Lab 7: Bayesian analysis of a dice toss problem using C++ instead of python

Due date: Thursday March 26, 11:59pm

Short version of the assignment

Take your python file from lab 6 and convert it into lab7 in C++; or reduce the problem to finding the only the probability of throwing a 2 using the C++ programming language.

Slightly longer version of the assignment

There are two possible ways to solve the assignment: (a) produce a full replacement of the python code in C++ (do not use any converter etc, but write the code yourself); (b) create a C++ program that solves a simpler solution: the coin tossing problem, except that we use it for the sides of the dice, for example, calculate the probability of throwing a 2 or a a "not-2". Use the same data D as in lab 6 for the exercise. We will construct a program that is doing Bayesian inference and estimate the posterior probabilities $P(p|D)$ of the probability p of each side of the die. A short refresher on Bayesian inference: Bayes theorem suggests that we can get probability of the parameters of a model (your p) given the data D but assuming some distribution of the parameters and also knowing how to calculate the likelihood that the data fits a particular model (with a specified set of p values), then we can formulate:

$$P(p|D) = \frac{P(p)P(D|p)}{P(D)}, \tag{1}$$

the quantity in the denominator is a scaler so that the posterior distribution integrates to 1.0, thus we could say the $P(D)$ is the integral over all possible values of p: $\int_p P(p)P(D|p)dp$, but for our analysis we can dodge the calculation of this because we use Markov chain Monte Carlo to estimate our quantities. We thus can use

$$P(p|D) \propto P(p)P(D|p). \tag{2}$$

Our task can be broken down into 4 steps:

1. Construct the likelihood function
2. Construct the prior
3. Construct Markov chain Monte Carlo sampler (including a method how to change the p using our prior)
4. Visualize the results, print means etc.

1 Likelihood

We observe results that could be summarized like this: 1: 5, 2: 10, 3:6, 4:9. We have 5 throws that resulted in a 1, 10 throws for 2, 6 throws for 3 and 9 throws for 4, for a total of 30 throws. We will use this data further as a list [5,10,6,9], or more abstract [a,b,c,d]. For (b) we treat our dice like coins for which we could report heads (for example rolling the 2) or tails (all other: 1 3 4) then we would use a binomial distribution, bur for (a) we have 4 sides, thus will need an extension of the binomial and use the *multinomial distribution*, that can be calculated like this

$$P(D|p) = \frac{n!}{a!b!c!d!} p_1^a p_2^b p_3^c p_4^d = \frac{30!}{5!10!6!9!} p_1^5 p_2^{10} p_3^6 p_4^9 \quad (3)$$

[if you solve (b) simply reduce the multinomial to 2 objects instead of using 4]
The problem with this is that the result will be difficult for large numbers of throws, for example 100 or 200 throws will result in problems to calculate the factorials, a remedy to this is to operate all calculations in logs, if we do that then we get

$$\log P(D|p) = (\mathrm{logf}(n) - (\mathrm{logf}(a) + \mathrm{logf}(b) + \mathrm{logf}(c) + \mathrm{logf}(d))) + a \log p_1 + b \log p_2 + c \log p_3 + d \log p_4 \quad (4)$$

[again for (b) use 2 instead of 4]
We could calculate $logf$ as the log of a factorial but that breaks with large numbers, we approximate using this

$$\log(x!) \equiv gammaln(x+1) \quad (5)$$

gammaln is available in the math.h files, here a code snippet that prints a result of the lgamma function:

```
/* lgamma example */
#include <stdio.h>      /* printf */
#include <math.h>       /* lgamma */

double mylogf(double value)
{
   return lgamma(value+1.0);
}

int main ()
{
  double param, result;
  param = 5;
  result = mylogf (param);
 printf ("lgamma(%f) = %f\n", param, result );
  return 0;
}
```

2 Prior

we will use prior that can take the p and calculate probability density function, appropriate for our problem is the Dirichlet distribution that takes p assuming that the p sum to 1 and also uses a set

of parameters, we are laze and use a vector α with all ones, for our problem $\alpha = [1, 1, 1, 1]$, this is equivalent to flat prior where we believe all p come from the same distribution. The Dirichlet PDF needs to be coded because neither numpy nor scipy have it (weirdly enough). There is sample code in this post
`http://stackoverflow.com/questions/10658866/calculating-pdf-of-dirichlet-distribution-in-python`,
take the code and create a function that may look like this:
`double pdf_dirichlet(double x[], double alpha[])` , the translation of the python code to C++ should be straightforward, for the simpler problem you could use a uniform prior, the PDF of the uniform distribution is $\mathrm{PDF}(x, a, b) = 1/(b-a)$ where a, b are the lower and upper bounds, pick 0.0 and 1.0 for these.

3 Markov chain Monte Carlo (MCMC)

Thi section will be the same for both versions (a) or (b) take your python code or the pseudocode below and translate into a C++. Instead of appending to an array I suggest to print directly to the standard out.

- Propose new values p: We can propose new values for p from the prior, for the simple solution draw uniform random number on the interval 0,1 the get a single p value and use $1-p$ for the other value. Check out the function rand(). For the more complex problem you will need to draw random numbers from the Dirichlet distribution, this is somewhat convoluted, you could use the stick breaking algorithm that uses a stick of size 1, then breaks of a piece using a random number for $\mathrm{Beta}(1, a_1)$, the remaining stick is again broken using $\mathrm{Beta}(1, a_2)$, ... until $\mathrm{Beta}(1, a_n)$, this will lead to a set of p values for the sides of the dice. How to generate a $\mathrm{Beta}(1, a_k)$ random variable: if X and Y are indpendent random draws from a Gamma distribution with parameters a_k and 1 then $X/(X+Y)$ is a $\mathrm{Beta}(1, a_k)$ random draw. In modern C++11 you can get a Gamma deviated number easily, try out this code snippet:

  ```
  #include <random>
  #include <iostream>
  int main()
  {
      typedef std::mt19937 G;
      typedef std::gamma_distribution<> D;
      G g;  // seed if you want with integral argument
      double k = .5;      // http://en.wikipedia.org/wiki/Gamma_distribution
      double theta = 1.0;
      D d(k, theta);
      std::cout << d(g) << '\n';
  ```

  Look up the `stick breaking` algorithm in
  http://mayagupta.org/publications/FrigyikKapilaGuptaIntroToDirichlet.pdf There they give a way to simulate a sample from the Dirichlet using Beta random numbers. If you find a better solution let me know.

- Start with an arbitrary value for example `p=np.random.dirichlet(alpha)`, evaluate the posterior with these p, `post=pdf_dirichlet(alpha) * like(data,p)`, then run for a large number of cycles through this recipe:

1. propose new p
2. evaluate the new posterior `new`
3. compare `new` with `old` (see above the `post` that probably should better called `old`); if the `new` is better than the `old` we will accept the new p and record it (for example append it to results), if `new` is smaller than `old` we accept with some probability r, this can be done easily using a condition $r < new/old$, but remember, we used logs to calculate all quantities, so our condition turns into this:

```
r = numpy.random.uniform(0,1)
if np.log(r) < new - old:
    append new p to results (or print to standard out)
    oldp = p
    old = new
else:
    append old p to results (or print to standard out)
```

The results contain now a chain of 'accepted' p values, a histogram of these will represent the posterior.

4 Visualize results

Write a short python program to read the printed out values and plot the result, this can be done by reusing some of your original lab 6 code. Use a histogram to show the bars for each posterior for each side of the die. Check out the `hist` examples. Discuss your results, if you are adventours, try to calculate the credibility intervals.

Peter Beerli March 2015