Bayesian estimation with Markov chain Monte Carlo using PyMC

Christopher J. Fonnesbeck

Department of Biostatistics
Vanderbilt University School of Medicine

Tuesday, November 9, 2010
Bayesian Inference

Bayes' Theorem for 2 variables:

Let $E, F$ be events with $P(E) > 0$ and $P(F) > 0$.

Then

$$P(F|E) = \frac{P(E|F)P(F)}{P(E|F)P(F) + P(E|\overline{F})P(F)}$$

$$P(E|F) = \frac{P(EF)}{P(F)} \quad \Rightarrow \quad P(E|F)P(F) = P(EF)$$

$$P(F|E) = \frac{P(EF)}{P(E)} \quad \Rightarrow \quad P(F|E)P(E) = P(EF)$$

$$P(E|F)P(F) = P(F|E)P(E)$$

$$P(E|F) = \frac{P(F|E)P(E)}{P(F)}$$
Conclusions in terms of probability statements

\[ p(\theta | y) \]
Classical inference conditions on unknown parameter

\[ p(y|\theta) \]
posterior distribution

unobserved quantities

\[ p(\theta | y) \]

observed data
Likelihood Function

\[ p(y|\theta) = L(\theta|y) \]
posterior \quad likelihood \quad prior

\[ p(\theta | y) \propto p(y | \theta) p(\theta) \]
“For a given sample of data, any 2 probability models

\[ p(y|\theta) \]

that have the same likelihood function yield the same inference for \( \theta \).”

Gelman et al., 2004
$p(\theta \mid y) \propto p(y \mid \theta)p(\theta)$
subjectivity
“If two Bayesians get different posteriors [from the same data], at least one has made a provable error.”

Joseph Kadane 2008
Between 1745 and 1770 there were 241,945 girls and 251,527 boys born in Paris.
Classical vs Bayesian Statistics
“... the Bayesian approach is attractive because it is useful. Its usefulness derives in large measure from its simplicity. Its simplicity allows the investigation of far more complex models than can be handled by the tools in the classical toolbox.”

Link and Barker (2010)
Bayesian computation
Example: binomial model

\[ p(\theta | y) \propto p(y | \theta)p(\theta) \]

\[ = \theta^y (1 - \theta)^{n-y} \]

\[ \times \theta^{\alpha - 1} (1 - \theta)^{\beta - 1} \]

(Beta prior)
Example: binomial model

\[ p(\theta \mid y) \propto p(y \mid \theta)p(\theta) \]
\[ = \theta^y (1 - \theta)^{n-y} \]
\[ \times \theta^{\alpha-1} (1 - \theta)^{\beta-1} \]

(Beta prior)

\[ = \theta^{y+\alpha-1} (1 - \theta)^{n-y+\beta-1} \]
\[ = \theta^{y+\alpha-1} (1 - \theta)^{n-y+\beta-1} \]

\[ \theta | y \sim \text{Beta}(y + 1, n - y + 1) \]
\[ Pr(\theta|y) = \frac{Pr(y|\theta) Pr(\theta)}{Pr(y)} = \frac{Pr(y|\theta) Pr(\theta)}{\int_{\theta} Pr(y|\theta) Pr(\theta) d\theta} \]
Markov chain Monte Carlo (MCMC)
Markov chain simulation

\[ \Theta \rightarrow \pi \rightarrow \text{sample} \rightarrow \text{parameter values} \rightarrow \text{improve} \rightarrow \text{posterior distribution} \]
detailed balance equation

\[ \pi(x) Pr(y \mid x) = \pi(y) Pr(x \mid y) \]

“reversible” Markov chain
model {
  for (i in 1:N) {
    r[i] ~ dbin(p[i], n[i])
    b[i] ~ dnorm(0, tau)
    logit(p[i]) <- alpha0 + alphal * x1[i] + alphal2 * x1[i] * x2[i] + b[i]
  }
  alpha0 ~ dnorm(0, 1.0E-6)
  alphal ~ dnorm(0, 1.0E-6)
  alphal2 ~ dnorm(0, 1.0E-6)
  tau ~ dgamma(0.001, 0.001)
  sigma <- 1 / sqrt(tau)
}
OpenBUGS

Overview

Overview...

