DRAFT III COM6509/4509—Assignment Sheet 1
Regression in Python

Neil Lawrence

October 12th, 2012

To get you used to some of the commands before starting the assignment, this provides a brief warm up with Python.

Start Python with pylab enabled. Once you've received the command prompt try the following commands. Do **NOT** copy and paste these commands into the python prompt if you do that you are not learning anything (other than how to copy and paste, which you should know by now). Type these commands using your fingers and the keyboard ... don't touch the mouse (it is evil, it sucks out your soul through the palm of your hand).

Remember to check what a command does, simply type:

```
np.random.randn?
```

You should store your answers to the questions in the assignment in a script file (e.g. assignmentOne.py) which will be submitted at the assignment deadline. Any answer where you are required to write something down will be requested in the assessment Your answers to this assignment will be given in the form of a Python script. This will be submitted before the start of the lab class in Week 7 via MOLE. **Late submissions will get zero** as we will mark the submissions directly after they are received in the lab class. We will try to run your script using

```
%run assignmentOne.py
```

Make sure it answers each of the questions below!

To start the script, you'll need to import some libraries. At the start of your script import the following libraries:

```
import scipy.io
import numpy as np
import matplotlib.pyplot as plt
```

When running `ipython` with the `--pylab` flag, these libraries are automatically imported, but you'll need to manually include them in your script (along with the associated namespaces before the command names, such as `np`).

You need to complete the first part before starting the second (which will make up your assignment mark).

Part One: Lab Class

The first part is really a warm up, to get you familiar with coding in Python and ready to answer the harder questions in the second part.

1. First, create some uniformly spaced input values in an array with 10 rows and 1 column, `x`,

   ```
   x = np.zeros((10, 1))
   x[:, 0] = np.r_[-1:1:10j]
   ```

   You can see what the values are by typing

   ```
   print(x)
   ```

   (a) The function `random.randn()` is gives you some samples from a *standard normal*. What is a standard normal?

   (b) Let's now sample a gradient for a line, `m`, and an offset, `c`

   ```
   m = np.random.randn(1)
   c = np.random.randn(1)
   ```

Have a look at m and c,

```
print(m)
print(c)
```

Now let's make some noise!

```
epsilon = random.randn(x.shape[0], x.shape[1])*0.01
```

What is the standard deviation of this noise? What is its variance?

(c) Now create a data set by taking the inputs, x and multiplying by m and adding c

```
t = m*x + c + epsilon
```

Can you write down the probability density for t? Do this before proceeding.

(d) Plot the small data set you've created using:

```
plt.plot(x, t, 'rx')
```

2. **Iterative Solution** Now we are going to fit a line to the data you've plotted. To do this, we are first of all going to use the formulae we computed in class. We are trying to minimize this error

$$E(m, c, \sigma^2) = \frac{N}{2} \log \sigma^2 + \frac{1}{2\sigma^2} \sum_{i=1}^{N} (t_i - m x_i - c)^2 .$$

with respect to m, c and σ^2.

We need to start with an initial guess for m, the actual value doesn't matter too much, try setting it to zero, then use this formula to set c,

$$c^* = \frac{\sum_{i=1}^{N} (t_i - m^* x_i)}{N},$$

followed by this formula to estimate m. Iterate between the two formulae and compute the error after each iteration, see how much it changes.

$$m^* = \frac{\sum_{i=1}^{N} x_i (t_i - c^*)}{\sum_{i=1}^{N} x_i^2},$$

Finally when the error stops changing update the estimate of the noise variance:

$$\sigma^{2\,*} = \frac{\sum_{i=1}^{N} (t_i - m^* x_i - c^*)^2}{N}.$$

Print the final recorded values for the error, m, c and σ^2.

3. **One Shot Solution** As we discussed in lectures, we don't need to run the iterative algorithm. Since there was a gradient and an offset, we will use the 'trick' of having a basis set containing the data and another basis set containing the 'constant function' (i.e. set to 1). The previous part of used an *iterative* solution

```
Phi = np.hstack((np.ones(x.shape), x))
```

We can use this basis set to learn m and c simultaneously.

