ME4 Machine Learning - Tutorial 3

Lecture 3 introduced the concept of regression/curve fitting. In this tutorial you will actually implement this using Python. You will firstly do this manually, implementing your own code for a simple case as shown in the lecture, then you will use the library from scikit learn. For loading data, you will use the library Pandas too.

Note: this tutorial has parts which must be submitted as coursework. While for most of the tutorial content you are encouraged to collaborate and learn from each other, sharing answers to these assessed sections will constitute plagiarism and be treated accordingly. The tutors have been instructed not to provide help with these questions, however, you may ask generic questions to help your overall understanding.

1 Standard linear regression

1.1 Import dataset using Pandas

Pandas is a python library for managing large datasets – it is very valuable to have some familiarity with it. We will use Pandas to import some data into Python to use in our regression. You will need to use the file xray.csv – for convenience this has been uploaded to some webspace at http://pogo.software/me4ml/xray.csv. You can download this, but with Colab, it is easiest just to load from the URL from your Python script. The code needed to load in the csv (comma separated value) file is given as:

```
import pandas

df = pandas.read_csv('http://pogo.software/me4ml/xray.csv')

x = np.array(df['Distance␣(mm)'][:])
y = np.array(df['Total␣absorption'][:])
```

Note that 'Distance' is followed by a space, not an underscore; this is not rendered particularly clearly. Also note that it is assumed that you have imported numpy as np – we will not explicitly mention all imports for basic libraries throughout these tutorials.

Pandas is very flexible and has broader capabilities than this, but this is all we will use for the time being. x and y now contain the raw data we wish to fit a line to.

Some background (if you are interested)

For simplicity, X-rays are often considered to have linear attenuation behaviour, which means that if the X-ray travels through a material a distance d, it will lose a particular fraction of its power, and if it travels through another distance d then it will lose the same fraction. This results in an exponential decay as the X-ray propagates. This is very convenient in many applications, particularly X-ray CT image generation, which relies on exactly this assumption. In practice, there is a phenomenon called 'beam hardening' which means that as an X-ray passes through an object, the beam becomes 'harder', i.e. more difficult to attenuate, so the attenuation value reduces. This means the behaviour will no longer be linear. We can try to address the challenges by finding a simple polynomial relationship, which is where regression becomes valuable.

1.2 Perform linear regression [part of coursework submission]

Using the matrix formulation from the lecture (also in Chapter 3 of the notes), perform a linear regression of this data. Plot the scattered points in the default colour (this is a blue colour), and plot the line fitted to them in black. Use numpy.linalg.solve() to invert the matrix. Note that to square a numpy array you can do x ** 2, and to get the sum of the squares you can then do np.sum(x ** 2). len(x) will give you the length of x, and np.dot(x, y) will give you the dot product between two vectors (multiplying each of the array elements between x and y then summing the result up for all elements). To plot the line, generate x values between 0 and 6 with 200 values, and plot the line as a function of these x values, in the same way as in earlier tutorials.

2 Higher order regression [part of coursework submission]

To fit a quadratic function, form $f(x) = \beta_1 + \beta_2 x + \beta_3 x^2$ to the data (hint: the notes contain the equations for this). Extend your code to plot the resulting curve from this on the same graph as before in red. Add your name behind the plotted items using the following function:

```
plt.text(0.5,100,'Your name here',size=20,zorder=0.,color='#aaaaaa')
```

The intention of adding your name is to ensure that everyone has a unique submission rather than the name being completely visible. IT IS INTENDED THAT THE DATA POINTS WILL BE ON TOP OF YOUR NAME. Please do not adjust the parameters (positioning etc.). Your name is recorded by blackboard when you submit anyway so it does not need to be legible.

Save the image, containing the scattered points in blue, the black linear fitted line, and the quadratic fitted line in red on a single plot, with your name behind the points. If using Colab, you can save the image locally just by right-clicking. This is part (A) of the submission. Note that we will not assess graph 'correctness' such as whether the axes are labelled; rather we are trying to establish that you have understood and been able to produce the plot requested. It is a requirement of part A that you include your name using the code above.