BUGS is a software package for performing Bayesian inference Using Gibbs Sampling. The user specifies a statistical model, of (almost) arbitrary complexity, by simply stating the relationships between related variables. The software includes an 'expert system', which determines an appropriate MCMC (Markov chain Monte Carlo) scheme (based on the Gibbs sampler) for analysing the specified model. The user then controls the execution of the scheme and is free to choose from a wide range of output types.

 Versions...

There are two main versions of BUGS, namely WinBUGS and OpenBUGS. This site is dedicated to OpenBUGS, an open-source version of the package, on which all future development work will be focused. OpenBUGS, therefore, represents the future of the BUGS project. WinBUGS, on the other hand, is an established and stable, stand-alone version of the software, which will remain available but not further developed. The latest versions of OpenBUGS (from v3.0.7 onwards) have been designed to be at least as efficient and reliable as WinBUGS over a wide range of test applications. Please see here for more information on WinBUGS. OpenBUGS runs on x86 machines with MS Windows, Unix/Linux or Macintosh (using Wine).

For additional details on the differences between OpenBUGS and WinBUGS see the OpenVsWin manual page.

How it works...

The specified model belongs to a class known as Directed Acyclic Graphs (DAGs), for which there exists an elegant underlying mathematical theory. This allows us to break down the analysis of arbitrarily large and complex structures into a sequence of relatively simple computations. BUGS includes a range of algorithms that its expert system can assign to each such computational task. One of the main differences between OpenBUGS and WinBUGS is the way in which the expert system makes its decisions. WinBUGS defines one algorithm for each possible computation type whereas there is no limit to the number of algorithms that OpenBUGS can make use of, making for much greater flexibility and extensibility.
NumPy is the fundamental package needed for scientific computing with Python. It contains among other things:

- a powerful N-dimensional array object
- sophisticated (broadcasting) functions
- tools for integrating C/C++ and Fortran code
- useful linear algebra, Fourier transform, and random number capabilities.

Besides its obvious scientific uses, NumPy can also be used as an efficient multi-dimensional container of generic data. Arbitrary data-types can be defined. This allows NumPy to seamlessly and speedily integrate with a wide variety of databases.

**Getting Started**

- Getting Numpy
- Installing NumPy and SciPy
- NumPy and SciPy documentation page
- NumPy Tutorial
- NumPy for MATLAB® Users
- NumPy functions by category
- NumPy Mailing List

**More Information**

- NumPy Sourceforge Home Page
- SciPy Home Page
- Interfacing with compiled code
- Older python array packages
Database Backends
SUBROUTINE t(x,nu,n,nnu,like)

  c Student's t log-likelihood function

  cf2py integer dimension(n),intent(in) :: x
  cf2py double precision dimension(nnu),intent(in) :: nu
  cf2py double precision intent(out) :: like
  cf2py integer intent(hide),depend(x) :: n=len(x)
  cf2py integer intent(hide),depend(nu) :: nnu=len(nu)
  cf2py threadsafe

IMPLICIT NONE
INTEGER n, i, nnu
INTEGER x(n)
DOUBLE PRECISION nu(nnu), like infinity, nut
PARAMETER (nu(i) = 1.5941136377940923d0)
DOUBLE PRECISION gammaln
DOUBLE PRECISION PI
PARAMETER (PI=3.141592653589793238462643d0)

nut = nu(1)

like = 0.0
do i=1,n
  if (nnu .GT. 1) then
    nut = nu(i)
  end if
  
end do

from pymc import *
from numpy import ones, array

n = 5*ones(4, dtype=int)
dose = array([-0.86, -0.3, -0.05, 0.73])
response = array([0, 1, 3, 5])

# Priors on unknown parameters
alpha = Normal('alpha', mu=0.0, tau=0.01)
beta = Normal('beta', mu=0.0, tau=0.01)

@deterministic
def theta(a=alpha, b=beta, d=dose):
    """theta = inv_logit(a+b)""
    return invlogit(a+b*d)

# deaths ~ binomial(n, p)
deaths = Binomial('deaths', n=n, p=theta, value=response, observed=True)
>>> from pymc import *
>>> import bioassay