The maximum likelihood solution from the lectures for $\mathbf{w}$ was

$$\mathbf{w}^* = \left[\boldsymbol{\Phi}^\top \boldsymbol{\Phi}\right]^{-1} \boldsymbol{\Phi}^\top \mathbf{t}$$

but you should solve the matrix equation

$$\boldsymbol{\Phi}^\top \boldsymbol{\Phi} \mathbf{w} = \boldsymbol{\Phi}^\top \mathbf{t}$$

directly to get the best solution (Hint: try `np.linalg.solve?`) and call the resulting variable `wStar`.

(a) Implement this update and show that w_1^* (the first element of $\mathbf{w}_*$) is equal to c and w_2^* is equal to m.

(b) Plot the data you generated using

```
plt.plot(x, t, 'rx')
```

(c) And now show a plot of the fit you found using. To do this you will create some 'test' data. The test data inputs will be uniformly spaced values along the x-axis. You can create these in Python using:

```
xTest = np.c_[-2:2:100j]
```

which will create one 100 test inputs between -2 and 2.
Now create an associated feature matrix to add the bias term:

```
PhiTest = np.hstack((xTest, np.ones(xTest.shape)))
```

To obtain the test outputs you can write

```
yTest = np.dot(PhiTest, wStar)
```

Finally you can plot the result with

```
plt.plot(xTest, yTest, 'b-')
```

Part Two: Assignment

Olympic Data For the next part we are going to load in some real data, we will use an example from Rogers and Girolami: the Olympic 100 m finals winners times. To load their data (which is in MATLAB format) we need to import a library

```
import scipy.io
```

which gives us access to the `loadmat` function. Download the olympics data from `http://staffwww.dcs.shef.ac.uk/people/N.Lawrence/olympics.mat`.

```
olympics = scipy.io.loadmat(olympics.mat)
```

This loads the data into a Python dictionary. We can extract the relevant portions using

```
x = olympics['male100'][:, 0]
```

to extract the Olympic years and

```
y = olympics['male100'][:, 1]
```

to extract the 100m winning times. We can plot these as before

```
plt.plot(x, y, 'rx')
```

1. **Linear Model**

   (a) Fit a linear model to the data as above. Plot the fit. Compute the final error.

   (b) Now we will fit a non-linear basis function model. Start by creating a quadratic basis

   ```
   Phi = np.hstack((np.ones(x.shape), x, x**2))
   ```

   and plot the fit, again computing the final error.

   (c) Large order polynomial fits tend to do fairly extraordinary things outside an input range of -1 to 1. Have a think about why this is, in particular, what is the result of 2012^8? Do you think that is a suitable basis?

2. **Radial Basis Functions** We turn to a fit based on exponentiated quadratic (or Gaussian or radial) basis functions:

$$\phi(x) = \exp\left(-\frac{(x-\mu)^2}{2\ell^2}\right)$$

   For each model, you can place the basis functions uniformly across the input space. So we have:

   ```
   muVector = np.r_[1896:2012:(numBasis*1j)]
   ```

   as a one dimensional array containing the centres of the basis, where `numBasis` is the number of basis functions. You can construct the basis matrix using

```
# Allocate the matrix for the basis set
Phi = np.zeros((x.shape[0], numBasis))
# Set the width of the basis functions
width = 2*(x.max() - x.min())/numBasis
# precompute the scale of the exponential
scale = -1/(2*width*width)
for mu in muVector:
    Phi[:, i] = np.exp(scale*(x-mu[i])**2)
```

Note here that we are setting the width of the basis functions according to the distance between the basis functions.

(a) Plot the basis function values for the inputs of years between 1896 and 2012. On the x-axis should be years, on the y-axis the values of the basis functions. Create the plot for 2, 5 and 10 basis functions. Make sure the basis functions are nicely overlapping (i.e. that you've set the width correctly) and that they are spread out across the input range.

