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 week's 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 a technique called Principal Components Analysis (PCA) which will construct features that are a linear combination of the original pixel values. It will produce good performance without the need for a complex feature selection algorithm.

1. Download the data and tools

Make a folder for the lab class and download from the MOLE week 8 lab folder 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. Principal Component Analysis – a primer

We will be covering PCA in detail in the lecture session on Friday. This section is intended to give you a brief overview that will help you understand what is happening in today's lab.

As you have seen, the letter images in your training and test sets are represented by 900-dimensional feature vectors. Hence, any particular letter image can be pictured as a single point in a 900-d space. The complete training set is therefore a collection of 699 points in in 900-d space, i.e. the training data forms a small cluster in 900-d space. Now, this cluster will be more spread about its centre in some directions than in others. The PCA technique simply identifies a set of orthogonal directions in which the amount of spread (i.e. variance) is the greatest. i.e. consider first finding the one direction in which the spread is greatest; then find a direction in which the spread is greatest but which is also at right-angles to the first direction found; then find a direction in which the spread is greatest but that is at right angle to the first two directions; and so on. (Remember than in 900-d space there can be 900 mutually perpendicular directions.) These directions – ordered in terms of spread -- will be known as **principal components**.

How does finding the principal components allow us to perform dimensionality reduction? Well, once the first few principal components have been found we can measure the distance of a point from the cluster center (i.e. from the mean) along these few directions, e.g. perhaps 10 directions. Now we can represent the 900-d point by these 10 measurements. Measuring the point positions along these directions is known as '**projecting onto the principal component axes'**. Crucially, because the directions are ordered by the size of the spread in that direction, the position along the first, say 10, principal components axes will approximate the position of the point in 900-d space, i.e. the other 890 orthogonal direction are less important. This projection step is simple to compute, it is actually just a linear transform of the 900-d vector, x, i,e, y = A x.

There is another way of thinking about the principal components. Each axis direction can be represented as a 900-d vector. This vector itself could be visualized as a 30 by 30 image, i.e. each principal component is now being conceptualized as an image (a `**basis image'**) rather than as an axis. Recall that when we projected onto the principal component axes we were taking a 900-d feature vector and describing its position relative to the centre of the cluster as a sum of movements along the principal component axes directions. In the alternative view, this is equivalent to describing the original image as being formed by an average image plus a weighted sum of the basis images, i.e. the position of the point along each principal component axes is telling us by how much to weight the corresponding basis image in the sum. i.e. we can approximate every image in the training and test set as the weighted sum of a small number of basis images.

All of the above would not be very interesting if it was not possible to actually compute the principal components. Fortunately it turns out – as will be explained in the lecture – that the principal components are the **eigenvectors of the data's covariance matrix**. This might sound complicated, but it is just a few lines of MATLAB.

Do not worry if you did not completely follow all of the above. Reread it at the end of the lab class and come along to the lecture on Friday.

4. Computing the Principal Components

As we will see in the lecture on Friday, 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 basis 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);
```

PCA will represent 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. Projecting 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=Vx' (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 V; don't let this confuse you). Also, it is necessary to `centre' 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 can be 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 when typing the instruction above not to miss the transpose on V i.e., `V'`)

Display a reconstructed image,

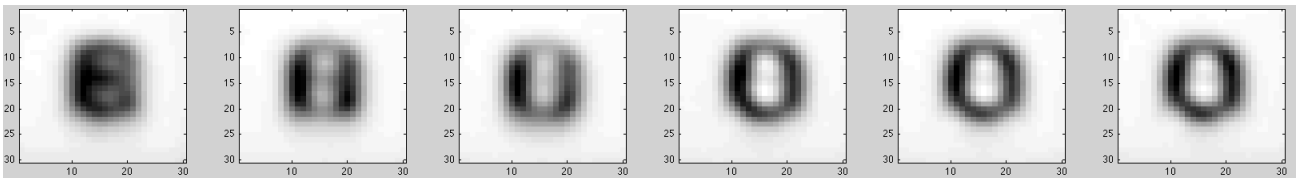
```
subplot(1,2,1);
imagesc(reshape(train_data(1,:), 30, 30));
subplot(1,2,2);
imagesc(reshape(reconstructed(1,:),30,30));
```

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 just the first image from the training data set into *N*-dimensional PCA space and back again we can use,

```
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. For example, how many components are needed to capture the `A'ness of an `A'?



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 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);
```

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.

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 1-D divergence and simply pick the best 10, i.e. assume that the 10-D divergence can be estimated as the sum of the 10 1-D divergences. (It is more valid to do this when using PCA features than when using 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 is a 10-d feature set 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. random numbers generated using `randn`.) 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.