>>> M = MCMC(bioassay, db='sqlite')
>>> M.sample(iter=10000, burn=5000)
>>> Matplot.plot(M)
Maplotlib

95% Credible Intervals

theta [1] [2] [3] [4]

R-hat

0.0 0.2 0.4 0.6 0.8 1.0

1 1.5 2+

Tuesday, November 9, 2010
PyMC Objects
Stochastic
automatic initialisation
>>> s = DiscreteUniform('s', 1851, 1962, value=1900)

>>> s.logp
-4.7184988712950942

>>> s.random()
1922

>>> s.value
array(1922)

>>> s.logp
-4.7184988712950942
decorator initialisation

```python
@stochastic(dtype=int)
def s(value=1900, t_l=1851, t_h=1962):
    """The switchpoint for the rate of disaster occurrence."""

    if value > t_h or value < t_l:
        # Invalid values
        return -numpy.inf
    else:
        # Uniform log-likelihood
        return -numpy.log(t_h - t_l + 1)
```
decorator initialisation

```python
@stochastic(dtype=int)
def s(value=1900, t_l=1851, t_h=1962):
    def logp(value, t_l, t_h):
        if value > t_h or value < t_l:
            return -Inf
        else:
            return -log(t_h - t_l + 1)
    def random(t_l, t_h):
        return round( (t_l - t_h) * random() ) + t_l
```
direct initialisation

def s_logp(value, t_l, t_h):
    if value > t_h or value < t_l:
        return -Inf
    else:
        return -log(t_h - t_l + 1)

def s_rand(t_l, t_h):
    return round((t_l - t_h) * random() ) + t_l

s = Stochastic(doc = 'The switchpoint for the rate of disaster occurrence."
    logp = s_logp,
    random = s_rand,
    name = 's',
    parents = {'t_l': 1851, 't_h': 1962},
    value = 1900,
    dtype=int)
Data
e.g. Poisson counts

\[ x = (3, 2, 2, 1, 4, 2, 1, 3) \]

\[ x \sim \text{Poisson}(\theta) \]
# Some count data
count_data = [3, 2, 2, 1, 4, 2, 1, 3]

# Poisson mean
mu = Gamma('mu', alpha=0.01, beta=0.01)

# Poisson likelihood
counts = Poisson('counts', mu=mu, value=count_data, observed=True)
# Some count data
count_data = [3, 2, 2, 1, 4, 2, 1, 3]

# Poisson mean
mu = Gamma('mu', alpha=0.01, beta=0.01)

# Poisson likelihood
counts = Poisson('counts', mu=mu, value=count_data, observed=True)
# Some count data
count_data = [3, 2, 2, 1, 4, 2, 1, 3]

# Poisson mean
mu = Gamma('mu', alpha=0.01, beta=0.01)

# Poisson likelihood
@observed
def counts(value=count_data, mu=mu):
    """counts ~ poisson(mu)"
    return poisson_like(value, mu)
# Some count data
count_data = [3, 2, 2, 1, 4, 2, 1, 3]

# Poisson mean
mu = Gamma('mu', alpha=0.01, beta=0.01)

# Poisson likelihood
counts = Poisson('counts', mu=mu, value=count_data, observed=True)
>>> counts.children
set([])

>>> counts.parents
{'mu': <pymc.distributions.Gamma 'mu' at 0x57a3970>}

>>> mu.children
set([<pymc.distributions.Poisson 'counts' at 0x5de67d0>])

>>> mu.parents
{'alpha': 0.01, 'beta': 0.01}
>>> mu.logp
-5.3517472886562549

>>> counts.logp
-12.364365060404872
>>> counts.value
array([3, 2, 2, 1, 4, 2, 1, 3])

>>> mu.value
array(2.0)

>>> mu.random()
1.587314860448013e-07

>>> mu.value
array(1.587314860448013e-07)
Missing Data
# Missing values indicated by None placeholders

count = np.array([4, 5, 4, 0, 1, 4, 3, 4, 0, 6, 3, 3, 4, 0, 2, 6, 3, 3, 5, 4, 5, 3, 1, 4, 4, 1, 5, 5, 3, 4, 2, 5, 2, 2, 3, 4, 2, 1, 3, -999, 2, 1, 1, 1, 1, 3, 0, 0, 1, 0, 1, 0, 0, 3, 1, 0, 3, 2, 2, 0, 1, 1, 1, 0, 1, 0, 1, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 1, 0, 1])

