Lab Session 3 Convolution, Classification and Clustering

Key aims of the Session:

- Better understand convolution
- Begin to look at classification, clustering and segmentation

Contents

Editor vs Command Window

While you can complete this worksheet by entering commands sequentially into the command window, you may find it more useful to save and run the commands as a script. To open up the script editor, type `edit` into the command window. Don't forget to save your script so that you don't lose it. Once saved, you can run your script by clicking "Run" on the top ribbon or by typing the script's name into the command window. Please note that MATLAB script filenames should have a .m extension.

1: Properties of Convolution and Edge Detection

In order for artificial intelligence models to make effective use of image and video data, the content must be interpreted beyond simply reading the colour intensity values. Convolution is an important step in achieving this. In this exercise, we will look at two edge detection approaches but there are many more to consider.

Import the image "LancsLogo.jpg" to Matlab, convert it to double, divide by 255 and take the mean across the red, green and blue colour channels, storing it as the variable `im`.

Display the image and convolved image in a the first subplot of a 3x1 array. Set the colormap to gray, turn off the axis and set the display aspect ratio to match the image with `axis image off`.

The image includes a lot of white space that we're not interested in. Remove the first 275 rows and the final 250 rows. Replace the previous plot with the cropped image without the axis and with the image's aspect ratio.

Store the size of the image in a variable:
```
sz = size(im);
```

Edge Detection Kernel

Create the following kernel matrix ker:

$$ker = \begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$$

Carry out the convolution of the kernel and image $cim = ker * im$ using the "conv2" command

```
cim = conv2(im,ker,'same');
```

Display the absolute value `abs(cim)` of the convolved image in the second subplot of the figure. What is this showing, and can you see how the kernel interacts with the image matrix via convolution?

In order to identify edges, you can define a cut-off threshold value τ for the kernel response values to form an edge map:

$$edgemap[i,j] = \begin{cases} 1 \text{ if } abs(cim[i,j]) \geq \tau \\ 0 \text{ otherwise} \end{cases}$$

Create an edgemap from your image with threshold $\tau = 0.15$ and display this in the third subplot.

Try varying the threshold τ to see the effect this has on the result.

Calculate the mean intensity of the image

$$\frac{1}{m \cdot n} \sum_{i=1}^{m} \sum_{j=1}^{n} im[i,j]$$

where *m* and *n* are the number of rows and columns of the matrix.

HINT: To encode a summation $c = \sum_{j=1}^{n} x[j]$ you can simply use a for loop such as the below, or you can use the *sum* function. In this case, you could use the *mean* function.

```
c = 0;
for j = 1:n
    c = c + x(j);
end
```

There is a rule stating that the should be equal to the sum of their convolution values. I.e.

$$\sum (ker * im) = \left(\sum ker\right) \cdot \left(\sum im\right)$$

Calculate the mean intensity of the kernel *ker* and convolved image *cim*. Do these values agree with the above rule? Why / why not?

Laplacian of Gaussian

The Laplacian of Gaussian filter approach can be very effective in identifying edges and multiple resolutions.

The filter, in 2-dimensions, is defined as

$$LoG(x,y) = -\frac{1}{\pi\sigma^4}\left(1 - \frac{x^2+y^2}{2\sigma^2}\right)e^{-\frac{x^2+y^2}{2\sigma^2}}$$

where x, y are coordinates, which can be interpreted as the row and column numbers of an image, σ is the standard deviation of the Gaussian, $e \approx 2.718$ is the exponential constant and $\pi \approx 3.142$.

In order to use this, we must discretise the function. We will use $\sigma = 1.4$ and create a 9×9 kernel matrix with $x = -4..4$ and $y = -4..4$.

Create a mesh grid to represent the coordinates on which the equation will be evaluated:

```
ls = linspace(-4,4,9);
[X,Y] = meshgrid(ls,ls);
```

Then we can evaluate the LoG function:

```
sigma = 1.4;
LoG = -1/(pi*sigma^4).*(1-(X.^2+Y.^2)/(2*sigma^2)).*exp(-
(X.^2+Y.^2)/(2*sigma^2) );
```

Plot a surface image of the *LoG* to see what it looks like using the `surf` command.

Try using this *LoG* matrix to carry out a convolution with a grayscale version of the Landscape image and display its absolute value in a figure alongside the original image.

Try doing the same with some other images, such as the InfoLab image. How does this compare with the examples we saw in week 1?

2: Classification with K-NN

In this section, we will train a k-nearest neighbour model to classify the flowers in the Fisher Iris dataset.

2.1: Load and visualise the data

First, load the `fisheriris` data file.

Create the features matrix using the first two meaures and the vector of labels using the species information:

```
features = meas(:,1:2);
labels = categorical(species);
```

features is a numeric matrix while labels is a cell array of character vectors.

Create a scatter plot of the data, using the labels to distinguish the species:

```
gscatter(features(:,1), features (:,2), labels,'rgb','osd');
xlabel('Sepal length');
ylabel('Sepal width');
```

What do you observe from this data?