For part (B), answer the following questions concisely (target 1 line for each answer, and no more than 2 lines – any text more than this will not be read. No need to use full sentences providing what you are saying is clear.):

1. Which line (linear or quadratic) will have the lower bias and why?
2. While line will have the lower variance and why?
3. How would you decide which of the two curves was best?

Submit the following parts: your image of the fitting to the scattered points (A), and your answers to the questions (B). Place these all into a single document containing your name in the filename, and your Imperial username (e.g. abc15 -- not your CID) at the top of the first page, with all responses marked with the labels (A)-(B), and submit this as a PDF (not Word!). Please make sure you follow these instructions – we reserve the right to deduct marks if these are not followed.

3 Using Scikit-learn to perform regression

Plastic water pipes are gaining increasing use within the nuclear power sector. However, the joints are difficult to inspect and ultrasonic techniques, typically developed for steel or other metals, rarely work well. To develop ultrasonic techniques further, we need to have a model for how the ultrasonic wave speed varies with frequency and temperature.

Some background (if you are interested)

Most ultrasonic inspection in industry is undertaken in steel components, simply because steel is such a common material. Ultrasound inspection of plastic (specifically HDPE) is significantly more complicated than steel, for a couple of reasons: attenuation is much higher and the velocity varies significantly. Steel has low attenuation because the material is quite elastic and therefore very little energy is lost from the wave. In plastic, the material is not particularly elastic and therefore waves are attenuated faster. Think about if you squeeze a plastic block, it is less likely to spring back afterwards; this causes waves to lose energy.

The velocity behaviour is also more complicated, in particular being a strong function of temperature. This is because plastic is typically much closer to its melting point than steel is, so the property change for a small increase in temperature is more significant. There may be a temperature gradient through the pipe wall thickness, and this will cause a velocity gradient, which will cause ultrasound beams to deviate and hence indicate defects at incorrect locations. We therefore need to properly understand this behaviour.

We have taken a set of measurements, which can be found in the file http://pogo.software/me4ml/hdpeVel.csv (note that this is from the paper https://asa.scitation.org/doi/10.1121/1.4976689). We will load in this data and perform a regression on it based on frequency and temperature. We will first do this in a linear form.

First of all we need to use Pandas to load in the CSV and extract the data:

```
#read in the CSV file
df = pandas.read_csv('http://pogo.software/me4ml/hdpeVel.csv')
#set the 'index' column as the one containing the temperature values
df = df.set_index('T/C␣f/MHz')
```

```
#extract the frequency values (and scale since they are MHz)
freq = df.columns.values.astype(np.float)*1e6
#extract the temperature values
temp = df.index.values.astype(np.float)
#extract the main part - the velocity values
vel = df.to_numpy()

#calculate the total number of values
tot_values = len(freq)*len(temp)
```

This data is arranged in a grid, with an output value for velocity for every combination of the frequency and temperature inputs. This should be rearranged into the standard form of parameter vector X and an output vector y.

```
x1grid, x2grid = np.meshgrid(freq, temp)
Xgrid = np.concatenate([x1grid.reshape([tot_values, 1]),
          x2grid.reshape([tot_values, 1])], axis=1)
ygrid = vel.reshape([tot_values, 1])
```

numpy.meshgrid is used here to generate gridded combinations of frequency and temperature values, i.e. from the vectors of 5 values for each, repeat them such that x1grid and x2grid have 25 values each. These are then put into vectors and concatenated to form the Xgrid vector. ygrid is then just what we get if we reshape the velocity from its grid into a vector.

Utilise the scikit-learn LinearRegression function:

```
#put at top:
from sklearn.linear_model import LinearRegression

reg = LinearRegression()
reg.fit(Xgrid, ygrid)
```

As with the Bayesian model in the last tutorial, we use the function fit() again to set the necessary coefficients in the model. However, here the function clearly does a very different process. Scikit-learn is very useful because it uses a very standardised interface for a huge range of different methods.

