Introduction to PyMC2

Bálint Szőke

March 11, 2016

New York University

What is it for?

Example setup:

1. You have a sample $\{\tilde{y}_t\}_{t=0}^T$
2. Want to characterize it by the probabilistic model

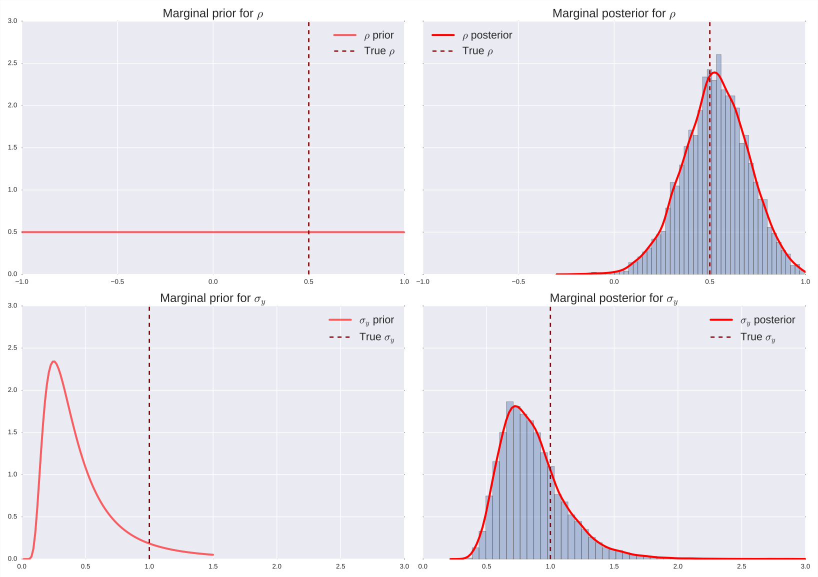
$$y_{t+1} = \rho y_t + \sigma_y \varepsilon_{t+1}, \quad \varepsilon_{t+1} \overset{iid}{\sim} \mathcal{N}(0,1), \quad \forall t \geq 0$$

   with the initial value $y_0 \sim \mathcal{N}\left(0, \frac{\sigma_y^2}{1-\rho^2}\right)$. $\Rightarrow$ likelihood function
3. You have prior beliefs about the parameters $\theta \equiv (\rho, \sigma_y)$

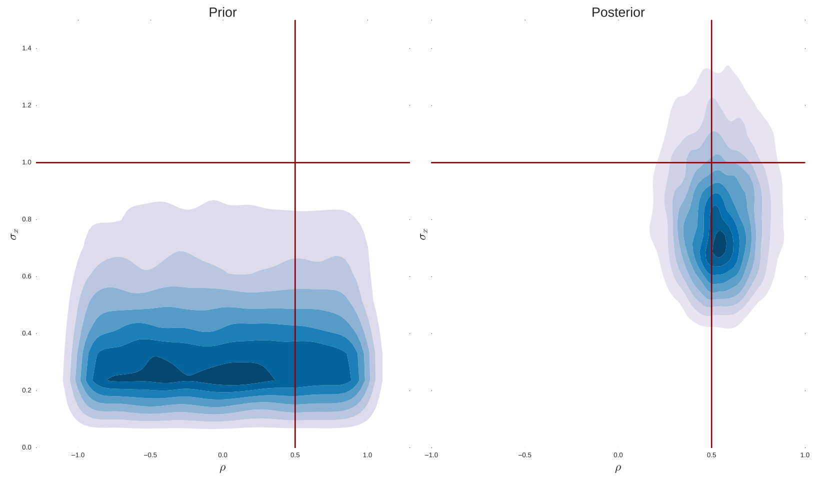
$$\rho \sim \mathcal{U}(-1,1) \qquad \sigma_x \sim \mathcal{IG}(\alpha, \beta)$$

Aim: learn about the posterior distribution $p(\theta | \{\tilde{y}_t\}_{t=0}^T)$

Going from prior to posterior



Going from prior to posterior II



Defining a model in PyMC

`pymc.Model` is a collection (list, set, tuple, dictionary, function, etc.) of random variables linked together according to some rules

- Linked in a hierarchical structure:

  parent: variables that influence another variable

  e.g. ρ and σ_y are parents of y_0; α, β are parents of σ_y

  child: variables that are affected by other variables

  e.g. y_{t+1} is a child of y_t, ρ and σ_y

$$f(y_{t+1}|y_t;\theta) = \mathcal{N}(\rho y_t, \sigma_y^2) = \mathcal{N}(\mu_t, \sigma_y^2)$$

What is the point?

child's current value automatically changes whenever its parents' change

- Two main classes of random variables in `pymc`

  `Stochastic` : value is not completely determined by its parents

  `Deterministic` : value is entirely determined by its parents

Stochastic variables

Two main examples:

