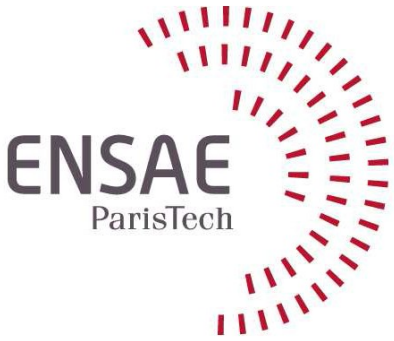

Hidden Markov Models and Sequential Monte-Carlo methods

Project : PMCMC on a parameter driven Poisson times series model

Nicolas Chopin

2015/2016

Nathan Elbaz, Morgan Schmitz & Robin Vogel

ENSAE M2 Data Science

I. Introduction

We worked on a model that was described by two statisticians in the paper [1]. We will use a particle filter and PMCMC (Particle Markov Chain Monte-Carlo) methods to simulate the a posteriori distribution of the model's parameters.

II. Model description

The original model comes from astrophysics, where (binned) photon counts from a source are modeled by a Poisson distribution:

$$Y_t|(\xi_t, \beta) \sim \mathcal{P}(d_t \mathrm{e}^{Z_t\beta+\xi_t}) \tag{1}$$

$$\begin{cases} \xi_t|(\xi_{1:t-1}, \rho, \delta) \sim \mathcal{N}(\rho\xi_{t-1}, \delta^2) & \text{if } t > 1 \\ \xi_1 \sim \mathcal{N}(0, \delta^2/(1-\rho^2)) & \text{if } t = 1 \end{cases} \tag{2}$$

Where d_t is a known constant (typically the width of the bin corresponding to t), and $Z_t = (1, t/T)$. The parameters we want to estimate are $\beta = (\beta_0, \beta_1)^\top$, ρ and δ. Of particular interest to astrophysicists is β_1, since a null value implies no significant trend over time (hence the definition of Z_t), which has implications on the type of object we are observing.
Note that in the paper, Z_t is actually called X_t. We renamed it in order to use X_t for the hidden Markov Chain of our state-space model instead, so as to keep the course's notations. That is, $\forall t \in \{1, \dots, T\}$, we define X_t as:

$$X_t := \xi_t + Z_t\beta \tag{3}$$

Which corresponds to η_t in the authors' notation. Let us now rewrite our model:

$$Y_t \sim \mathcal{P}(d_t \mathrm{e}^{X_t}) \tag{4}$$

$$\begin{cases} X_t|X_{1:t-1} \sim \mathcal{N}(\rho X_{t-1} + (Z_t - \rho Z_{t-1})\beta, \delta^2) & \text{if } t > 1 \\ X_1 \sim \mathcal{N}(Z_1\beta, \delta^2/(1-\rho^2) & \text{if } t = 1 \end{cases} \tag{5}$$

This brings us back to a classical model with observed states Y_t and a hidden Markov Chain X_t. Our goal in this project is to be able to simulate according to the posterior distribution of our parameters of interest, $\theta := (\beta_0, \beta_1, \rho, \delta)$. We will use a PMCMC algorithm to do so.

III. Particle Filter

The first step was to be able to estimate the likelihood of a set of observations $y_{1:T}$ given a set of parameters θ. We used the simplest Particle Filter, a Bootstrap Filter.

I. Initialization

We first generate N initial particles from the X_1 distribution (see (5)) and compute their weights:

$$w_0^n = f(y_0|X_0^n)$$

Note that in our case, since Y_t follows a Poisson distribution with e_t^X (times a constant) as parameter, we can use the probability mass function as f. We then store the mean across all particles of those weights, L_0^N, and the normalized weights W_0^n (so that they sum to 1). In practice, we manipulate the log of the weights instead for computational reasons.

