BIP - Bayesian Inference with Python Documentation

Release 0.5.13

Flávio Codeço Coelho

2014-05-27

This documentation corresponds to version 0.5.13.

CONTENTS

1.1 Overview

The Bip Package is a collection of useful classes for basic Bayesian inference. Currently, its main goal is to be a tool for learning and exploration of Bayesian probabilistic calculations.

Currently it also includes subpackages for stochastic simulation tools which are not strictly related to Bayesian inference, but are currently being developed within BIP. One such package is the BIP.SDE which contains a parallelized solver for stochastic differential equations, an implementation of the Gillespie direct algorithm.

The Subpackage Bayes also offers a tool for parameter estimation of Deterministic and Stochastic Dynamical Models. This tool will be fully described briefly in a scientific paper currently submitted for publication.

1.2 Parameter Estimation in Dynamic Models

A growing theme in mathematical modeling is uncertainty analysis. The Melding Module provides a Bayesian framework to analyze uncertainty in mathematical models. It includes tools that allow modellers to integrate Prior information about the model's parameters and variables into the model, in order to explore the full uncertainty associated with a model.

This framework is inspired on the original Bayesian Melding paper by Poole and Raftery [1], but extended to handle dynamical systems and different posterior sampling mechanisms, i.e., the user has the choice to use Sampling Importance resampling, Approximate Bayesian computations or MCMC. A deeper description of the methodology implemented in this package is available as published research paper [2]. This paper also contains a more extensive example of parameter estimation. If you intend to use this package for a scientific publication, you should cite this paper [1].

Once a model is thus parameterized, we can simulate the model, with full uncertainty representation and also fit the model to available data to reduce that uncertainty. Markov chain Monte Carlo algorithms are at the core of the framework, which requires a large number of simulations of the models in order to explore parameter space.

1.2.1 Single Session Retrospective estimation

Frequently, we have a complete time series corresponding to one or more state variables of our dynamic model. In such cases it may be interesting to use this information, to estimate the parameter values which maximize the fit of our model to the data. Below are examples of such inference situations.

---

[1] Poole, D., & Raftery, A. E. (2000). Inference for Deterministic Simulation Models: The Bayesian Melding Approach. Journal of the American Statistical Association, 95(452), 1244-1255. doi:10.2307/2669764

[2] Coelho FC, Codeço CT, Gomes MGM (2011) A Bayesian Framework for Parameter Estimation in Dynamical Models. PLoS ONE 6(5): e19616. doi:10.1371/journal.pone.0019616

Example Usage

This first example includes a simple ODE (an SIR epidemic model) model which is fitted against simulated data to which noise is added:

```python
#-*- coding: utf-8 -*-
"""
Parameter estimation and series forcasting based on simulated data with moving window.
Deterministic model
"""
#
# Copyright 2009- by Flávio Codeço Coelho
# License gpl v3
#
from BIP.Bayes.Melding import FitModel
from scipy.integrate import odeint
import scipy.stats as st
import numpy as np


beta = 1 #Transmission coefficient
tau = .2 #infectious period. FIXED
tf = 36
y0 = [.999,0.001,0.0]
def model(theta):
    beta = theta[0]
    def sir(y,t):
        '''ODE model'''
        S,I,R = y
        return   [-beta*I*S, #dS/dt
                  beta*I*S - tau*I, #dI/dt
                  tau*I] #dR/dt
    y = odeint(sir,inits,np.arange(0,tf,1))
    return y


F = FitModel(500, model,y0,tf,['beta'],['S','I','R'],
            wl=36,nw=1,verbose=0,burnin=100)
F.set_priors(tdists=[st.norm],tpars=[(1.1,.2)],tlims=[(0.5,1.5)],
    pdists=[st.uniform]*3,ppars=[(0,.1),(0,.1),(.8,.2)],plims=[(0,1)]*3)
d = model([1.0]) #simulate some data
noise = st.norm(0,0.01).rvs(36)
dt = {'I':d[:,1]+noise} # add noise
F.run(dt,'MCMC',likvar=1e-5,pool=True,monitor=[])
#==Uncomment the line below to see plots of the results
F.plot_results()
```

The code above starts by defining the models parameters and initial conditions, and a function which takes in the parameters runs the model and returns the output.