2.2: Train and test k-nearest neighbours

We will use the "fitcknn" command to train a k-nearest neighbour model:

```
knnmodel = fitcknn(features,labels);
```

In order to obtain the labels from the trained model, we use the `predict` function. This takes two arguments: the first should be the trained model and the second should be the set of features that we want to predict the label for. Ordinarily, we would split the dataset into training and testing sets. For this exercise, we will test with the training data and see how well the model works and store the predicted labels in a variable called "predicted":

```
predicted = predict(knnmodel, features);
```

Note that, alternatively, we could have used the `resubPredict(model)` function to automatically resubmit the features of the training data to the model for prediction.

2.3: Evaluation of Performance

Now that we have predictions, we want to evaluate the performance of our model, i.e. we compare our predictions with the correct "ground-truth" labels.

One approach to this is to calculate the proportion of correctly-predicted labels to the total number of observations.

```
correct_predictions = sum(labels == predicted)
accuracy = correct_predictions /size(labels,1)
```

How accurate is the model? Is this the result that you expected?

In this type of task, we are also interested in the misclassification error (the proportion of misclassified observations) on the training set, which is the value we hope to minimise. We can compute the resubstitution error with the with the `resubLoss()` function:

```
knn_resub_err = resubLoss(knnmodel)
```

What the performance loss of the knnmodel?

2.4: Confusion Matrix

To get more insight into the results of our model, we can also compute and plot the confusion matrix. In the confusion matrix, we list the true class down the rows and predicted class across the columns. The value in row *i* and column *j* represents the number of elements predicted to be class *j* that are really class *i*. We can use the `confusionchart()` function to view confusion matrix

```
knnmodelCM = confusionchart(labels,predicted)
```

Given the confusion matrix:

1. Which species are predicted well by the model?
2. Where does the model make mistakes?

2.5: Separate training and testing data

So far, we have looked at the performance by training and evaluating our models on the same dataset. In practice, this will lead to over-fitting and underestimating the actual error on unseen data. Ordinarily, we would split our data in training and testing subsets. We will use the training data to train our model and the unseen testing data to evaluate our model.

Let's start by creating training and testing sets of the features and labels. We will use 80% of the data for training, and the remaining 20% for testing. You can use the below or create your own splits of the data.

```
trfeatures = features([1:40,51:90,101:140],:);
tefeatures = features([41:50,91:100,141:150],:);
trlabels = labels([1:40,51:90,101:140]);
telabels = labels([41:50,91:100,141:150]);
```

Now, we will retrain our model using the training data and use this to predict the labels for the testing data:

```
Knnmodel2 = fitcknn(trfeatures,trlabels);
predicted = predict(knnmodel2, tefeatures);
```

Calculate the accuracy and the confusion matrix of this testing data:

```
correct_predictions = sum(telabels == predicted)
accuracy = correct_predictions /size(telabels,1)
knnmodelCM = confusionchart(telabels,predicted)
```

What difference do you observe now that the testing data is different from the training data?

2.6: Create your own KNN code

Now, we'll code KNN for ourselves. Start by loading the data, selecting the features and the labels:

```
load fisheriris
features = meas(:,1:2);
labels = categorical(species);
```

Setup training and testing features and labels:

```
trind = [1:40,51:90,101:140];
teind = [41:50,91:100,141:150];
trfeatures = features(trind,:);
tefeatures = features(teind,:);
trlabels = labels(trind);
telabels = labels(teind);
```

Create an empty categorical array to store the prediction output:

```
tepredict = categorical.empty(size(tefeatures,1),0);
```

Set the parameter k. NB: we will use $k = 1$ here but we could use another value:

Check each testing item using the index i with a for loop

```
for i = 1:size(tefeatures,1)
```

Get the training features and the test features for the current item. We will need to use the `repmat` function to create a replicating matrix to ensure that each training item is compared with the current testing item:

```
    comp1 = trfeatures;
    comp2 = repmat(tefeatures(i,:),[size(trfeatures,1),1]);
```

Calculate the distance using the L_2 measure:

```
    l2 = sum((comp1-comp2).^2,2);
```

Get the indices of the k smallest L_2 values:

```
    [~,ind] = sort(l2);
    ind = ind(1:k);
```

Get the labels for the testing data:

```
    labs = trlabels(ind);
    tepredict(i,1) = mode(labs);
```

