ME4 Machine Learning - Tutorial 1

In lecture 1 we covered some of the machine learning basics. The purpose of this tutorial is to build familiarity with Python and some of the Python libraries relevant to machine learning, and you will also attempt to manually define a decision boundary for an artificial dataset. This includes using some plotting tools and doing some matrix manipulation (note you may need to revise rotation matrices!). If you are not familiar with covariance matrices, the first lecture and notes should discuss these. This should be useful background knowledge which will help with the course.

While care has been taken to link tutorial exercises to the real world as much as possible, in some cases it is a challenge to do this directly. You may find that the data is artificial, or you are practising concepts which may not appear useful, however, the intention of all this work is to reinforce what you are learning in the lectures.

You are advised to use Jupyter notebooks available within Anaconda, which is installed directly on the Imperial systems in the computer room. This should have the necessary packages installed. There is a guide on blackboard on how to get started with this, but to quickly summarise: run Anaconda Navigator, then select Jupyter Notebooks (note that there are other notebook options) then Launch. You can then select “New” and “Python 3” to get a new notebook open.

You may use any alternative approach you wish, however, we may not be able to provide support for this. You should also not expect tutors to provide technical support for installations. Note that the intention is that the exam will use the setup described above so you may wish to become familiar with this.

The intention is for you to learn how to do things with Python and the associated libraries. There is a huge amount online to help you with Python - in particular the official documentation from scikit-learn is very useful to get a good idea about what the library functions are capable of. I have put a video at https://youtu.be/GMexl8U2T3o which should take you through the process of looking up documentation for scikit-learn and I would encourage you to have a look at this. Do make use of Google and other tools to learn. However, please do not just blindly copy sections of code from other websites, even if they are doing very similar things; the intention is for you to learn and you will not do this if you just copy.

Note that due to issues coping and pasting code from PDFs, separate text files are available for each tutorial which contain the relevant code snippets. You are advised to use these.

Also note that throughout these tutorials, we use the word ‘function’ to refer to the operations available in the libraries. Technically the majority of these are ‘methods’ in that they belong to classes in an object orientated framework, however, we utilise them primarily in the traditional ‘function’ manner.

1 Generate and plot a test dataset

Set the random seed to zero (this keeps the 'random' numbers generated the same for everyone, and makes sure the code will behave the same each time you run the script). Do this for all tutorials on the course unless otherwise specified. You will need to do this via

```
np.random.seed(0)
```

assuming that you have included 'import numpy as np' at the top of your script. You can look online to find out more about how to set the random seed using numpy if you wish.

Begin by making a test dataset which we will then use to practice plotting, which will be useful in later tutorials. To do this we will use the function sklearn.datasets.make_classification(). At the top of your script you will need

```
from sklearn import datasets
```

then to use this function to set the feature vectors and the classification vector for the test data:

```
X, y = datasets.make_classification(...)
```

You will need to fill in the arguments for the function yourself (look this up online!). Set it to produce two features in your dataset, of which two are informative and zero redundant, with 100 samples. Please note that you need to follow these exact numbers otherwise you will end up with a different distribution and other parts of the tutorial won't work. Note that there are two outputs: X is a matrix containing the samples themselves, and y is a vector giving a label for each sample in the X vector. As an example of the output for two features and five samples, you could have

$$X = \begin{array}{cc} 0.34 & 0.21 \\ 0.87 & 0.24 \\ 0.24 & 0.62 \\ 0.72 & 0.36 \\ 0.81 & 0.16 \end{array}$$

and

$$y = \begin{array}{c} 1 \\ 0 \\ 0 \\ 1 \\ 0 \end{array}$$

so the first and fourth samples (parameter 1 = 0.34, parameter 2 = 0.21 and parameter 1 = 0.72, parameter 2 = 0.36) both belong to class 1 and the rest belong to class 0. Redundant features are features which have no influence on the class of the data; as an example, if trying to predict the score for a Premier League match you would consider the temperature in Kyoto not to have any influence, and having this temperature included as a column in X would therefore be a redundant feature. You can access the first column/feature with X[:, 0] and the second with X[:, 1]. It is important to understand this structure of data, because much of the training and testing data available on the course will be available in this form. Similarly you can identify the nth dataset with X[n,:] and see the corresponding category label y[n].