# Mean count
m = Exponential('mu', beta=1)

# Create masked array with data
masked_values = masked_array(count, mask=count===-999)

# Pass masked array to data stochastic, and it does the right thing
D = Poisson('D', mu=m, value=masked_values, observed=True)
Deterministic
decorator initialisation

```python
@deterministic
def theta(a=alpha, b=beta, x=data):
    '''theta = invlogit(a + b*x)'''
    return invlogit(a + b*x)
```
automatic initialisation

```python
def theta(a, b):
    return lambda alpha, beta, x: invlogit(a + b*x)
```
Lambda functions

```python
>>> div_by_two = lambda x : x % 2 == 0
>>> div_by_two(7)
False
>>> div_by_two(16)
True
```
Factor Potential
arbitrary log-probability terms

\[ p(\theta, \phi | y) \propto p(y | \theta)p(\theta | \phi)p(\phi) \]
\[ \downarrow \]
\[ p(\theta, \phi, \tau | y) \propto p(y | \theta)p(\theta | \phi)p(\theta | \tau)p(\tau)p(\phi) \]
```python
@potential

def mean_zero(y=theta, mu=0, tau=tau):
    """mean_zero = normal(y, mu, tau)""
    return normal_like(y, mu, tau)
```
Containers
```python
x_0 = Normal('x_0', mu=0, tau=1)

# Initialize array of stochastics
x = [x_0]

# Loop over number of elements in N
for i in range(1,N):
    # Instantiate Normal stochastic
    # based on value of previous element in x
    xi = Normal('x_%i' % i, mu=x[-1], tau=1)
    # Append to x
    x.append(xi)
```
Fitting Models

\[
\frac{\Gamma(k + x)}{\Gamma(k) x!} \left( \frac{\Gamma(\alpha + \beta)}{\Gamma(\alpha) \Gamma(\beta)} x^{\alpha-1} (1 - x)^{\beta-1} \right)
\]
1. Approximations
2. Samplers
MAP
Normal Approximation
Sampler
Sampler

- sample(iter, length=None, verbose=0)
- isample(iter, length=None, verbose=0)
- tally()
- stats(alpha=0.05, start=0)
- pause()
- trace(name, chain=-1)
- save_state()
- get_state()

MCMC

- use_step_method(step_method_class)
- assign_step_methods()
- sample(iter, burn=0, thin=1,
  tune_interval=1000, tune_throughout=True,
  save_interval=None, verbose=0)
- tune()
- dic()
Step Methods
Smoltz:~ fonnesbeck\$ ipython
Python 2.6.1 (r261:67515, Jul 7 2009, 23:51:51)
Type "copyright", "credits" or "license" for more information.

IPython 0.11.bzr.r1205 -- An enhanced Interactive Python.
?       -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help     -> Python's own help system.
object?  -> Details about 'object'. ?object also works, ?? prints more.

In [1]:  


Performance
caching
L = LazyFunction(fun, arguments)
vectorized functions

```python
>>> x = [2.1, 6.5, -1.0]

>>> p = invlogit(x)

>>> probs = Beta('probs', alpha=1, beta=1, value=p, observed=True)

>>> probs.value
array([ 0.89090318,  0.99849882,  0.26894142])
```
Directed Acyclic Graphs (DAGs)

>>> graph.dag(M)
Disease Dynamics

Zipkin et al, 2010

Methods in Ecology and Evolution
prevalence
incidence
force of infection
infection probability
recovery probability
Markov models of disease dynamics
Markovian dependence

\[ Pr(X_{t+1} = x_{t+1}|X_t = x_t, X_{t-1} = x_{t-1}, \ldots, X_0 = x_0) = Pr(X_{t+1} = x_{t+1}|X_t = x_t) \]
\[ P = \begin{bmatrix} p_{00} & p_{01} & p_{02} \\ p_{10} & p_{11} & p_{12} \\ 0 & 0 & 1 \end{bmatrix} \]
\[ P = \begin{bmatrix}
  p_{00} & p_{01} & p_{02} \\
  p_{10} & p_{11} & p_{12} \\
  0 & 0 & 1
\end{bmatrix} \]

transition matrix
\[ X_{t+1} = X_t P \]

where \( X_t = (S_t, I_t, D_t) \)
How long until initial infection?
\[ f_{01}^{(2)} = Pr\{X_{t+2} = 1, X_{t+1} = 0 | X_t = 0\} = p_{00}p_{01} \]