After that, we Instantiate our fitting Object:

```python
F = FitModel(300,model,y0,tf,['beta'],['S','I','R'],
            wl=36,nw=1,verbose=False,burnin=100)
```

Here we have to pass a few arguments: the first (`K=300`) is the number of samples we will take from the joint prior distribution of the parameters to run the inference. The second one (`model`) is the callable(function) which corresponds to the model you want to fit to data. Then you have the initial condition vector(`inits=y0`), the list of parameter names (`thetanames = ['beta']`), the list of variable names (`phinames=['S','I','R']`), inference window length (`wl=36`), number of juxtaposed windows (`nw=1`), verbosity flag (`verbose=False`) and finally the number of burnin samples (`burnin=1000`), which is only needed for if the inference method chosen is `MCMC`.

One should always have `verbose=True` on a first fitting run of a model or if the simulations seems to be taking longer than expected. When `verbose` is true, printed and graphical is generated regarding the behavior of fitting, which can be useful to fine tune its parameters.

The next line of code also carries a lot of relevant information about the inference: the specification of the prior distributions. By now you must have noticed that not all parameters included in the model need to be included in the analysis. any number of them except for one can be set constant, which is what happens with the parameter `tau` in this example:

```
F.set_priors(tdists=[st.norm],tpars=[(1.1,.2)],tlims=[(0.5,1.5)],
    pdists=[st.uniform]*3,ppars=[(0,.1),(0,.1),(.8,.2)],plims=[(0,1)]*3)
```

here we set the prior distributions for the theta (the model's parameters) and phi (the model's variables). `tdists`, `tpars` and `tlims` are theta's distributions, parameters, and ranges. For example here we use a Normal distribution (`st.norm`) for `beta`, with mean and standard deviation equal to 1.1 and .2, respectively. we also set the range of `beta` to be from 0.5 to 1.5. We do the same for phi.

The remaining lines just generate some simulated data to fit the model with, run the inference and plot the results which should include plots like this:
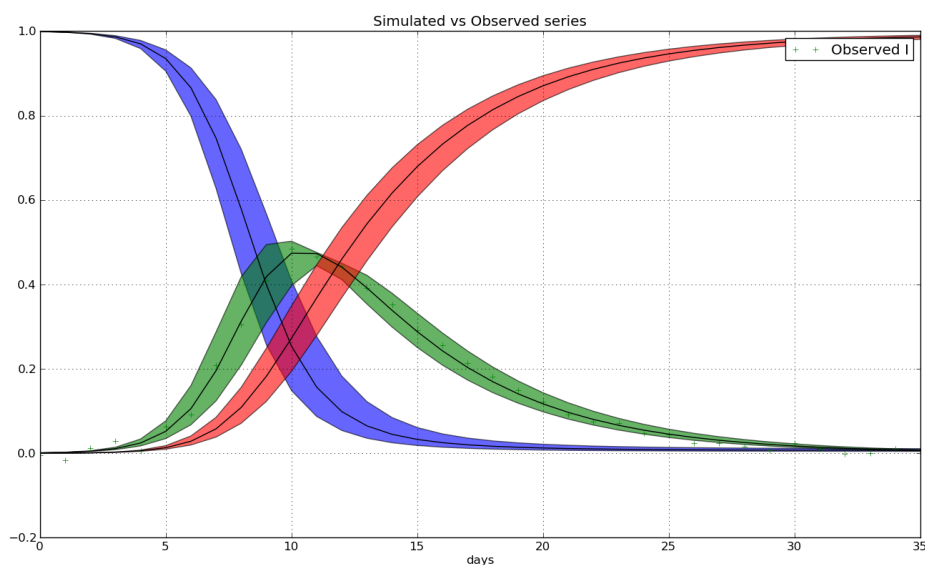

Figure 1.1: Series posterior distributions. Colored areas represent 95% credible intervals.

One important argument in the run call, is the likvar, Which is the initial value for the likelihood variance. Try to increase its value if the acceptance ratio of the markov chain is too llow. Ideal levels for the acceptance ratio should be between 0.3 and 0.5.

The code for the above example can be found in the examples directory of the BIP distribution as `deterministic.py`

