Problem Set #6

Due 26 May 2024

Note, you are submitting all your problems as Jupyter notebooks, so make your code, figures and text readable in a single file.

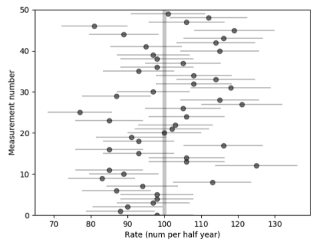
1) Load the sklearn diabetes dataset (`from sklearn.datasets import load_diabetes`). Plot the target values versus all 10 parameters to start. Run a least squares fit on all the parameters versus the target.

2) Load the dataset into a pandas dataframe. Use the linear regression module from sklearn to find the coefficients of a linear regression model (`from sklearn.linear_model import LinearRegression`).

3) Perform a PCA analysis of the data. Include a skee plot for the proportion of variance and eignvalues. Show the first and 2nd PC components with a biplot of the PC components. Comment on your ability to classify individuals at risk.

4) When the LIGO and Virgo detectors have achieved their design sensitivity and gravitational waves the detections from binary black hole should occur regularly. We want to be able to measure the astrophysical rate of these black holes mergers. To do so, we need to be able to count the number that we have detected in a given amount of time at this fixed sensitivity. Let's say that we have finished 50 half-year experiments. While we known the number of mergers can vary a bit, let's say that the true number of mergers that we should detect in a half-year experiment is 100. Because this is a counting experiment, a Poisson distribution is a good approximation to the measurement process.

(a) Generate some data for these N=50 experiments and see if you can make a plot like the one shown in the Figure below. Measurements of the rates R from a Poisson distribution can be achieved with `R = scipy.stats.poisson(R_true).rvs(N)` where R_true is the true rate value of 100. Include error bars on each rate number measured from the 50 experiments. Errors e_i on Poisson counts can be estimated via the square root of each measurement: $e_i = \sqrt{R_i}$.



(b) Calculate the regular mean and σ for your data assuming it is a gaussian distribution.

(c) For a Bayesian approach we want to compute $P(R_{true}|D)$. To do this, apply the Bayes' Theorem:

$$P(R_{true}|D) = \frac{P(D|R_{true})P(R_{true})}{P(D)}$$

First, compute $P(R_{true}|D)$ as a function of R_{true} with the MCMC sampling method using the `emcee` module. Start by defining the following functions in terms of the standard notation for an array of parameters θ, which for this case is just $\theta = [R_{true}]$:

For the log prior $P(R_{true}$ use a flat prior:

```
def log_prior(theta)
  return 1
```

The log likeihood $P(D|R_{true})$ is a function of the experimental results R_i and e_i:

```
def log__likelihood(theta, R, e):
      return -0.5*numpy.sum(numpy.log(2*numpy.pi*e**2) + (R - theta[0])**2/e**2)
```

The log posterior $P(R_{true}|D)$ will be the sum of $\ln P(R_{true})$ and $\ln P(D|R_{true})$

```
def log_posterior(theta, R, e):
      return log_prior(theta) + log_likelihood(theta, R, e)
```

Set-up the problem, including some initial parameters and starting guess for multiple chains of points

```
ndim = 1      # number of parameters in the model
nwalkers = 50 # number of MCMC walkers
nburn = 1000  # "burn-in" period to let chains explore parameter space and stabilize
nsteps = 2000 # number of MCMC steps to take

# Start with some random locations between 0 and 500
starting_guesses = 500 * numpy.random.rand(nwalkers, ndim)
```

Now import the necessary module `emcee` which requires the logarithm of the probability density functions. This is why we defined `log_posterior(theta, R, e)` above. The first argument of `log_posterior` is the position of a single walker. The other arguments come from the `args` parameter of the `emcee.EnsembleSampler`. Do the production run with `emcee.EnsembleSampler.run mcmc` in 2000 steps:

```
import emcee
sampler = emcee.EnsembleSampler(nwalkers, ndim, log_posterior, args=[R, e])
sampler.run_mcmc(starting_guesses, nsteps)
```

The sampler has a property `EnsembleSampler.chain` that is a numpy array with `shape = (nwalkers, nsteps, ndim)`. You can use this to obtain a `sample` array that is reshaped into a flat list and that has the burn-in points discarded:

```
sample = sampler.chain # shape = (nwalkers, nsteps, ndim)
sample = sampler.chain[:, nburn:, :].ravel() # discard burn-in points
```

(d) Plot a histogram of `sample` with `bins=50` and overlay it with a best-fit Gaussian distribution. This can be achieved with `stats.norm(numpy.mean(sample), numpy.std(sample)).pdf(R_fit)` where `R_fit` is the range of points to fit the Gaussian.

(e) Report the result as `numpy.mean(sample)` $\pm$ `numpy.std(sample)`. Is this consistent with $R_{true} = 100$? How does this compare to the mean and σ from part (a)?