II. Particle Filtering

Then at every instant $t \leq T$, we perform a resampling step (see below), and update our particles using (5) (where X_{t-1} is the selected ancestor). We then recalculate the weights, their normalized counterparts, and update our estimator of the likelihood:

$$L_t^N := L_{t-1}^N \frac{\sum_{n=1}^N w_t^n}{N}$$

At the end of the loop over t, we have in L_T^N an unbiased estimate of $L(y_{1:T}|\theta)$.
The code is given in the appendix.

III. Resampling

Our Particle Filter function, `PF`, can take as optional arguments a callable `resampler`. This function itself is expected to take as arguments a list of normalized weights, and returns a boolean.
By default, the `"systematic"` resampler is used; it simply returns True no matter what, thus leading to the classical Bootstrap Filter with resampling done at every iteration over t.
We also tried to compute the Effective Sample Size (ESS), and to trigger the resampling step only when that value was beyond a certain threshold. The code can be found in the appendix.
In both cases, when the resampling was meant to happen, we pick the ancestor from a multinomial distribution with the normalized weights as their probabilities of selection. (When no resampling step is done, every particle is propagated.)
In the results that follow (based on simulated data), we decided to use $T = 100$ instead of $T = 1000$, as was the case in the paper's real data, for speed's sake.

IV. PMCMC

The PMCMC algorithm is a technique that combines the Metropolis-Hastings algorithm and the particle filter. The Particle filter is used to compute an estimate of the likelihood of the parameters, which in turn is used in a sort of Metropolis-Hastings, where the real likelihood is replaced by the estimates obtained with the filter. Our implementation of it is given in Listing 3.
Because of obvious constraints on ρ, and to ensure that δ does not lead to identifiability problems, in the results shown below, we chose to draw new proposed values of the PMCMC algorithm from:

- A bivariate random normal variable centered in β with covariance matrix I_2 for the proposal β^*.

- A random normal variable truncated on $[-1, 1]$ centered in ρ for the proposal ρ^*.
- A random normal variable left-truncated at 0 centered in δ^2 for the proposal δ_*^2.

Our PMCMC function, `pmcmc`, has a `prop` optional argument, expected to be an object with `rvs` and `logpdf` methods (similar to those of the distribution classes from `scipy.stats`), allowing for easy modification of the proposal kernel. In our case, it is defined in `h_norms` (see the appendix).

V. Results

I. Particle Filter

In order to test our algorithms, we simulated data using equations (4-5), with parameters:

$$\begin{aligned} d_t &= 1 \\ \beta &= (1, 0)^\top \\ \rho &= 0.2 \\ \delta &= 2 \end{aligned}$$

The data thus created is represented in Figure 1.
We tried applying our PF to this dataset with different thetas, number of particles, and both of our sampling schemes. The number of runs performed, the average of the resulting likelihood estimators, its Monte Carlo error and the average computing time are given in Table 1.

II. PMCMC

In this section, we present the results of our PMCMC algorithm applied to the data we presented above. The initialization was made at $\beta_0 = \beta_1 = \rho = \delta = 0$. 100 particles were used in each call to the Particle Filter.
The histograms we obtain from 4500 iterations (after removal of 500 burn-in points) are given in Figure 2.

References

[1] YU, *Y. and* MENG, *X.L. (2015)*
To Center or Not to Center: That Is Not the Question
An Ancillarity–Sufficiency Interweaving Strategy (ASIS) for Boosting MCMC Efficiency

θ	`Resampler`	Number of runs	$\hat{L}_T^N$	Monte Carlo error	Average computing time
θ_d	systematic	500	-264.366298565	4.94814065282	0.24996315
	ESS	500	-263.045679032	3.92571292512	0.18878245
	systematic	50	-260.269817255	0.714430193034	7.33796948433
θ_1	systematic	500	-301.836653808	2.10812340986	0.21910377
θ_0	systematic	500	-760.593698079	1.13686837722e-13	0.16407303

Table 1: *Comparison of different PF runs. θ_d: parameters used to generate data; $\theta_1 = (1.5, 1, 0.11, 1.5)$; $\theta_0 = (0, 0, 0, 0)$*