Hint: for plotting the function given by your prediction you can use code like this.

```
# Make predictions every year from 1896 until 2012
xPred = np.r_[1896:2012:1]
# Allocate the basis function matrix
PhiPred = np.zeros((xPred.shape[0], numBasis))
for i in range(0, numBasis):
    PhiPred[:, i] = np.exp(width*(xPred-mu[i])**2)
```

(b) Observe what type of functions these basis sets can generate by sampling a weight vector, $\mathbf{w}$, from a Gaussian density and using it to create a function.

```
# Sample a candidate weight vector from a Gaussian.
w = np.random.randn(numBasis, 1)
y = np.dot(PhiPred, w)
plt.plot(xPred, y)
```

Do this a few different times, obtaining a different sample from the Gaussian for $\mathbf{w}$ each time. Do this for models containing 2, 5 and 10 basis functions.

(c) Fit basis function models with between 1 and 27 basis functions (for each number of basis functions have your code create the plot, show the error as the title `plt.title(errorValue)`, pause until a key is pressed and then create the next plot. Store the error values and plot them in a separate plot against the number of basis functions.

Don't forget to fit the noise variance!!

Once you have the estimate of $\mathbf{w}$ (`wStar`), you can make the prediction using a matrix multiplication.

```
yPred = np.dot(PhiPred, wStar)
```

(d) **Data Fit** Which number of basis functions has the lowest error? Do you think this is a good fit?

(e) **Validation Data** Let's assume that we only have the data up until 1976. Hold-out the data from 1980 onwards. Now retrain your models and test the result on this held out data. What is the difference in your results? Which number of basis functions now has the best error?

(f) **Cross Validation** For each number of basis compute and plot:

   i. the leave-one-out cross validation error.

   ii. the 10 fold cross validation error.

   Which model should be chosen according to each of these model selection criterion?

3. **Bayesian Solution** In this question you will perform Bayesian inference on the data. Use the same basis sets from the previous question. Select a Gaussian prior for the mapping vector, $\mathbf{w}$,

$$p(\mathbf{w}) = \mathcal{N}(\mathbf{w}|\mathbf{0}, \alpha\mathbf{I}),$$

where α is the prior variance. Assume a standard deviation of $\sigma = 0.3$ for the noise (i.e. a variance of $\sigma^2 = 0.01$) and a variance of $\alpha = 100$ for the prior variance on $\mathbf{w}$.

(a) **Marginal Likelihood** Now compute and plot the *marginal likelihood*, $\log p(\mathbf{T})$, for each of the models

$$\log p(\mathbf{T}) = -\frac{N}{2}\log 2\pi - \frac{1}{2}\log|\mathbf{K}| - \frac{1}{2}\mathbf{t}^\top \mathbf{K}^{-1}\mathbf{t},$$

where

$$\mathbf{K} = \alpha \boldsymbol{\Phi}\boldsymbol{\Phi}^\top + \mathbf{I}\sigma^2.$$

Find the model with the best fit according to the marginal likelihood. Does it match that chosen by cross-validation or validation?

(b) **Posterior Mean Prediction and Variance** For the selected model compute the mean and covariance of the posterior density for $\mathbf{w}$.

$$\left\langle \mathbf{w}\mathbf{w}^\top \right\rangle = \mathbf{C}_w + \langle \mathbf{w} \rangle \langle \mathbf{w} \rangle^\top$$

$$\mathbf{C}_w = \left[\frac{1}{\sigma^2}\boldsymbol{\Phi}^\top \boldsymbol{\Phi} + \alpha^{-1}\mathbf{I}\right]^{-1}$$

$$\langle \mathbf{w} \rangle = \mathbf{C}_w \sigma^{-2}\boldsymbol{\Phi}^\top \mathbf{t}$$

Plot the mean fit to the data and one standard deviation above and below. For this plot extend the input range of your plot to be between 1850 and 2050.

(c) **Posterior Samples** For the same model take 10 samples from this posterior density to plot 10 fits to the data on top of the data. Plot all the samples in the same plot (continue to use the 200 year input range from 1850 to 2050).

(d) **Quality of Error Bars** Comment on the quality of the error bars. Are the variances for the predictions reasonable across the input range?