COM2004/3004 Lab Week 7
Feature Selection Challenge!

Objective

Using the assignment data, what is the best classification score that you can achieve using just 10 of the 900 available features (i.e. 10 different pixel positions)?

You will try to find the best performing set of 10 features by using the feature selection ideas discussed in the lectures. Some tools have been provided. The lab sheet will lead you through the initial steps to get you started.

There will be a *prize* for the person (or team) who gets the highest score!

Background

In Part 2 of the assignment you will be using a typed-character classifier that takes a 30x30 pixel image as input. Each image can therefore be represented as a 900-dimensional feature vector containing the pixel values. You should find that using these feature vectors and a nearest neighbour classifier you are able to achieve pretty accurate classification.

One drawback of the nearest-neighbour classifier is that it has to store the complete training set in memory. This can be a problem if the feature vectors are large and memory is in short supply (e.g. in embedded systems). Computing distances between large feature vectors can also take a long time. It is therefore desirable to reduce the size of the feature vector. In this practical we are going to use feature selection to reduce the size of the feature vector from 900 to just 10 (i.e. 90 times less memory, 90 times less computation). It is important that we choose the 10 features carefully so that we do not loose too much classification performance.

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
classify.m         -- a nearest neighbour classifier
divergence.m       -- for measuring 1-D divergence
multidivergence.m – for measuring multi-dimensional divergence
```

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
```

The matrices `train_data` and `test_data` contain a portion of the data that is being used in the assignment, but here it is being stored in matrix form. Each row represents one feature vector and

contains the 900 pixel values for one character. There are 700 training characters and 200 for testing.

View the first character in the training set by typing,

```
imagesc(reshape(train_data(1,:),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.

Check the label for the first character in the training set by typing,

```
train_labels(1)
```

3. Using the classify.m code

I have provided a function that performs nearest-neighbour classification, classify.m.

View the code by typing,

```
type classify.m
```

The code is quite compact and is a little hard to understand. It has been written in a way that avoids using loops which ensures that it runs as quickly as possible. It is using a cosine distance rather than the more commonly-employed Euclidean distance. (The cosine distance is based on the angle between a pair of feature vectors rather than the distance between points.)

Try out the code by typing,

```
score=classify(train_data, train_labels, test_data, test_labels)
```

You should get a score of around 97% … rather higher than achieved with a Euclidean distance. (Why?)

The function `classify.m` can take a 5th argument: a vector of integers representing the indexes of features to be selected. e.g. to test the classifier using just the 50th and 150th pixel in the image,

```
score=classify(train_data, train_labels, test_data, test_labels, [50 150])
```

Performance now should be much lower!

COMPETITON: Your challenge is to find a vector of 10 numbers, `call it`, `features`, such that,
`classify(train_data, train_labels, test_data, test_labels, features)`,
returns a result as close to 100% as possible. Email them to me and maybe win a prize!

4. Measuring Divergence

The function `divergence.m` computes the 1-D divergence between a pair of classes, i.e. each feature is considered separately and is modeled by a Gaussian distribution. As input it takes two parameters: the first is a matrix of data for class 1, and the second is a matrix of data for class 2.

Look at the code by typing,

```
type divergence.m
```

The first couple of lines are computing the means and variances of each of the 900 pixels for each class. The last line is using the formula for 1-D divergence, but it works on a vector of means and variances, so it can compute 900 1-D divergences in a single go without needing a loop.

To use `divergence.m` we must first select feature vectors from `train_data` belonging the pair of classes that we want to consider. This can be done using the `train_label` vector. For example, to select all the 'A's and 'B's type,

```
adata = train_data(train_labels==1,:);
bdata = train_data(train_labels==2,:);
```

Now we can measure the 900 separate 1-d divergences between the 'A' and 'B' class,

```
d12 = divergence(adata, bdata);
```

To see the divergences as an image type,

```
imagesc(reshape(d12,30,30));
colormap(1-gray);
colorbar;
```

5. Feature Selection

Let's say that we now want to select features that will be good for separating 'A' s and 'B's. We need to choose features with high divergence. To do this we will sort the divergence values, d12, and look at the indexes of the highest values,

```
[d12_sorted, indexes] = sort(-d12);
```

(Why are we sorting –d12 and not d12?).
The simplest feature selection algorithm would then just take the top 10 highest, i.e. the first 10 entries in the indexes vector. Type,

```
features = indexes(1:10);
classify(train_data, train_labels, test_data, test_labels, features)
```

The overall performance might not be very good. Remember the features have only been selected on the basis of how well they separate A and B. The other 24 letters have not been considered.

To examine the performance of the classifier in more detail we can look at a confusion matrix. The function classify.m will produce a confusion matrix if it is called like this

```
[score, cm] = classify(train_data, train_labels, test_data,
test_labels, features);
cm
```

If you look at the matrix you should see that not all the A's and B's have been correctly classified, but there are at least no cases of A's mistaken for B's or vice versa.

6. Improving the Feature Selection

i/ Two classes versus many classes

To get a good overall classification score you need to find a set of features that have high divergence scores not just with respect to the A and B class, but for *every pair* of letters. Look again at Section 4 and consider how you might use a nested loop to call divergence repeatedly in such a way that you compute divergence between every pair of letters. How might all these pairwise scores be combined to compute some overall divergence score?

ii/ Correlation between features

Remember, when selecting multiple features we can't estimate the overall divergence by simply summing the divergences of each individual feature. If the features are correlated the combined divergence is reduced. You can compute the correlation between each of the 900 features and each other 900 feature and store the results in a 900x900 matrix using,

```
corr = corrcoef(train_data);
imagesc(corr);
```

(Can you explain the pattern of the corr image?)

So when selecting your 10 features you want to find features for which the divergence is high but which have low correlation, e.g. compare the correlation between pixels 1 and 2 with the correlation between pixels 322 and 338, i.e. type,

```
corr(1,2)
corr(338,322)
```

iii/ Multivariate divergence

The function `multidivergence.m` computes the multidimensional divergence while assuming that the data is normally distributed (next lecture). So you can test the divergence for a complete feature set as follows,

```
features = [1, 20, 50, 60];   % Pick 4 features
multidivergence(adata, bdata, features)
```

Using this function you could try implementing a sequential forward search like that discussed in the lectures. e.g. first of all find the 1-feature that produces the highest divergence

```
for i=1:nfeatures
d(i) = multidivergence(adata, bdata, i);
end
[~, index1] = max(d);
index1
```

Now try to find the best pair,

```
d=zeros(1, nfeatures);
for i=1:nfeatures
  if (i~=index1)
    d(i) = multidivergence(adata, bdata, [index1, i]);
  end
end
[~, index2] = max(d);
[index1, index2]
```

Now repeat until you have found all 10.

Again, if you want to get a good overall classification result you will need to use the ideas from 6,i) and sum the multidivergences over not just 'A' versus 'B' but over all letter pairs.

7. Competition

Find the 10 features that produce the best classification result. I don't mind whether you do this by guesswork or by using some fancy feature selection algorithm. For example,

```
features=[ 7    74   134   162   248   278   305   741   772   821]
classify(train_data, train_labels, test_data, test_labels, features)
```

Produces a score of 18%. Not very good. I believe it is possible to get the score up to at least 80%.

Once you are happy that you have a good set of numbers, email them to me along with the score that they produce. Use COMPETITION as the subject title. Send the email before Sunday.

I will announce the winner in the Monday lecture.