Both features will be scaled to standard axes when output from the function (typically mean 0 and standard deviation around 1, although this is not guaranteed). We want the first feature to be defect length in mm, so we want to halve the standard deviation and shift it to be centred at 5mm. The second one will be in brightness (out of 256 colours) so we scale up by 30 and add a mean of 160. This effectively means that the first column is multiplied by 0.5 then 5 is added, and the second is multiplied by 30 and has 160 added[1]. We will also take the absolute to avoid unphysical negative numbers appearing. Write Python code to do this:

```
X[:, 0] = np.abs(X[:, 0] * 0.5 + 5)
X[:, 1] = np.abs(X[:, 1] * 30 + 160)
```

Using matplotlib, generate a scatter plot, using the classification vector y to set the colour of the points in each set. Start with

```
import matplotlib.pyplot as plt
fig, ax = plt.subplots()
```

which will then set up the axes, which are needed for ax.scatter().

As with Matlab, you can select a subsection of an array using boolean operations. For example, all the rows in X where y is equal to zero can be found with X[y == 0, :]. Taking the example data from above, since y==0 for the second, third and fifth values, this will select the values of X from the second, third and fifth rows, making

$$X[y == 0, :] = \begin{matrix} 0.87 & 0.24 \\ 0.24 & 0.62 \\ 0.81 & 0.16 \end{matrix} .$$

You can select which column with the second parameter in the square brackets (currently ':'). Therefore, to plot the first class, the command is

```
ax.scatter(X[y == 0, 0], X[y == 0, 1])
```

which gives the first column of X as the x axis values and the second column of X as the y axis values (note potential for confusion over x and y axes vs common definition of X and y as input and output parameters for ML models - do not get muddled up with this!)

Note that if using an IDE (e.g. pycharm, spyder) you will need to add plt.show() in order to see the final result at the end (this is not necessary in Jupyter notebooks). Add a follow up command to plot the second class, where y == 1; note that if you do not specify colours, Python will automatically use a different colour each time the 'scatter' command is run.

[1] Note that these are arbitrary values chosen to give some representative outputs rather than having any specific justification behind them.

We can define a curve of $x_2 = -280x_1 + 1400$ in an attempt to separate the two sets. Plot this. To do this, you will need to define suitable x values; something like 'x1 = np.linspace(A, B, C)' could be useful for this, where A and B define the upper and lower values in the range and C defines the number of points between them. Then you should calculate x2 and use the command 'plt.plot(x1, x2)' to plot the final line. Make sure your axes are suitable scaled in the final plot - matplotlib's xlim() and ylim() commands will help with this.

What do you think of the separation line? Now spend a few minutes adjusting the parameters manually to improve the fit of the line to separate the data (note that you don't need to add extra terms, e.g quadratic ones, just adjust the coefficients 280 and 1400). Congratulations, you've defined your first classification algorithm!

2 Make a function to generate a suitable covariance matrix

Covariance matrices are important for defining how parameters in the model vary with each other, and are a critical thing to understand for machine learning. See the notes for more information about these. In this section we will define a covariance matrix between two parameters by identifying standard deviations in two parameters, then applying a rotation to this to introduce some covariance between the two. While this will not be a particularly common occurrence in real datasets (try to think of an example if you can!), this approach is convenient for us because it gives good practice getting started with Python and numpy, as well as being a useful demonstrator distribution that we will use later in the course – specifically for principal component analysis.

Use the function below to generate an arbitrary covariance matrix with a rotation, and fill in the rotation matrix:

```python
import numpy as np

def get_cov(sdx=1, sdy=1, rotangdeg=0):
    covar = np.array([[sdx**2, 0], [0, sdy**2]])
    rot_ang = rotangdeg / 360 * 2 * np.pi
    rot_mat = #... you need to generate your own rotation matrix here

    covar = np.matmul(np.matmul(rot_mat, covar), rot_mat.T)
    return covar
```

Add this function to your script (note that you don't need to import numpy twice if you already have it), and fill in what you think the rotation matrix should be – np.sin() and np.cos() will be handy functions. Note that to rotate a matrix $\boldsymbol{A}$ to $\boldsymbol{A}'$ with a rotation matrix $\boldsymbol{R}$ we are using the form $\boldsymbol{A}' = \boldsymbol{R}\boldsymbol{A}\boldsymbol{R}^T$; this is a fairly standard mathematical approach. You can find more online or it is probably covered elsewhere on your degree.

