DDM 2017 - Problem set 2

1. Bayesian regression

a) Show that Bayesian linear regression with a prior on the parameters $\theta \sim N(0,\tau^2)$ is equivalent to ridge regression.

b) What prior on the parameters would lead to Lasso regression?

2. Visualisation & summaries

On the GitHub repository in `Problemsets/Problem Set 2/` there is a sqlite3 table called `Thirteen-Datasets.db`. This contains 13 tables called `Set01` to `Set13` - each one has two columns x & y.

a) Calculate summary 1D statistics for these datasets - do they differ significantly?

b) Compare the data in 2D plots.

c) Do the dataset differ in 1D plots? Attempt to find some 1D visualisation that shows the dataset to be different.

3. Basic use of emcee

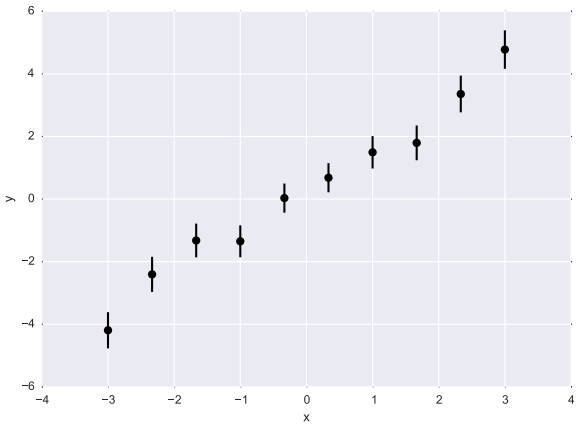
The task here is to carry out the simple use of `emcee` that I covered in the lecture.

Get the dataset from the course GitHub site (the file is called `points_example1.pkl`). It is pickled and can be read in using the routines given on Blackboard or in the lecture notes:

I stored the data as a dictionary so a call:

```
d = pickle_from_file('points_example1_lecture3.pkl')
```

will read the data into a dictionary d. This contains four elements: the x coordinate (key = `x`), the y coordinate(key=`y`), the no noise y value (key=`y_true`) and the uncertainty on a given point (=`sigma`). The data are plotted on the right.



Our task is to fit this with a straight line in a Bayesian framework.

We will do this using emcee.

a) Define the likelihoods using the lecture notes and import emcee.

b) Use Maximum Likelihood to get starting positions for the Bayesian fitting. Check whether you get something close to my best fit values (intercept=0.28233725, slope=1.31299656).

c) Setup the initial positions and run the MCMC code:

```
# Set up the properties of the problem.
ndim, nwalkers = 2, 100
# Setup a number of initial positions.
pos = [p_init + 1e-4*np.random.randn(ndim) for i in range(nwalkers)]
# Create the sampler.
sampler = emcee.EnsembleSampler(nwalkers, ndim, lnprob, args=(x, y_obs,
  sigma))
# Run the process.
```

```
sampler.run_mcmc(pos, 500)
```

Run this and verify that you get a similar output to what I found.

For some extra experience it is worthwhile also running this with a second order polynomial but we will also look at this below.

File content:
A Python dictionary with three keys:

x - independent variable, assumed known precisely.
y - observed quantity = true value plus Gaussian noise.
sigma_y - the standard deviation of the Gaussian noise.

Problem 4 - choosing models

In the lecture I did not discuss model comparisons in detail since we will not dig very deep into this in the lectures, but it is an important topic and this problem is dedicated to cover some of these issues.

Here we are not so much interested in what the parameters of the best fit model are, but rather which of two different models is the better fit to the data.

The dataset you will use here is in the file `data-for-poly-test.pkl` on the GitHub repository. The goal of this problem is to fit polynomials of various powers to the data and to decide which is the best fit to the data.

a) Read in the data and plot these.

b) One common thought when comparing models, is that the difference in the likelihood of the best fits is good way to contrast models. Calculate the maximum likelihood (ML) solution for polynomials with highest order n for n=1 to 9 and plot ln L as a function of n. Does the likelihood of the best-fitting model provide a good way to decide which model is the best?

c) Hopefully your answer to the above is No! (but with some justification). The value of the maximum likelihood alone is therefore not a good way to compare models. In the literature there are a number of alternatives presented - two that are worth knowing about are the Bayesian Information Criterion (BIC) and the Akaike Information Criterion (AIC). These are defined as

$$\mathrm{BIC} = -2\ln L_{\max} + k\ln n \qquad\qquad \mathrm{AIC} = 2k - 2\ln L_{\max}$$

with L_{max} = the maximum of the likelihood, k is the number of free parameters to estimate, and n is the number of data points used in the fit. For both, the model that has the lowest value is to be preferred. But note: the absolute value of BIC or AIC should not be interpreted - a low AIC or BIC does **not** imply that the model is a *good* fit - it should only ever be used for relative rankings.

Calculate BIC and AIC for the models in b) and assess which model is the better. Does the result match your expectations?

In frequentist statistics, an alternative to compare models using BIC or AIC is to compare their reduced χ^2 values against that expected from a χ^2 distribution. Although this problem is one well-suited to χ^2- based model selection, we will not discuss this in great detail here as you probably have seen this in previous courses and the BIC/AIC might be more suitable in generic situations - but see below for some pointers.