2 steps
$f_{01}^{(m)} = Pr\{X_{t+m} = 1, X_{t+m-1} = 0, \ldots, X_{n+1} = 0 | X_n = 0\} = p_{00}^{m-1} p_{01}$

$m$ steps
$$Pr(0 \rightarrow 1) = \sum_{m=1}^{\infty} p_{00}^{m-1} p_{01}$$

$$= p_{01} + p_{00}p_{01} + p_{00}^2 p_{01} + \ldots$$

$$= p_{01} \sum_{i=0}^{\infty} p_{00}^i$$

$$= \frac{p_{01}(1 - p_{00}^{\infty})}{1 - p_{00}} = \frac{p_{01}}{1 - p_{00}}$$
Life Expectancy
Recurrent sub-matrix

\[ Q = \begin{pmatrix} p_{00} & p_{01} \\ p_{10} & p_{11} \end{pmatrix} \]
Expected time to absorption

\[ W = (I - Q)^{-1} \cdot \begin{pmatrix} 1 \\ 1 \end{pmatrix} \]
\[ P = \begin{bmatrix}
0.90 & 0.05 & 0.05 \\
0.10 & 0.70 & 0.20 \\
0 & 0 & 1
\end{bmatrix} \]

0 = susceptible
1 = infected
2 = dead
categorical probabilities

\( (x_{i,0}, x_{i,1}, x_{i,2}) \sim \Gamma(\alpha = 0.01, \beta = 0.01) \)

\[ p_{i,j} = \frac{x_{i,j}}{\sum_j x_{i,j}} \]

\( y^{(t)} \sim \text{Cat}(p_{y^{(t-1)},0}, p_{y^{(t-1)},1} \cdot p_{y^{(t-1)},2}) \)
# Create identity matrix for calculating life expectancies
Id = eye(2)

m0 = Gamma('m0', alpha=1, beta=0.01, value=ones(3))
m1 = Gamma('m1', alpha=1, beta=0.01, value=ones(3))

@deterministic
def P(m0=m0, m1=m1):
    """Transition matrix"
    return array([[m0/sum(m0), m1/sum(m1), [0, 0, 1]]])

@observed
def encounters(value=y, P=P):
    """Categorical likelihood for encounters"
    return sum([[categorical_like(value[i,t], P[value[i,t-1]]) for t in range(1,T)] for i in range(N)])

# Calculated nodes

"""Probability of moving from susceptible (0) to infected (1) over course of study period and expected time to first transition"
Pr01 = Lambda('Pr01', lambda P: P[0,1]/(1-P[0,0]))
Ex01 = Lambda('Ex01', lambda P: 1/(1-P[0,0]))

"""Probability of moving from infected (1) to susceptible (0) over course of study period and expected time to first transition"
Pr10 = Lambda('Pr10', lambda P: P[1,0]/(1-P[1,1]))
Ex10 = Lambda('Ex10', lambda P: 1/(1-P[1,1]))

# Life expectancy
# Create identity matrix for calculating life expectancies
Id = eye(2)

m0 = Gamma('m0', alpha=1, beta=0.01, value=ones(3))
m1 = Gamma('m1', alpha=1, beta=0.01, value=ones(3))

@deterministic
def P(m0=m0, m1=m1):
    """Transition matrix""
    return array([[m0/sum(m0), m1/sum(m1), [0, 0, 1]]])

@observed
def encounters(value=y, P=P):
    """Categorical likelihood for encounters""
    return sum([[categorical_like(value[i,t], P[value[i,t-1]]) for t
    in range(N)]]