Then use the reg.predict() function based on the training values X to estimate values for the fitted function, in a variable called y_lin. This will match your use of the .predict() function from tutorial 2.

We will do a 3D scatter plot of both the true data (black) and the linear fitted data (red), which is best as an interactive plot you can drag around. If you are using Colab, you can do this with:

```
import plotly.graph_objects as go

fig = go.Figure()
fig.add_trace(go.Scatter3d(x=Xgrid[:, 0], y=Xgrid[:, 1], z=ygrid[:, 0],
                    mode='markers',
                    marker=dict(size=2, color='#000000', symbol='x')))
fig.add_trace(go.Scatter3d(x=Xgrid[:, 0], y=Xgrid[:, 1], z=y_lin[:, 0],
                    mode='markers',
                    marker=dict(size=3, color='#ff0000', symbol='circle')))
```

Otherwise if using a standard IDE you can use this:

```
from mpl_toolkits.mplot3d import Axes3D

fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
ax.scatter(Xgrid[:, 0], Xgrid[:, 1], ygrid, marker='x', color='#000000')
ax.scatter(Xgrid[:, 0], Xgrid[:, 1], y_lin, marker='o', color='#ff0000')
```

You can see that this is quite a good fit, but we do have some curves in the original data that do not match the linear function well, suggesting that introducing higher-order polynomial terms may be useful.

In the lecture (and Sect. 3.2.1 of the notes) we looked at the concept of expressing higher order fitting by adding additional parameters to the data, specifically the example of fitting $f(\boldsymbol{x'}) = \beta_1 + \beta_1 x'_1 + \beta_2 x'_2$ where $x'_1 = x$ and $x'_2 = x^2$ rather than fitting $f(x) = \beta_1 + \beta_2 x + \beta_3 x^2$. This can also be used for cases where there are already two parameters, e.g. we may want to fit $f(\boldsymbol{x}) = \beta_1 + \beta_2 x_1 + \beta_3 x_2 + \beta_4 x_1^2 + \beta_5 x_1 x_2 + \beta_6 x_2^2$, which contains all the combinations of $x_1$and x_2 up to polynomial order 2. The dataset we are considering here has this two parameter form, where velocity is a function of temperature and frequency, and we wish to include the 2nd order polynomial terms to achieve a better fit compared to linear. Instead of using higher powers of $x_1$and x_2, we can define additional parameters to capture these, then do a linear fit with all these parameters together. This function would have the form $f(\boldsymbol{x'}) = \beta_1 x'_1 + \beta_2 x'_2 + \beta_3 x'_3 + \beta_4 x'_4 + \beta_5 x'_5 + \beta_6 x'_6$, where $x'_1 = 1$, $x'_2 = x_1$, $x'_3 = x_2$, $x'_4 = x_1^2$, $x'_5 = x_1 x_2$ and $x'_6 = x_2^2$.

By exploiting this approach, the same LinearRegression function from before can be used, but now with many more parameters. Scikit-learn includes functions to generate the polynomial features without you having to do too much. Add the following code:

```
from sklearn.preprocessing import PolynomialFeatures

poly = PolynomialFeatures(degree=2)
#generate the new feature vector:
X_poly = poly.fit_transform(Xgrid)

#check its shape
print(X_poly.shape)
```

The shape is larger. Does it match what you would expect based on above (you may also wish to check X.shape)? You may also wish to run

```
print(poly.powers_)
```

to confirm the features which have been generated - search online to find out what the output means, and compare this to the description of powers given above.

Having done that, the following can be used to fit:

```
reg_poly = LinearRegression()
reg_poly.fit(X_poly, ygrid)
```

Predict data from this, then plot the resulting data points. If using Plotly in Colab, you can do it in a green colour #00ff00, size 3 and with diamonds as the symbols (symbol='diamond'). Adding a new code cell and including the 'add_trace' command from above for the new data will generate a new plot adding the new data on top of the old data, so you do not need to regenerate the entire plot.

If using PyPlot, you can plot on the same axes in green triangles (marker '^').