1. parameters with a given distribution, e.g. ρ and σ_y
2. observable variables: realization of a random variable

Initialization (parameter):

```
rho = pymc.Uniform('rho', lower = -1, upper = 1)
sigma_x = pymc.InverseGamma('sigma_x', alpha = 3, beta = 1)
```

- built-in (capitalized!) or your own distribution
- Optional flags:
  - `value` : for a default initial value; if none, draw from prior
  - `size` : multivariate array of independent stochastic variables
  - `observed` : boolean to fix the value (forever)

Stochastic **variables**

Treated by the back end as random number generators

```
rho.value    # current internal value (given parents)
rho.random() # redraw the value
rho.logp     # logprob at value (for vectors the sum)
rho.parents  # dictionary of parents' names and values
rho.children # set (!) of children
```

Warning: Don't update stochastic variables' values in-place

The only way a stochastic variable's value should be updated is this:

```
rho.value = new_value
```

NEVER use things like:

```
rho.value += 3
rho.value[2,1] = 5
```

`Deterministic` **variables**

"exact functions" of stochastic variables

> defined as functions, but always *specified with default values*, so for all purposes we can treat them as variables

Main examples

> how the parameters and the observable variables are related
>
> - var(y_0) is a function of ρ and σ_y
> - $\mu_t = \mathbb{E}[y_{t+1}|y_t]$ is an exact function of ρ and y_t

Three ways to create a `Deterministic` variable

- `@pymc.deterministic` decorator
- elementary operators: $+$, $-$, $*$, $/$
- `pymc.Lambda`

Deterministic **variables**

1. Decorator form:

```
@pm.deterministic
def y0_stdev(rho = rho, sigma = sigma_x):
    return sigma / np.sqrt(1 - rho**2)
```

2. The same with `pymc.Lambda`:

```
y0_stdev = pm.Lambda('y0_stdev',\
 lambda r = rho, s = sigma_x: s / np.sqrt(1 - r**2) )
```

3. Elementary operator:

```
mu_y = rho * sample_path[:-1]
```

What to do with the sample?

Define the data as a `Stochastic` variable with *fixed values* (=data) by setting the `observed` equal to `True`

```
y0 = pm.Normal('y0', mu = 0.0, tau = 1 / y0_stdev,\
               observed = True, value = sample_path[0])
Y = pm.Normal('Y', mu = mu_y, tau = 1 / sigma_y, \
               observed = True, value = sample_path[1:])
```

The values are fixed, but the parents' values are always updated

```
Y.value                    # numpy array
Y.parents['tau'].value     # parents is a dictionary
```

Create a `pymc.Model` instance

Just a collection of all the created `Stochastics` and `Determinsitics`

```
ar1_model = pm.Model([rho, sigma_x, y0, Y, y0_stdev, mu_y])
```

You can easily look into this collection

```
ar1_model.stochastics        # (unordered) set of names
ar1_model.deterministics
```

Fitting the model to the data: Markov Chain Monte Carlo

MCMC algorithms

Aim: sampling from the posterior distribution

$\approx$ exploring the posterior which is sitting on the parameter space

...could pick random points on the parameter space (Monte Carlo)

...intelligent way of exploring the surface (Markov Chain)

MCMC is an iterative search mechanism, in each step j with $\theta_j = \theta$, it

- proposes a nearby point θ'
- asks 'how likely that θ' is close to the maximizer?'
  - Accept if the likelihood exceeds a particular threshold $\Rightarrow \theta_{j+1} = \theta'$
  - Reject otherwise $\Rightarrow \theta_{j+1} = \theta$

Key feature: proposals come from simulating a Markov Chain for which the posterior is the *unique*, *stationary distribution*

PyMC2's MCMC algorithm

By default Metropolis-within-Gibbs (in my opinion)

1. **Blocking and conditioning**:
    - split the N-vector θ into $K \leq N$ blocks
    - at scan t, cycle through the K blocks: $\theta^{(t)} = [\theta_1^{(t)}, \theta_2^{(t)}, \theta_3^{(t)}, \ldots, \theta_K^{(t)}]$
    - Sample from the conditionals:

$$
\begin{aligned}
\theta_1^{(t+1)} &\sim f(\theta_1 \mid \theta_2^{(t)}, \theta_3^{(t)}, \ldots, \theta_K^{(t)}; \text{data}) \\
\theta_2^{(t+1)} &\sim f(\theta_2 \mid \theta_1^{(t+1)}, \theta_3^{(t)}, \ldots, \theta_K^{(t)}; \text{data}) \\
&\ldots \\
\theta_K^{(t+1)} &\sim f(\theta_3 \mid \theta_1^{(t+1)}, \theta_2^{(t+1)}, \ldots, \theta_{K-1}^{(t+1)}; \text{data})
\end{aligned}
$$

PyMC2's MCMC algorithm

By default Metropolis-within-Gibbs (in my opinion)

2. **Block-wise sampling with** `pymc.StepMethod`
   - If $f(\theta_i|\theta_{-i})$ has a (semi-)analytic form sample from that distribution
   - If $f(\theta_i|\theta_{-i})$ is not available, use Metropolis-Hastings
     1. Start at θ
     2. Propose a new point according to the proposal density $J(\theta'|\theta)$
     3. Accept the proposed point with probability

$$\alpha = \min\left(1, \frac{p(\theta' \mid \text{data})\ J(\theta \mid \theta')}{p(\theta \mid \text{data})\ J(\theta' \mid \theta)}\right)$$

        If accept: Move to the proposed point θ' and return to Step 1.