Stochastic Model Example

This example fits a stochastic model to simulated data. It uses the *SDE* package of BIP:

```python
#-*- coding:utf-8 -*-
"""
Parameter estimation and series forcasting based on simulated data with moving window.
Stochastic model
"""
#
# Copyright 2009- by Flávio Codeço Coelho
# License gpl v3
#
from BIP.SDE.gillespie import Model
```
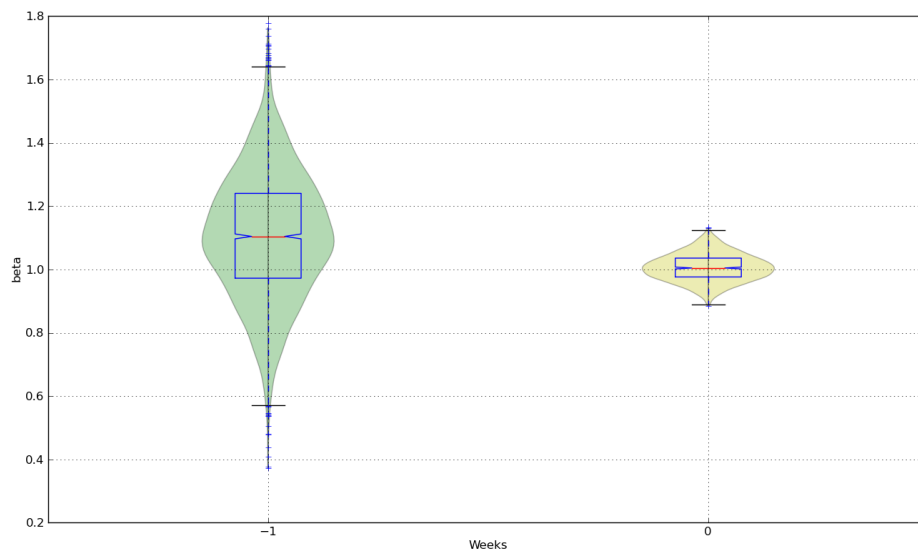
Figure 1.2: Parameters prior and posterior distributions.

```python
from BIP.Bayes.Melding import FitModel
import numpy as np
from scipy import stats as st


mu = 0.0 #birth and death rate.FIXED
beta = 0.00058 #Transmission rate
eta = .5 #infectivity of asymptomatic infections relative to clinical ones. FIXED
epsilon = .1 #latency period
alpha = .2 #Probability of developing clinical influenza symptoms
sigma = .5 #reduced risk of re-infection after recovery
tau = .01 #infectious period. FIXED
# Initial conditions
global inits,tf
tf= 140
inits = [490,0,10,0,0]
pars = [beta,alpha,sigma]


# propensity functions
def f1(r,inits):return r[0]*inits[0]*(inits[2]+inits[3])#S->E
def f2(r,inits):return r[1]*inits[1]#E->I
def f3(r,inits):return r[3]*inits[2]#I->R
def f4(r,inits):return r[2]*inits[1]#E->A
def f5(r,inits):return r[4]*inits[3]#A->R

def runModel(theta):
    global tf,inits
    step = 1
    #setting parameters
    beta,alpha,sigma = theta[:3]
    vnames = ['S','E','I','A','R']
    #rates: b,ki,ka,ri,ra
    #r = (0.001, 0.1, 0.1, 0.01, 0.01)
    r = (beta, alpha*epsilon, (1-alpha)*epsilon, tau, tau)
    #print r,inits
    # propensity functions
```

```
    propf = (f1,f2,f3,f4,f5)

    tmat = np.array([[-1, 0, 0, 0, 0],
                     [ 1,-1, 0,-1, 0],
                     [ 0, 1,-1, 0, 0],
                     [ 0, 0, 0, 1,-1],
                     [ 0, 0, 1, 0, 1]
                    ])
    M=Model(vnames=vnames,rates = r,inits=inits,tmat=tmat,propensity=propf)
    #t0 = time.time()
    M.run(tmax=tf,reps=1,viz=0,serial=True)
    t,series,steps,events = M.getStats()
    ser = st.nanmean(series,axis=0)
    #print series.shape, ser.shape
    return ser

d = runModel([beta,alpha,sigma])
#~ import pylab as P
#~ P.plot(d)
#~ P.show()

dt = {'S':d[:,0],'E':d[:,1],'I':d[:,2],'A':d[:,3],'R':d[:,4]}
F = FitModel(900, runModel,inits,tf,['beta','alpha','sigma'],['S','E','I','A','R'],
             wl=140,nw=1,verbose=0,burnin=100)
F.set_priors(tdists=[st.uniform]*3,tpars=[(0.00001,.0006),(.1,.5),(0.0006,1)],tlims=[(0,.001),(.00
             pdists=[st.uniform]*5,ppars=[(0,500)]*5,plims=[(0,500)]*5)

F.run(dt,'MCMC',likvar=1e1,pool=0,monitor=[])
#~ print F.optimize(data=dt,p0=[0.1,.5,.1], optimizer='oo',tol=1e-55, verbose=1, plot=1)
#==Uncomment the line below to see plots of the results
F.plot_results()
```

This example can be found in the examples folder of BIP under the name of `flu_stochastic.py`.

1.2.2 Iterative Estimation and Forecast

In some other types of application, one's data accrue gradually and it may be interesting to use newly available data to improve previously obtained parameter estimations.