# Calculated nodes

"""Probability of moving from susceptible (0) to infected (1) over course of study period and expected time to first transition"""
# Calculated nodes

"""Probability of moving from susceptible (0) to infected (1) over course of study period and expected time to first transition"

Pr01 = Lambda('Pr01', lambda P: P[0,1]/(1-P[0,0]))
Ex01 = Lambda('Ex01', lambda P: 1/(1-P[0,0]))

"""Probability of moving from infected (1) to susceptible (0) over course of study period and expected time to first transition"

Pr10 = Lambda('Pr10', lambda P: P[1,0]/(1-P[1,1]))
Ex10 = Lambda('Ex10', lambda P: 1/(1-P[1,1]))

# Life expectancy

Q = Lambda('Q', lambda P: P[:2, :2])
IminusQ = Lambda('IminusQ', lambda Q: Id-Q)

InvIminusQ = Lambda('InvIminusQ', lambda IQ=IminusQ: inverse(IQ))

W = Lambda('W', lambda IIQ=InvIminusQ, IQ=IminusQ: [IIQ[i,0]+IQ[i,1] for i in range(2)])
# Calculated nodes

"""Probability of moving from susceptible (0) to infected (1) over course study period and expected time to first transition"
Pr01 = Lambda('Pr01', lambda P=P: P[0,1]/(1-P[0,0]))
Ex01 = Lambda('Ex01', lambda P=P: 1/(1-P[0,0]))

"""Probability of moving from infected (1) to susceptible (0) over course study period and expected time to first transition"
Pr10 = Lambda('Pr10', lambda P=P: P[1,0]/(1-P[1,1]))
Ex10 = Lambda('Ex10', lambda P=P: 1/(1-P[1,1]))

# Life expectancy
Q = Lambda('Q', lambda P=P: P[:2,:2])
IminusQ = Lambda('IminusQ', lambda Q=Q: Id-Q)

InvIminusQ = Lambda('InvIminusQ', lambda IQ=IminusQ: inverse(IQ))

W = Lambda('W', lambda IIQ=InvIminusQ, IQ=IminusQ: [IIQ[i,0]+IQ[i,1] for
susceptible state transitions
infected state transitions
life expectancy
$P =$

$$
\begin{bmatrix}
0.90 & 0.05 & 0 & 0.05 \\
0 & 0.70 & 0.10 & 0.20 \\
0 & 0 & 0.95 & 0.05 \\
0 & 0 & 0 & 1
\end{bmatrix}
$$

0 = susceptible
1 = infected
2 = immune
3 = dead
m0 = Gamma('m0', alpha=1, beta=0.01, value=ones(4))
m1 = Gamma('m1', alpha=1, beta=0.01, value=ones(4))
m2 = Gamma('m2', alpha=1, beta=0.01, value=ones(4))

@deterministic
def P(m0=m0, m1=m1, m2=m2):
  """Transition matrix""
  return array([[m0/sum(m0), m1/sum(m1), m2/sum(m2), [0, 0, 0, 1]]])

@observed
def encounters(value=y, P=P):
  """Categorical likelihood for encounters""
  return sum([[categorical_like(value[i,t], P[value[i,t-1]]) for t in range(1,T)] for i in range(N)])

# Calculated nodes

"""Probability of moving from susceptible (0) to infected (1) over course of study period and expected time to first transition"
Pr01 = Lambda('Pr01', lambda P: P[0,1]/(1-P[0,0]))
Ex01 = Lambda('Ex01', lambda P: 1/(1-P[0,0]))

"""Probability of moving from infected (1) to immune (2) over course of study period and expected time to first transition"
Pr12 = Lambda('Pr12', lambda P: P[1,2]/(1-P[1,1]))
Ex12 = Lambda('Ex12', lambda P: 1/(1-P[1,1]))

# Life expectancy
Q = Lambda('Q', lambda P: P[:3, :3])
IminusQ = Lambda('IminusQ', lambda Q: Id-Q)
InvIminusQ = Lambda('InvIminusQ', lambda IQ=IminusQ: inverse(IQ))
W = Lambda('W', lambda IQ=InvIminusQ: [sum(IQ[i]) for i in range(3)])
m0 = Gamma('m0', alpha=1, beta=0.01, value=ones(4))
m1 = Gamma('m1', alpha=1, beta=0.01, value=ones(4))
m2 = Gamma('m2', alpha=1, beta=0.01, value=ones(4))