Some background, in case you are interested: while generating the on-diagonal terms of a covariance matrix is generally straightforward, since these relate easily to the standard deviation, it is not so easy to do the same for the off-diagonal, covariance terms. Their values have to lie in certain ranges. Therefore we instead choose this approach - define a matrix with independent parameters x_1 and x_2 then rotate the distribution, such that we get some dependency between the values (assuming that the variance is different in each direction).

We will now try to understand what this distribution looks like. First of all, we will plot the probability density function in 2D. Then we will take some samples from this distribution and generate a scatter plot of this. The distribution is given as

$$p(\boldsymbol{x}) = \frac{1}{(2\pi)^{d/2} |\boldsymbol{\Sigma}|^{1/2}} \exp\left[-\frac{1}{2}(\boldsymbol{x}-\boldsymbol{\mu})^t \boldsymbol{\Sigma^{-1}} (\boldsymbol{x}-\boldsymbol{\mu})\right].$$

Note that in this, $\boldsymbol{\Sigma}$ is the covariance matrix and the symbol does not indicate a summation. Also note that d is the number of dimensions (parameters), which is 2 in this case, so we can drop the $d/2$ power.

Define a grid of 200 $\times$ 200 points at which we will calculate the distribution. The initial step with this is to define the coordinate values for x and y separately, which you can do between -1 and 1 with 200 points as follows:

```
x1line = np.linspace(-1, 1, 200)
x2line = np.linspace(-1, 1, 200)
```

We want to calculate the function at every point in this grid, i.e. all of $200 \times 200 = 40000$ points. This means that we need to generate x and y values to correspond to every single point, rather than just a single line in each dimension. The function meshgrid() within numpy will do this:

```
x1grid, x2grid = np.meshgrid(x1line, x2line)
```

meshgrid() will take two vectors and generate two vectors containing all the combinations. The lecture will have covered meshgrid, and you can read more in the course notes. Now we wish to sample our feature space based on this, and this means that these x and y values must be combined into a single X array (of size (2, 200, 200)) and reshaped to give a numpy array of size (2, 40000):

```
Xgrid = np.array([x1grid, x2grid]).reshape([2,40000]).T
```

We have also transposed (.T) to make the feature the second dimension.

A common question is why can we not just directly reshape to [40000, 2]. To understand this, think about how data is stored in computer memory. All data is stored in a linear manner, but we store a matrix by going along each row in turn (this is a 'row major' format which is used by numpy - other languages may do other things). Therefore if you provide a vector $\begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 \end{pmatrix}^T$ and reshape it into a 3×2 matrix, it will become $\begin{pmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{pmatrix}$. This is different from putting it into a 2×3 matrix $\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$ and transposing, which will generate $\begin{pmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{pmatrix}$. In the case above, we are reshaping the last two dimensions of 200×200 into to a single one of 40000 - this ordering will then be maintained through the processing of X until it is reshaped back into 200×200 later.

By sampling at each of these points, we will have recorded values of a 2D function across the defined space. You may wish to put this into a complete function:

```python
def gen_sample_grid(npx=200, npy=200, limit=1):
    x1line = np.linspace(-limit, limit, npx)
    x2line = np.linspace(-limit, limit, npy)
    x1grid, x2grid = np.meshgrid(x1line, x2line)
    Xgrid = np.array([x1grid, x2grid]).reshape([2,npx*npy]).T
    return Xgrid,x1line,x2line
```

This will be useful in later tutorials so you may wish to copy and paste it in future.

Produce a covariance matrix of standard deviation 1 in x and 0.3 in y, then rotated around 30 degrees, using the function from earlier. We then wish to calculate the probability density function given above, for all values of x given in Xgrid. One option would be to utilise a for loop to loop through every sample, however, this would be inefficient, so we prefer to utilise matrix multiplication functions instead. Knowing or understanding all these steps is not essential for the course, however, the next paragraph tries to give an explanation of them. You may skip this and just use the code provided below if you wish.