Here we envision two types of scenarios: one assuming constant parameters and another where parameter values can actually vary with time. These two scenarios lead to the two fitting strategies depicted on figure

1.2.3 References

1.3 Stochastic Differential Equations

The SDE package in BIP, was born out of the need to simulate stochastic model to test the Parameters estimation routines in the Bayes Package. However, it is useful in many general-purpose application since it provides a pure Python implementation of an SDE solver.

Currently it provides a single solving algorithm, the Gillespie SSA. but other algorithms are planned for future releases
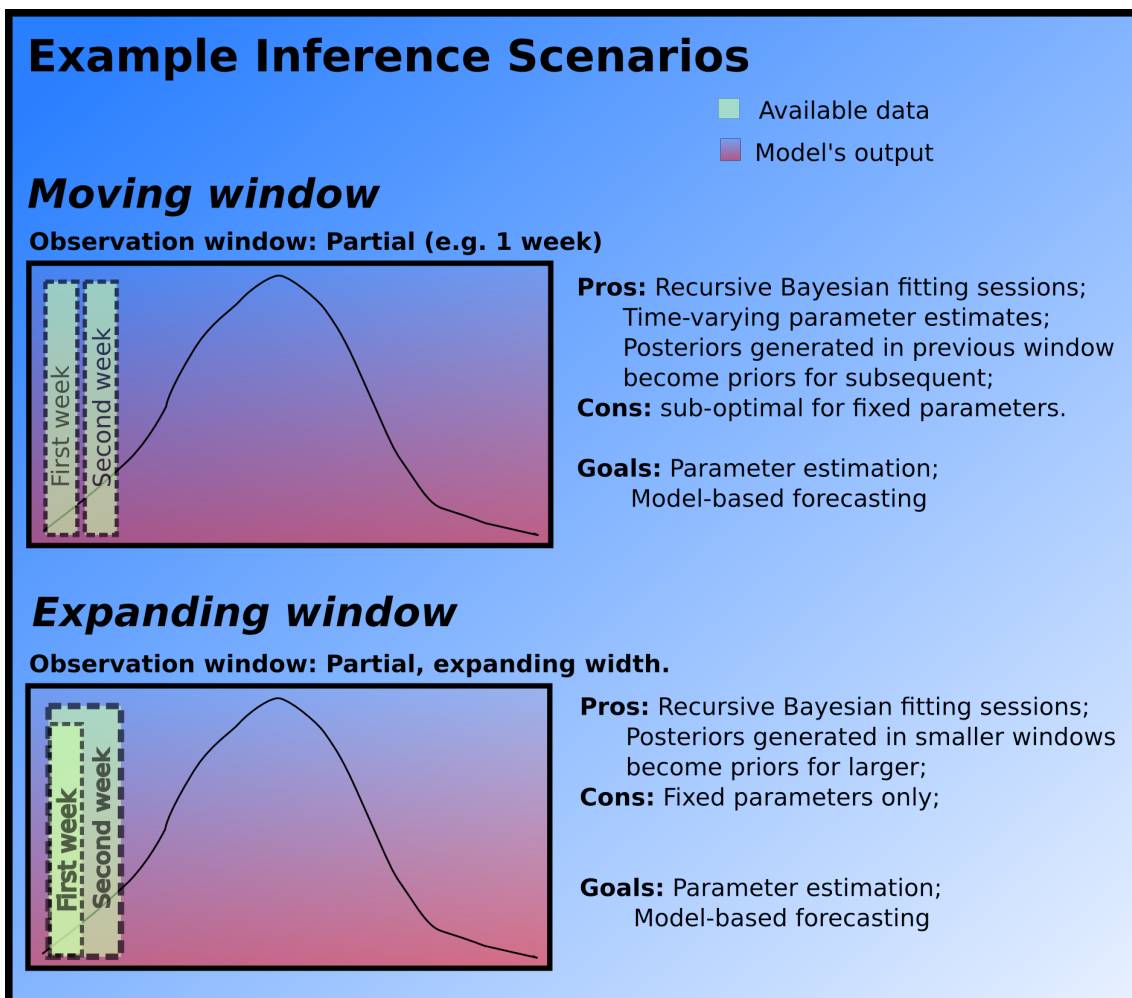
Figure 1.3: Fitting scenarios: Moving windows and expanding windows.

MODULES

`BIP`	Bayesian Inference Package containing usefull classes and functions for doing inference in various a
`BIP.Bayes`	Basic Likelihood tools such as functions for computing likelihoods, Latin Hypercube sampling (effi
`BIP.Bayes.Samplers`	

THREE

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*