(actually in this case - where the higher order model contains the lower order as a special case - a kind of what is known as nested models, you can compare likelihoods directly - this is know as the likelihood ratio test in this case).

d) You will now decide between the models in a Bayesian way. To do this, you can reuse your likelihood function from before, but you also need a prior. For simplicity we will adopt a flat prior over a range +/- 100 in each parameter. In contrast to the previous problem, this prior must be properly normalised so that it has integral unity over the full range of possible parameter values.

Run the fit for a linear and second order model and tabulate your best-fit models and save both the trace and the probability of the model. I expect that you use emcee, in which case the relevant part of the code can look like:

```
sampler.run_mcmc(starting_guesses, nsteps)
trace = sampler.chain[:, nburn:, :].reshape(-1, ndim)
lnprob = sampler.lnprobability[:,nburn:].reshape(-1, 1)
return trace, lnprob
```

where `nburn` is the number of burn-in samples to use (a few hundred to 1000 is probably ok), and `ndim` is the dimensionality of the problem (=polynomial power + 1). `sampler` is the object returned by `emcee.EnsembleSampler`.

Plot the marginal and 2D likelihood distributions using `corner` (or anything else you might prefer), as with the previous problem.

e) The Bayesian way to choose between models is to calculate their *odds ratio*. If we start with the likelihood of model 1 and model 2:

$$p(M_1|D) = \frac{p(D|M_1)p(M_1)}{p(D)} \qquad p(M_2|D) = \frac{p(D|M_2)p(M_2)}{p(D)}$$

we know how to calculate *p(D|M)* since that is the likelihood, and we have to have decided on the prior *p(M)*, which leaves *p(D)*. Luckily we do not need this for calculating the odds ratio, O_{12}, which is the ratio of the posteriors:

$$O_{12} = \frac{p(M_1|D)}{p(M_2|D)} = \frac{p(D|M_1)p(M_1)}{p(D|M_2)p(M_2)}$$

To actually calculate this, you need to integrate the posteriors over the whole parameter space. This can be challenging in general because we might need a large number of samples. If this is infeasible it is also possible to focus on *p(D|M)*, what is frequently called the *evidence* and assume that the ratio of priors is close to unity.

Calculate the odds ratio between a linear and second order polynomial fit to the data. Which do you prefer in this case?

Hints and code suggestions:

If you get stuck, it is a good idea to look at http://jakevdp.github.io/blog/2015/08/07/frequentism-and-bayesianism-5-model-selection/ which covers this topic in a similar way (although it does cover frequentist statistics in some detail where I introduce AIC & BIC. There is a bug on that page though in the prior definition.

For the calculation of the likelihood there are various structures you might adopt. The loop over data points (this assumes that from scipy.stats import norm has been done earlier and that the data are in x, y and sigma_y with the model prediction in y_fit.

The direct loop with indices is:

```
ln_pdf = 0.0
for i in range(len(x)):
   ln_pdf += norm.logpdf(y[i], y_fit[i], sigma_y[i])
```

The possibly more Pythonic loop is:

```
    ln_pdf = 0.0
    for y_v, y_fit_v, sigma_v in zip(y, y_fit, sigma):
        ln_pdf += norm.logpdf(y_v, y_fit_v, sigma_v)
```

and a somewhat different way to structure it would be:

```
    ln_pdf = sum(norm.logpdf(*args)
                 for args in zip(y, y_fit, sigma))
```

These approaches take essentially exactly the same time - what you prefer is up to you.

The integration of the posterior for the Bayes factor is time-consuming. I ended up using the subroutines defined on the blog linked above - see that page for pointers on how to choose the integration limits.

```
# Taken from http://jakevdp.github.io/blog/2015/08/07/frequentism-and-bayesianism-5-model-selection/
from scipy import integrate

def integrate_posterior_2D(log_posterior, xlim, ylim, data):
```

```
    func = lambda theta1, theta0: np.exp(log_posterior([theta0, theta1], data))
    return integrate.dblquad(func, xlim[0], xlim[1],
                             lambda x: ylim[0], lambda x: ylim[1])

def integrate_posterior_3D(log_posterior, xlim, ylim, zlim, data):
    func = lambda theta2, theta1, theta0: np.exp(log_posterior([theta0, theta1, theta2], data))
    return integrate.tplquad(func, xlim[0], xlim[1],
                             lambda x: ylim[0], lambda x: ylim[1],
                             lambda x, y: zlim[0], lambda x, y: zlim[1])
```

Problem 5 - k-nearest neighbour regression

If you have a dataset of $\{x_i, y_i\}$ and a integer k, you can define the nearest neighbour regression as

$$\hat{y} = \frac{1}{k} \sum_{x \in N_k(x)} y_i$$

In other words, for each output x value, find the k closest points in the $\{x_i, y_i\}$ dataset and take the average of the y_i values to get your predicted value.

a) Write your own k-nearest neighbour regression function.

b) Take the data from problem 3 and calculate the k-nearest neighbour fit for k=1, 5 and 9 [if you were unable to solve a), you can use sklearn.neighbors.KNeighborsRegressor]

c) If you have N data points, what is your estimate for the running time of your algorithm as N increases.

d) Create a simulation that tests your assumption in c).