        If reject: Don't move, keep the point θ and return to Step 1.

     4. After a large number of iterations (once the Markov Chain convereged), return all accepted θ as a sample from the posterior

PyMC2's MCMC algorithm

Construct an MCMC instance ...

```
M = pymc.MCMC(ar1_model)    # ready to be sampled from
```

...to create and coordinate a collection of step methods

Main built-in `pymc.StepMethods` (assigned automatically) ...

- Metropolis
- AdaptiveMetropolis
- Slicer

...or you can assign step methods manually

```
M.use_step_method(pymc.Metropolis, rho, proposal_sd = 2.0)
```

Sampling with PyMC

After all the 'hard work', sampling from the posterior is one line

```
# drop the first 20,000 and keep only every 10th draw
M.sample(iter = 100000, burn = 20000, thin = 10)
```

The sample can be reached by the `trace` method

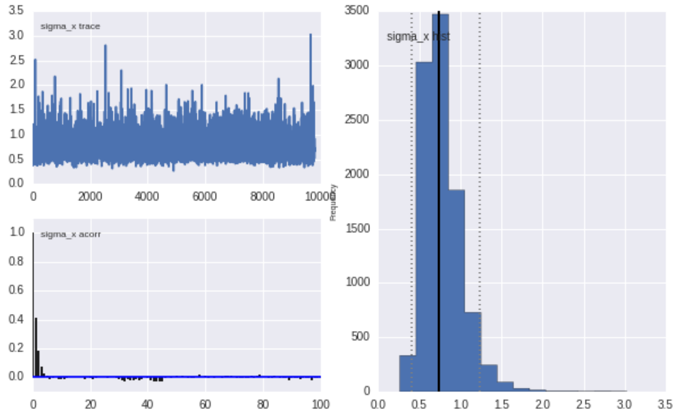
```
M.trace('rho')[:]             # numpy array for the whole sample
M.trace('sigma_x')[:100]      # first 100 elements
```

For a non-graphical summary about the marginal posterior

```
M.stats('rho')          # dictionary of a variety of stats
M.summary()
```

PyMC magic

```
from pymc.Matplot import plot as fancy_plot
fancy_plot(M.trace('sigma_x'))   # immediately save it
```



Some ‘protips’

(Use them at your own peril!)

How to manipulate the `pymc.StepMethod`?

`pymc.StepMethod` is meant to be subclassed!

```
class My_method(pymc.StepMethod):
    def __init__(self, stochastic, Y, rho, sigma_x):
        pymc.StepMethod.__init__(self, stochastic)
```

...last line creates `self.stochastics`, which is an unordered set (!)

Poor man's way of making `stochastic` an ordered list

```
    x_dict = {x:y for y, x in enumerate(stochastic)}
    pymc.StepMethod.__init__(self, stochastic)
    my_list = [None]*len(self.stochastics)
    for element in self.stochastics:
        my_list[x_dict[element]] = element
    self.my_list = my_list
```

...and use `my_list` instead of `stochastics`

How to manipulate the `pymc.StepMethod`?

The key function that **must** be rewritten is `step()`

```
class My_method(pymc.StepMethod):
    def step(self):
        # some updating scheme
        self.rho.value = rho_new_value
```

...it should update the `Stochastic` variables in the block

How to manipulate the pymc.StepMethod?

Update array-valued random variables:

1. Store scalar-valued `Stochastic` variables in a list

```
Theta = np.empty(T + 1, dtype = object)
for i in range(1, T + 1):
    Theta[i] = pymc.Normal('theta_{:d}'.format(i),\
                            mu = Theta[i-1], tau = 1)
Theta = pymc.Container(Theta)
```

2. In `My_method.__init__` construct the ordered `my_list` as above
3. In `My_method.step` update element-wise with a loop on `my_list`

```
for ind, stoch in enumerate(self.my_list):
    stoch.value = theta[ind]
```

How to manipulate the `pymc.StepMethod`?

We can also 'split' step into pieces by the two other methods (optional)

```
class My_method(pymc.StepMethod):
    def propose(self):
        self.rho.value = rho_new_value

    def reject(self):
        self.rho.revert()  # go back to the previous value

    def step(self):          # rejection sampling
        self.propose()
        if something:
            self.reject()
```