We have $\mathbf{\Sigma}$ defined already; its inverse, expressed as $\mathbf{\Sigma^{-1}}$, can be calculated in Python with np.linalg.inv(covar). We are only interested in understanding the distribution, so we set the mean to zero $\boldsymbol{\mu} = [0, 0]^T$and hence effectively just ignore it. For a single $\boldsymbol{x}$ point, we would do the matrix multiplication as $\begin{pmatrix} x_1 & x_2 \end{pmatrix} \begin{pmatrix} \Sigma_1^{-1} & \Sigma_2^{-1} \\ \Sigma_3^{-1} & \Sigma_4^{-1} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}$. However, we want to keep an extra dimension for the 40000 samples we wish to do the multiplication for, and this complicates the situation. Considering the first multiplication of $\boldsymbol{X}\mathbf{\Sigma^{-1}}$, it is straightforward to do straight on Xgrid: np.matmul(Xgrid,np.linalg.inv(covar)) - this does $\begin{pmatrix} x_{1,1} & x_{1,2} \\ \vdots & \vdots \\ x_{40000,1} & x_{40000,2} \end{pmatrix} \begin{pmatrix} \Sigma_1^{-1} & \Sigma_2^{-1} \\ \Sigma_3^{-1} & \Sigma_4^{-1} \end{pmatrix}$ and the output will be a 40000×2 matrix, where each row is the output of the individual matrix multiplication for each of the samples. Each of these then needs to be matrix-multiplied by its corresponding value in Xgrid, as the second multiplication stage. There are potentially a variety of methods to achieve this. The approach used here is to do an element-by-element multiplication (achieved just with *) with Xgrid, then a summation along the last dimension (.sum(-1), with -1 indicating that the dimension to be summed in is the last one). This leads the whole matrix multiplication to be (np.matmul(Xgrid,np.linalg.inv(covar)) * Xgrid).sum(-1). We then multiply by -1/2 and take the exponential. For the terms outside, np.linalg.det() gives the determinant $|\mathbf{\Sigma}|$, np.sqrt() gives the square root and np.pi is used for π, so the constants become 1 / (2 * np.pi * np.sqrt(np.linalg.det(covar))).

This leads to the following code to calculate the probability density function for this distribution:

```
p = 1 / (2 * np.pi * np.sqrt(np.linalg.det(covar))) * np.exp(
    -1 / 2 * (np.matmul(Xgrid, np.linalg.inv(covar)) * Xgrid).sum(-1))
```

This will result in a single vector length 40000, which can then be reshaped into a grid (size 200 x 200, using the function np.reshape()), before being plotted with matplotlib.pyplot.contourf. In contourf, set the first two arguments to x1line and x2line so that these can form the grid points in each dimension - otherwise the distribution will just use the pixel values to do this plotting (i.e. 1 to 200 in each dimension).

Then use np.random.multivariate_normal to generate 100 values from this distribution and plot these on a separate figure (use a mean of 0 for each dimension, i.e. you will need to pass a vector of two zeros, and size can just be set to 100). Compare to the distribution – you may wish to set the limits to -1, +1 on each axis for this. Do you think these represent the underlying distribution well? Try to adjust your code so you can see whether the distribution looks clearer from the points.

3 Generate a circular distribution

If you wish to have more practice with Python, you can try to generate and plot a different distribution. Note that this case is more challenging and you will have to look up various functions yourselves – less assistance is provided for this. We wish to generate a dataset consisting of two classes, where the classes are far from linearly separable. Class two will largely lie within a ring, and class one will be both inside the ring and outside it. The distributions are defined in polar coordinates as uniform around the angle theta and three normal distributions in the radius, two of which are combined for class one.

Define class 2 as 200 points with a radius having a normal distribution centred around 5 with a standard deviation of 1. Class one should be 100 points with a radius centred at 2, and a standard deviation of 1, combined with 400 points centred at 8, also with a standard deviation of 1. Theta values should be made uniform from 0 to 2π in all cases. Produce a single scatter plot showing these two distributions together. What do you think the challenges are in separating these? Could you adjust your generation code to make it easier to do this separation?