@deterministic
def P(m0=m0, m1=m1, m2=m2):
    """Transition matrix"""
    return array([[m0/sum(m0), m1/sum(m1), m2/sum(m2), [0, 0, 0, 1]]])

@observed
def encounters(value=y, P=P):
    """Categorical likelihood for encounters"""
    return sum([[categorical_like(value[i,t], P[value[i,t-1]]) for t in range(N)]]

# Calculated nodes

"""Probability of moving from susceptible (0) to infected (1) over course of study period and expected time to first transition"""
Pr01 = Lambda('Pr01', lambda P=P: P[0,1]/(1-P[0,0]))
Ex01 = Lambda('Ex01', lambda P=P: 1/(1-P[0,0]))
life expectancy
Hierarchical Model
Infection probability model

\[ p^{(i)}_{01} = \pi^{(i)}_0 s_0 \]

\[ \text{logit}(\pi^{(i)}_{01}) = \alpha_0 + \alpha_1 x_i \]

\[ x_i = I(i \text{ is female}) \]
Recovery probability model

\[ p_{10}^{(i)} = \pi_{10}^{(i)} s_1 \]

\[ \text{logit}(\pi_{10}^{(i)}) = \beta_0 + \beta_1 x_i \]

\[ x_i = I(i \text{ is female}) \]
# Probability of infection
alpha0 = Normal('alpha0', mu=0.0, tau=0.01, value=0.0)
alpha1 = Normal('alpha1', mu=0.0, tau=0.01, value=0.0)
pi01 = Lambda('pi01', lambda a0=alpha0, a1=alpha1: [invlogit(a0), invlogit(a0+a1)])

# Probability of recovery
beta0 = Normal('beta0', mu=0.0, tau=0.01, value=0.0)
beta1 = Normal('beta1', mu=0.0, tau=0.01, value=0.0)
pi10 = Lambda('pi10', lambda b0=beta0, b1=beta1: [invlogit(b0), invlogit(b0+b1)])

# Probabilities of survival
gamma0 = Normal('gamma0', mu=0.0, tau=0.01, value=0.0)
gamma1 = Normal('gamma1', mu=0.0, tau=0.01, value=0.0)
s = Lambda('s', lambda g0=gamma0, g1=gamma1: [invlogit(g0), invlogit(g0+g1)])

@deterministic
def P(pi01=pi01, pi10=pi10, s=s):
    """Transition matrix""
    return array([[s[0]*(1-pi01[i]), s[0]*pi01[i], 1-s[0]],
                  [s[1]*pi10[i], s[1]*(1-pi10[i]), 1-s[1]],
                  [0, 0, 1]]) for i in range(2)]

@observed
def encounters(value=y, P=P):
    """Categorical likelihood for encounters""
    return sum([[categorical_like(value[i,t], P[sex[i],value[i,t-1]]) for t in range(1,T)] for i in range(N)]
# Probability of infection
alpha0 = Normal('alpha0', mu=0.0, tau=0.01, value=0.0)
alpha1 = Normal('alpha1', mu=0.0, tau=0.01, value=0.0)
pi01 = Lambda('pi01', lambda a0=alpha0, a1=alpha1: [invlogit(a0), invlogit(a0+a1)])

# Probability of recovery
beta0 = Normal('beta0', mu=0.0, tau=0.01, value=0.0)
beta1 = Normal('beta1', mu=0.0, tau=0.01, value=0.0)
pi10 = Lambda('pi10', lambda b0=beta0, b1=beta1: [invlogit(b0), invlogit(b0+b1)])

# Probabilities of survival
gamma0 = Normal('gamma0', mu=0.0, tau=0.01, value=0.0)
gamma1 = Normal('gamma1', mu=0.0, tau=0.01, value=0.0)
s = Lambda('s', lambda g0=gamma0, g1=gamma1: [invlogit(g0), invlogit(g0+g1)])