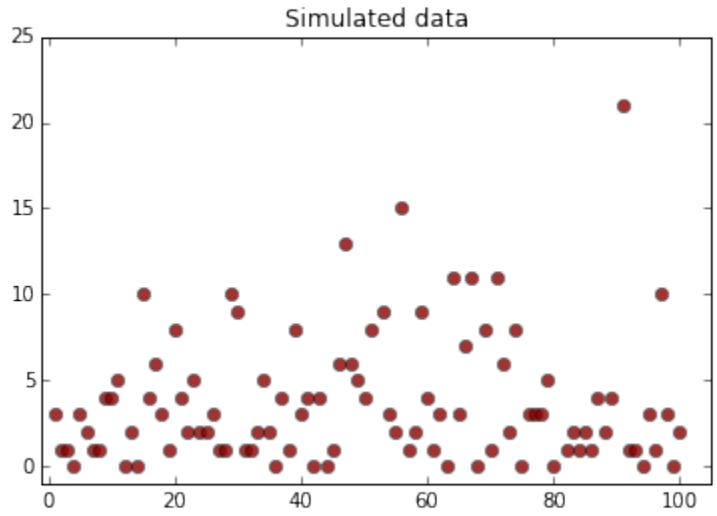

Figure 1: *Simulated data*

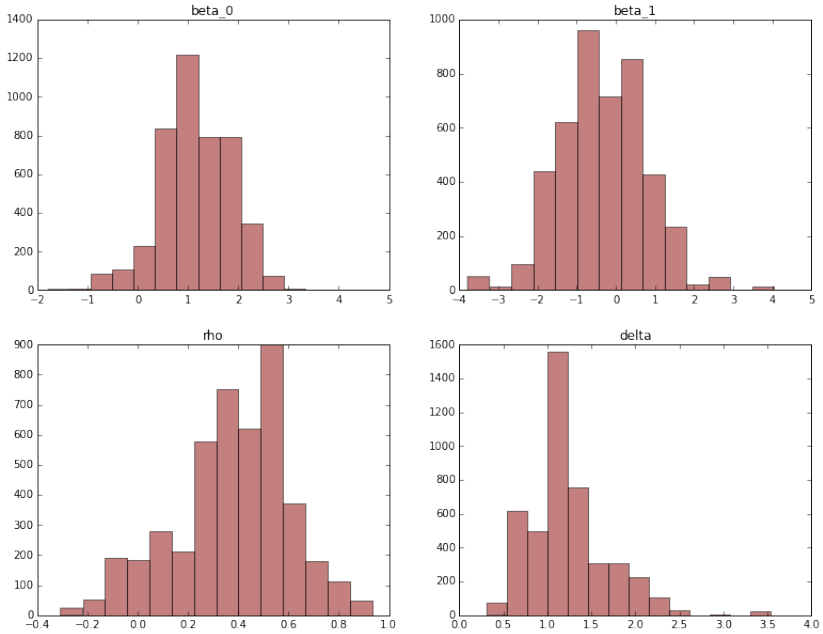

Figure 2: *Histograms for the posterior of our parameters of interest*

Listing 1: *Particle Filter*

```
import numpy as np
from scipy.stats import norm, poisson, truncnorm
from numpy.random import multinomial
from math import sqrt, exp, log, pi
import matplotlib.pyplot as plt
import time
def PF(theta, N, yt, resampler=systematic, gamma=0.4, Verbose=False):
    # define parameters from theta
    beta = theta[0:2]
    rho = theta[2]
    delta = theta[3]
    T = len(yt)
    dt = 1.

    # t=0
    X = np.array(norm.rvs(size=N, loc = (beta[0] + float(beta[1])/T),
                          scale = (sqrt(delta) / sqrt(1-rho**2))))
    logw = poisson.logpmf(yt[0], mu = dt * np.exp(X))
    m = np.max(logw)
    W = np.exp(logw-m)
    logL = log(1./N * sum(W)) + m
    W = W/sum(W)
    if Verbose==True:
        print "Initialization", "\tLikelihood estimator: ", logL

    # t>0
    for t in range(2, T+1):
        if t%200 == 0 & Verbose==True:
            print "Iteration: ", t, "\tLikelihood estimator: ", logL
        X_new = np.empty(N)
        if resampler(W, gamma) == True:
            for n in range(N):
                An = np.where(multinomial(1, W))[0][0]
                mu = rho*X[An] + (1-rho)*beta[0] + (t*(1-rho)
                    + rho)/T*beta[1]
                X_new[n] = norm.rvs(loc=mu, scale=sqrt(delta))
        else:
            for n in range(N):
                An = n
                mu = rho*X[An] + (1-rho)*beta[0] + (t*(1-rho)
                    + rho)/T*beta[1]
                X_new[n] = norm.rvs(loc=mu, scale=sqrt(delta))
        X = X_new
        logw = poisson.logpmf(yt[t-1], mu = dt * np.exp(X))
        m = max(logw)
        W = np.exp(logw-m)
        logL = logL + log(sum(W)) + m - log(N)
```

```
        W = W/sum(W)
    return logL
```

Listing 2: *Functions to determine whether or not to resample at each iteration of the PF*

```
def systematic(W, gamma):
    return True
def ESS(W, gamma):
    ESS = 1. / sum([w**2 for w in W])
    if ESS < gamma*len(W):
        return True
    else:
        return False
```

Listing 3: *Implementation of PMCMC*

```
