted, Uncompressed:
Training Sample Face, Decompressed:
```

```
percent_correct = 0.9466
is_main = logical
   0
Using Eigenvectors: 20
Training Sample Face, Reshaped, Unmodified:
Training Sample Face, Mean-Adjusted, Uncompressed:
Training Sample Face, Decompressed:
Testing Sample Face, Reshaped, Unmodified:
Training Sample Face, Mean-Adjusted, Uncompressed:
Training Sample Face, Decompressed:
```

```
percent_correct = 0.9579
is_main = logical
   0
Using Eigenvectors: 25
Training Sample Face, Reshaped, Unmodified:
Training Sample Face, Mean-Adjusted, Uncompressed:
Training Sample Face, Decompressed:
Testing Sample Face, Reshaped, Unmodified:
Training Sample Face, Mean-Adjusted, Uncompressed:
Training Sample Face, Decompressed:
```

```
percent_correct = 0.9607
is_main = logical
   0
Using Eigenvectors: 30
Training Sample Face, Reshaped, Unmodified:
Training Sample Face, Mean-Adjusted, Uncompressed:
Training Sample Face, Decompressed:
Testing Sample Face, Reshaped, Unmodified:
Training Sample Face, Mean-Adjusted, Uncompressed:
Training Sample Face, Decompressed:
```

```
percent_correct = 0.9551
is_main = logical
   0
Using Eigenvectors: 40
Training Sample Face, Reshaped, Unmodified:
Training Sample Face, Mean-Adjusted, Uncompressed:
Training Sample Face, Decompressed:
Testing Sample Face, Reshaped, Unmodified:
Training Sample Face, Mean-Adjusted, Uncompressed:
Training Sample Face, Decompressed:
```

```
percent_correct = 0.9607
is_main = logical
   0
Using Eigenvectors: 50
Training Sample Face, Reshaped, Unmodified:
Training Sample Face, Mean-Adjusted, Uncompressed:
Training Sample Face, Decompressed:
Testing Sample Face, Reshaped, Unmodified:
Training Sample Face, Mean-Adjusted, Uncompressed:
Training Sample Face, Decompressed:
```

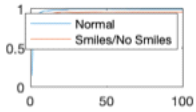
```
percent_correct = 0.9635
is_main = logical
   0
Using Eigenvectors: 75
Training Sample Face, Reshaped, Unmodified:
Training Sample Face, Mean-Adjusted, Uncompressed:
Training Sample Face, Decompressed:
Testing Sample Face, Reshaped, Unmodified:
Training Sample Face, Mean-Adjusted, Uncompressed:
```

```
Training Sample Face, Decompressed:
```

```
percent_correct = 0.9607
is_main = logical
   0
Using Eigenvectors: 100
Training Sample Face, Reshaped, Unmodified:
Training Sample Face, Mean-Adjusted, Uncompressed:
Training Sample Face, Decompressed:
Testing Sample Face, Reshaped, Unmodified:
Training Sample Face, Mean-Adjusted, Uncompressed:
Training Sample Face, Decompressed:
```

```
percent_correct = 0.9607
```

```
plot(nums_eigs_normal, percents_correct_normal); hold on;
plot(nums_eigs_smiles, percents_correct_smiles);
legend("Normal", "Smiles/No Smiles");
```




```
function [nums_eigs, percents_correct] = spread_algorithm(use_smiles)
    global num_eigenvectors
    global load_smiles_only

    load_smiles_only = use_smiles;

    % number of eigenvectors to be used, each entry will be tested
    nums_eigs = [1 2 3 4 5 6 7 8 9 10 15 20 25 30 40 50 75 100]; % arbitrary list of values
    percents_correct = zeros(length(nums_eigs), 1);
    % Let's try the algorithm with a bunch of different numbers of eigenvectors
    % and compare the accuracy.
    for i=1:length(nums_eigs)
        num_eigenvectors = nums_eigs(i);
        run_face_rec;
        percents_correct(i) = percent_correct;
    end
end
```