# COM2004/3004 Lab Week 8

**Principal Component Analysis**

**Objective**

**In the lab you will use Principal Components Analysis to produce a compact representation of the letter image data. You will demonstrate that this compact representation provides a basis for robust letter classification.**

## Background

In last weeks lab you performed dimensionality reduction using simple feature selection. Using a combination of brute force and ingenuity you found it was possible to select individual pixel features that allowed robust classification with feature vectors with only 10 elements. This week you will perform a similar degree of dimensionality reduction but this time using PCA to construct features from a linear combination of the original pixel values. It will require a lot less tweaking!

## 1. Download the data and tools

Make a folder for the lab class and from MOLE download the following files.

lab\_data.mat -- the data

divergence.m -- for measuring 1-D divergence

classify.m -- a nearest neighbour classifier

## 2. Loading the lab data

Start MATLAB and load the lab\_data (i.e. “load lab\_data.mat”).

The following variables should appear in your MATLAB workspace

train\_data, train\_labels, test\_data, test\_labels, test2\_data, test2\_labels

Similarly to last week, the matrices train\_dat, test\_data and test2\_data contain 699, 200 and 200 feature vectors respectively stored in a matrix form. Each row represents one feature vector and contains the 900 pixel values for one character.

Remember, you can view the i*th* character in the training set by typing,

imagesc(reshape(train\_data(i,:),30,30))

colormap (gray)

The vector train\_labels and test\_labels store the character labels as integers using a code where 1=A, 2=B, 3=C. Try displaying some other characters.

## 3. Computing the Principal Components

Remember the principal components are simply *the eigenvectors of the covariance matrix*. They can be computed using the training data with just two lines of MATLAB code. e.g. to compute the first 40 principal components use the following,

covx = cov(train\_data);

[v, d] = eigs(covx, 40);

The function eigs will return the eigenvectors (i.e. principal component axes) as column vectors in the matrix v. Check the sizes of covx and v using ‘size(covx)’ and ‘size(v)’. Make sure that you understand why these matrices have the sizes that they have.

You can view the principal components as images (`eigenletters’ ?) by reshaping them into a 30 by 30 matrix. Look at the first 4,

subplot(2,2,1);

imagesc(reshape(v(:,1), 30, 30));

subplot(2,2,2);

imagesc(reshape(v(:,2),30,30));

subplot(2,2,3);

imagesc(reshape(v(:,3), 30, 30));

subplot(2,2,4);

imagesc(reshape(v(:,4),30,30));

colormap(gray);

You can think of these as `basis images’. PCA represents the original letter data as a weighted sum of these eigenletter images. Loosely speaking, they are like the common components from which the set of letter images are `built’.

**4. Project the data onto the Principal Component Axes**

We will now perform the actual dimensionality reduction by projecting the 900 dimensional images onto the first 40 principal components, i.e. the ‘linear transform, y=Ax’ (actually, because, oppositely to the lecture notes, our images are stored as *row vectors* and our principal components as *column vectors*, the equation looks like y = x A, but don’t let this confuse you). Also, it is helpful to ‘center’ the data before transforming it by subtracting the mean letter vector. So we have,

pcatrain\_data = (train\_data - repmat(mean(train\_data), 699, 1)) \* v;

pcatest\_data = (test\_data – repmat(mean(train\_data), 200, 1)) \* v;

pcatest2\_data = (test2\_data – repmat(mean(train\_data), 200, 1)) \* v;

The mysterious ‘repmat’ (*repeat matrix*) command copies the mean vector so that it can be subtracted from every training or test vector in the dataset in one go, without needing a loop.

Because we have only used the first 40 principal components some information has been lost. But because the principal component are ordered by the amount of variance that they capture, the amount of information lost will be minimized. (Note, the word ‘information’ is being used in a rather wooly way here, but the above statement is true in a more technical sense as long as certain assumptions are made about the distribution of the data. But these details needn’t overly concern us.)

To see how closely the original image can be reconstructed we can project back from the 40-d space to the original 900 dimension. After projecting back we should remember to undo the centering by adding back the mean vector. This can be done simply by

reconstructed = pcatrain\_data \* v’ + repmat(mean(train\_data),699,1);

**(Take care not when typing the instruction above not to miss the transpose on v i.e., v’)**

Display a reconstructed image,

subplot(1,2,1);

imagesc(reshape(train\_data(1,:), 30, 30));

subplot(1,2,2);

imagesc(reshape(reconstructed\_train\_data(1,:),30,30));

You should see that the reconstructed letter looks very similar to the original. Notice how, because only 40 PCA components have been used rather than all 900, the image is somewhat ‘smoothed’. This is a good thing – the largely irrelevant noise component has been effectively removed.