def pmcmc(n_iter, data, theta0, prop=h_norms(), pf=PF, n_particles=100):
    L = lambda theta: pf(theta, n_particles, data)
    h = lambda theta, mean: np.sum(prop.logpdf(theta, mean))
    #p = lambda theta: np.prod(prior.pdf(theta))
    theta_i = theta0
    results = np.empty((n_iter, 4))

    for i in range(n_iter):
        if i%50==0:
            print "Iteration ", i, theta_i
        theta_star = prop.rvs(theta_i)
        logr = L(theta_star) + h(theta_i, theta_star) - (
            L(theta_i) + h(theta_star, theta_i))
        if logr > 0:
            results[i,:] = theta_star
            theta_i = theta_star
        else:
            u = np.random.uniform(0,1)
            if u < exp(logr):
                results[i,:] = theta_star
                theta_i = theta_star
            else:
                results[i,:] = theta_i
    return(results)
```

Listing 4: *Definition of our proposal kernel*

```
class h_norms():
    def __init__(self):
        pass

    def logpdf(self, theta, param, scale=1.):
        logpdfs = np.empty(4)
```

```
        logpdfs[0:2] = norm.logpdf(theta[0:2], loc=param[0:2],
            scale=scale)

        leftcut = (-1-param[2])/scale
        rightcut = (1-param[2])/scale
        logpdfs[2] = truncnorm.logpdf(theta[2], leftcut, rightcut,
            loc=param[2], scale=scale)

        myclip_a, myclip_b = 0, float("inf")
        my_mean = param[3]
        my_std = scale
        a, b = (myclip_a -
            my_mean) / my_std, (myclip_b - my_mean) / my_std
        logpdfs[3] = truncnorm.logpdf(theta[3], a, b, loc=my_mean,
            scale=my_std)

        return logpdfs

    def rvs(self, param, scale=1.):
        new_theta = np.empty(4)
        new_theta[0:2] = norm.rvs(size=2, loc=param[0:2], scale=scale)

        leftcut = (-1-param[2])/scale
        rightcut = (1-param[2])/scale

        new_theta[2] = truncnorm.rvs(leftcut, rightcut, loc=param[2],
            scale=scale)

        myclip_a,myclip_b = 0,float("inf")
        my_mean = param[3]
        my_std = scale
        a, b = (myclip_a - my_mean) / my_std, (myclip_b -
            my_mean) / my_std
        new_theta[3] = truncnorm.rvs(a, b,loc=my_mean, scale=my_std)
        return new_theta
```