@detemjnistic
def P(pi01=pi01, pi10=pi10, s=s):
    """Transition matrix""
    return array([[s[0]*(1-pi01[i]), s[0]*pi01[i], 1-s[0]]),
pi10 = Lambda('pi10', lambda b0=beta0, b1=beta1: [invlogit(b0), invlogit(b0+b1)])

# Probabilities of survival
gamma0 = Normal('gamma0', mu=0.0, tau=0.01, value=0.0)
.gamma1 = Normal('gamma1', mu=0.0, tau=0.01, value=0.0)
s = Lambda('s', lambda g0=gamma0, g1=gamma1: [invlogit(g0), invlogit(g1)])

@deterministic
def P(pi01=pi01, pi10=pi10, s=s):
    """Transition matrix""
    return array([[s[0]*(1-pi01[i]), s[0]*pi01[i], 1-s[0]],
                  [s[1]*pi10[i], s[1]*(1-pi10[i]), 1-s[1]],
                  [0, 0, 1]]) for i in range(2)]

@observed
def encounters(value=y, P=P):
    """Categorical likelihood for encounters""
    return sum([categorical_like(value[i,t], P[sex[i],value[i,t]-range(1,T)]) for i in range(N)])
female effect on infection probability
female effect on recovery probability
Hidden Markov Model
Process Model

Observation Model

True State

Observed State
Process Model

Observation Model

True State

1

? 

Observed State

0

1

Pr(0|1)

Pr(1|1)

Pr(0|0)

Pr(1|0)
\[ \pi = Pr(y = 1 | z = 1) \]
\[ 1 - \pi = Pr(y = 0 | z = 1) \]

false negative
\( O = \begin{bmatrix}
1 & 0 & 0 \\
1 - \pi & \pi & 0 \\
0 & 0 & 1
\end{bmatrix} \)

observation process
# Probability of detection

```
pi = Beta('pi', alpha=1, beta=1)
```

```
@deterministic
def P(m0=m0, m1=m1):
    """Transition matrix""
    return array([m0/sum(m0), m1/sum(m1), [0, 0, 1]])
```

```
z_init = [[[max(i) for i in o] for o in obs]
```

```
@stochastic(dtype=int)
def z(value=z_init, P=P):
    """Categorical probabilities for true states""
    like = 0.0
    for i in range(N):
        for t in range(1,T):
            if not 0<=value[i,t]<3: return -inf
            like += categorical_like(value[i,t], P[value[i,t-1]])
    return like
```

```
@potential
```
```
def O(z=z, pi=pi):
    """Returns log-probability of observed states""
    M = log(array([[1, 0, 0], [1-pi, pi, 0], [0, 0, 1]]))
    like = 0.0
    for i in range(N):
        for t in range(T):
            for j in range(k):
                like += M[z[i,t], obs[i,t,j]]
    return like
```
# Probability of detection
pi = Beta('pi', alpha=1, beta=1)

@deterministic
def P(m0=m0, m1=m1):
    """Transition matrix""
    return array([m0/sum(m0), m1/sum(m1), [0, 0, 1]])

z_init = [[max(i) for i in o] for o in obs]
@stochastic(dtype=int)
def z(value=z_init, P=P):
    """Categorical probabilities for true states""
    like = 0.0
    for i in range(N):
        for t in range(1, T):
            if not 0<=value[i,t]<3: return -inf
            like += categorical_like(value[i,t], P[value[i,t-1]])
    return like
"""Categorical probabilities for true states""
like = 0.0
for i in range(N):
    for t in range(1,T):
        if not 0<=value[i,t]<3: return -inf
        like += categorical_like(value[i,t], P[value[i,t]])
return like

@potential
def O(z=z, pi=pi):
    """Returns log-probability of observed states""
    M = log(array([[1, 0, 0], [1-pi, pi, 0], [0, 0, 1]]))
    like = 0.0
    for i in range(N):
        for t in range(T):
            for j in range(k):
                like += M[z[i,t], obs[i,t,j]]
    return like
detection
pymc.googlecode.com

github.com/pymc-devs/pymc
Dirichlet processes
Gaussian processes
t-walk sampling
multiprocessor support
enhanced GOF
... and more!
Thanks to

Evan Cooch
David Huard
Anand Patil
Drew Conway