Close the *for* loop:

```
end
```

Calculate accuracy and produce the confusion matrix

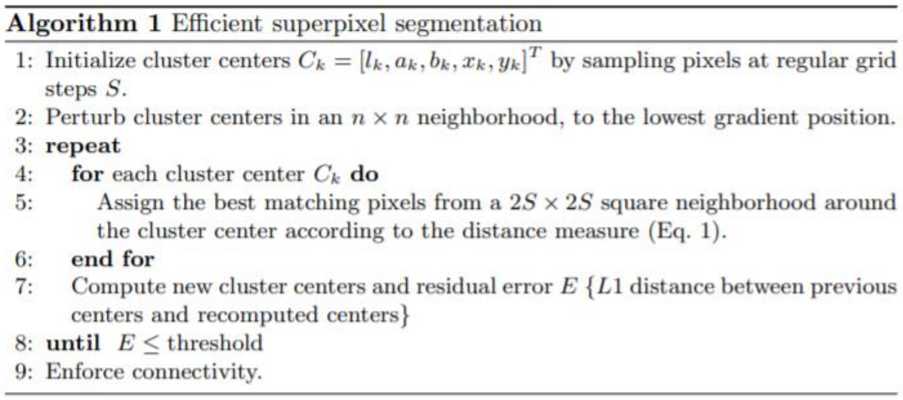
```
correct_predictions = sum(telabels==tepredict);
accuracy = correct_predictions /size(telabels,1);
confusionchart(telabels,tepredict);
```

Do the results from your model agree with those from the matlab model? You could now try varying k or the distance measure to see the impact on the results.

3: Clustering

In this example, we will look at an example of data clustering via image pixel clustering. We will use a technique called SLIC (Achanta R, Shaji A, Smith K, Lucchi A, Fua P, Süsstrunk S. Slic superpixels. 2010). This algorithm (Algorithm 1 below) begins with a regular grid of *m* squares and progressively deforms this to bring the grid boundaries closer to perceived edges in the image.

Since this is a fairly complex task, I have given most of the code for achieving it. Try writing the code yourself based on the description of the problem, and refer to the code if you get stuck.



Algorithm 1 Efficient superpixel segmentation

```
1: Initialize cluster centers $C_k = [l_k, a_k, b_k, x_k, y_k]^T$ by sampling pixels at regular grid steps $S$.
2: Perturb cluster centers in an $n \times n$ neighborhood, to the lowest gradient position.
3: repeat
4:     for each cluster center $C_k$ do
5:         Assign the best matching pixels from a $2S \times 2S$ square neighborhood around the cluster center according to the distance measure (Eq. 1).
6:     end for
7:     Compute new cluster centers and residual error $E$ {$L1$ distance between previous centers and recomputed centers}
8: until $E \leq$ threshold
9: Enforce connectivity.
```

The distance measure D_S is defined as:

$$
\begin{aligned}
d_{lab} &= \sqrt{(l_k - l_i)^2 + (a_k - a_i)^2 + (b_k - b_i)^2} \\
d_{xy} &= \sqrt{(x_k - x_i)^2 + (y_k - y_i)^2} \\
D_s &= d_{lab} + \frac{m}{S} d_{xy}
\end{aligned}
\qquad (1)
$$

where *S* is the grid interval and *m* is a **compactness** parameter that we can tweak.

MATLAB has a function to carry out this algorithm for us:

Import the “Dog.jpg” image to a variable called *im*, convert to double and divide by 255;

Obtain the superpixels with a target initialised number of 50 squares:

```
[L,N] = superpixels(im,50);
```

The value *N* is the number of superpixels resulting from the operation and *L* is a matrix of superpixels. Try displating *L* in an image. What do you think this represents?

The result is more intuitive if the boundaries are displayed as an overlay on the image:

```
figure(2)
BW = boundarymask(L);
imagesc(imoverlay(im,BW,'cyan'));
```

You should see that the image is now broken up by contours, some of which lie on a boundary, and some of which do not.

Create an image where the superpixel value is replaced with the mean image intensity in that region for each colour channel, and display this.

```
im1 = im(:,:,1); bim1 = 0*im1;
im2 = im(:,:,2); bim2 = 0*im2;
im3 = im(:,:,3); bim3 = 0*im3;
for i = 1:N
    ind = L == i;
    bim1(ind) = mean(im1(ind));
    bim2(ind) = mean(im2(ind));
    bim3(ind) = mean(im3(ind));
end
bim = cat(3,bim1,bim2,bim3);
figure(3)
imagesc(bim), axis image off
```

Let’s try to segment the dog using the found superpixels and thresholding of the mean intensity values. We’ll use the following rule with a threshold value of 0.8.

$$ind = \begin{cases} 1 \text{ if } dis \le 0.8 \\ 0 \text{ otherwise} \end{cases} \qquad \text{where} \qquad dis = \sqrt{(bim_r - 0)^2 + \left(bim_g - 0\right)^2 + (bim_b - 0)^2}$$

and bim_r, bim_g and bim_b are the red, green and blue colour channels of the mean intensity image that we created earlier. What do you think is the logic behind the 0 values?

```
ind = sqrt(sum(bim.^2,3))<0.8;
figure(4), colormap gray
imagesc(ind), axis image off
```

Use this map to extract the dog from the background and vice versa:

```
ind2 = cat(3,ind,ind,ind);
figure(5)
subplot(121), imagesc(im.*ind2), axis image off
subplot(122), imagesc(im.*(1-ind2)), axis image off
```

It can also be useful to display the image with the contour map:

```
figure(6)
imagesc(im), axis image off
hold on
contour(ind,'b')
hold off
```

Try this out with some other images, and try experimenting with different parameters. What happens if you change the compactness *m*? What is the effect if you change the zero values in the distance equation above?