ME4 Machine Learning - Tutorial 8

In Lecture 8 we looked at decision trees and random forests to classify data, then also discussed preprocessing. In this tutorial you will put this into practice, using Python and Scikit-learn to test out the nonmetric methods for a set of classification data. You will then implement some preprocessing tools.

1 Decision tree classifier

Generate a dataset with

```
from sklearn import datasets
...
X, y = datasets.make_classification(n_samples=100, n_features=2,
                                    n_informative=2, n_redundant=0)
```

This will generate a 2 class problem. Use the built in decision tree classifier:

```
from sklearn import tree
...
clf = tree.DecisionTreeClassifier()
```

This classifier uses the same form for the fit() function as the other scikit-learn tools throughout this course – use this to fit the data. Sample the prediction function clf.predict() across the grid, with 200 points from -4 to +4, as well as a scatter plot of the data on top. This should match what you have done in earlier tutorials, so reuse code from that. Plot the result. How sensible do you think this classification approach is? What do you think the advantages and disadvantages may be relative to other techniques such as SVM or linear discriminant functions?

2 Random forest classifier

As discussed in the lectures, the random forest classifier generates several trees, classifying on random parameters, and these all take a vote at the end to classify each test point. Set up a random forest with:

```
from sklearn import ensemble

rf = ensemble.RandomForestClassifier(...)
```

You will need to look up the documentation for RandomForestClassifier and set 100 trees (estimators) with a maximum depth of 20.

The function rf.predict_proba() will output probability for the given samples. It will output an array of shape [n_samples, 2], giving probability for each of the two classes (note that these sum to 1, so technically it is redundant to have both). Select the probability for the first class and plot this. Code is at the end of the sheet which can help with this if you do get stuck.

What do you think of the random forest classification? What do you think would happen if you increase the number of trees? Try this. What about if you decrease? Try increasing and decreasing the number of samples in your training dataset too, and see how these affect the result.

3 Data processing

We will use the image window.png to do some data processing. If using Colab, you should upload it first with

```
from google.colab import files
print('Upload window.png here')
uploaded = files.upload()
```

You can also upload the file using the colab file manager – click the folder logo on the left hand side and you can drag and drop the file in. Note that you will need to connect to a remote machine first.

You can load this image in, do some processing and view it with

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.image as mpimg

#load in the image
img = mpimg.imread('window.png')

#just extract one channel - technically this is red, but
#it doesn't matter since they are all equal
imgBw = np.squeeze(img[:, :, 0])

#get the size
npy, npx = imgBw.shape

plt.figure()
plt.imshow(imgBw, cmap=plt.cm.gray)
plt.axis('off')
```

Note that the image loaded in consists of three channels, for red, green and blue. We just extract one of these channels (and use ‘squeeze’ to eliminate the third dimension of the array) because for this black and white image the other channels are redundant.

Also note that npy is the size of the first dimension and npx is the size of the second when we extract the image dimensions; Python (like Matlab) represents images as a matrix, and matrix notation indicates the row (y) first, then the column second (x).

3.1 Fourier processing

Plot the 2D image spectrum of this. You will need to use np.fft.fft2(), and plot the absolute of the result with np.abs() because the Fourier transform output is a complex number:

```
fim = np.fft.fft2(imgBw)

plt.figure()
plt.imshow(np.abs(fim))
#show the colour scale:
plt.colorbar()
#set the colour limits to something so we can see the image better
plt.clim(0, 1e3)
```

By default the frequency component for (0,0) will be in the corner, but the function np.fft.fftshift() will shift the image such that this is in the centre, which is typically desired – add this to your plot when you show the image.

Use the following code to set all the values other than 60 points either side of (0,0) in x to zero (note that Fourier transforms 'wrap around', so pixels less than zero are taken from the other end of the array).

```
#make a separate copy of the image to work on:
fim2 = fim.copy()
#set everything in all except the first row to zero
fim2[1:npy, :] = 0
#set everything in the first row to zero except the first 61 points and the last 60
fim2[0, 61:npx - 60] = 0
```

Fourier transform this back with np.fft.ifft2(), and plot the resulting image - note, you may have to apply np.real() around your ifft2() to force the numbers to be real (and not complex) numbers.

Now do the same process, extracting a slice in the y direction, and plot that.

Think about how this could be useful in practice for machine learning. In this case we are looking at 2D data - how could Fourier transforms be useful for 1D data too? What about in higher dimensional space?

3.2 Edge detection

We will follow the edge detection approach developed in the lectures. Use np.diff() to differentiate, setting axis=1 for x and axis=0 for y:

```
diffx = np.diff(imgBw, axis=1)
diffy = np.diff(imgBw, axis=0)
```

```
diffx = diffx[0:npy - 1,:]
diffy = diffy[:,0:npx - 1]
```

Note that the last two lines reduce the size of each differential result - have a think about why this is necessary [hint: sketch out a 3x3 matrix and see what the shape would be if you subtract each column from the next]. Combine diffx and diffy together using a square root of the sum of the squares - useful functions are np.sqrt() and np.power(). Plot the resulting image of the edges.

3.3 Thresholding

We will now apply a threshold to the edge image to produce a binary map of which points are edges and which are not. Add a colour scale 'plt.colorbar()' to your edge plot from before so you get an idea of the values. Select what you think may be a suitable value to apply the threshold at - call this 't'. Assuming your previous edge image is in the array 'edgeIm' you can generate the threshold image by the following code:

```
edgeThresh = (edgeIm > t).astype('int')
```

Note that you need to use the 'astype('int')' to convert from boolean to integer numbers to enable it to be plotted. Plot the image as before. What do you think about your threshold selection? Try increasing and decreasing it. What happens if the threshold is too low? What happens if it is too high?

4 Code

For part 2:

```
YgridRf = rf.predict_proba(Xgrid)
YgridRf = np.reshape(YgridRf[:, 0], [npx, npy])
```