To project into N-dimensional PCA space and back again we can do,

N=6

reconstructed = (train\_data(1,:)-mean(train\_data))\*v(:,1:N)\*v(:, 1:N)’ + mean(train\_data);

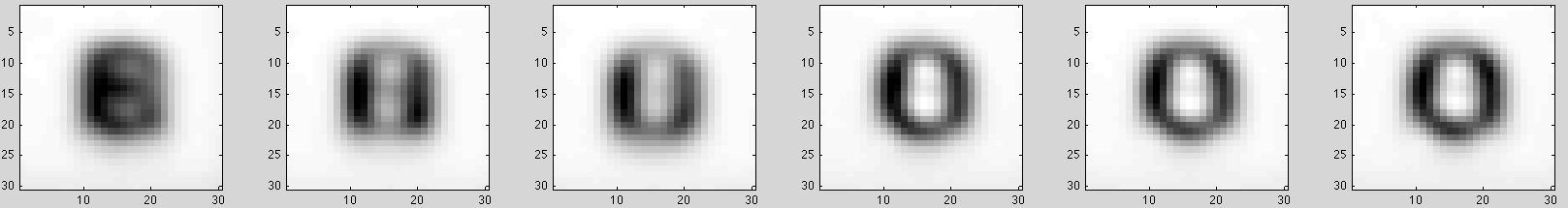
subplot(1,2,1);

imagesc(reshape(train\_data(1,:),30,30));

subplot(1,2,2);

imagesc(reshape(reconstructed,30,30));

Experiment with different values of N. (Save the commands in a .m file to save yourself having to retype them.) The figure below shows an example using the letter ‘O’ in train\_data(1,:) with N equal to 1, 2, 3, 4, 5 and 6. Notice how the O’ness of the ‘O’ is captured by just the first 4 principal components! Experiment with different letters.



## 5. Performing the classification

We can now test our classifier on the PCA’ed data

Try first using all 40 dimensions,

classify(pcatrain\_data, train\_labels, pcatest\_data, test\_labels,1:40);

Now try with just 10,

classify(pcatrain\_data, train\_labels, pcatest\_data, test\_labels,1:10);

Results may seem a little disappointing, probably about 88 and 83%. This is not so high compared with the figures in the 90’s that could be achieved with simple feature selection. But now try classifying again but this time using PCA components 2 to 11,

classify(pcatrain\_data, train\_labels, pcatest\_data, test\_labels,2:11);

(Also try evaluating the classifier with the pcatest2\_data.) Classification performance when using features 2 to 11 should be considerably better than when using features 1 to 10 (try and think why?). Performance should now be comparable to the best results that were achieved in Week 7. But note, these results have been achieved with considerably less ‘fuss’. e.g. there was no trial and error or brute force search. PCA just worked. It will work equally well for many similar problems and for larger and more difficult data sets (e.g. face recognition).

**6. Feature Selection**

Typically, with PCA, we would take the first *N* PCA components as our features. However, let us try combining PCA and the feature selection ideas from last week. Starting with the 40 principal components that you have computed can you find the 10 that give best performance.

You could try using trial and error or you could try reusing the divergence code from last week: Compute the 1-D divergences for the PCA features. Remember you will need some way of summing divergences over pairs of classes. Rank the PCA features according to their divergence and pick the best 10. (It is more valid to do this on PCA features than on raw pixel features. Why?)

Test your feature selection on the test2 dataset. Can you score higher than the highest score that we achieved by selecting individual pixels, i.e. 94%? I haven’t tried this myself, so I will be genuinely interested to see if there are 10-d feature sets that can beat the set [2,3,4,5,6,7,8,9,10,11] that we used in Section 5.

**7. Robustness to noise** (if you have time)

Try adding Gaussian noise to the 900-dimensional test data (i.e. in the same way that you did for the assignment). Now, transform the noisy data into the PCA domain and rerun the classifier. Compare the noise robustness of the original 900-d feature vectors (assignment stage 1) and the 10-d PCA based features (this week). Which of these feature vectors can tolerate the greatest amount of added noise in the test data? Why do you think this is? Again, I haven’t tried this myself and will be interested to hear what you find.

First written by Jon Barker, 17/11/2011