Blocker: random effects meta-analysis of clinical trials

Ported to PyMC by Abraham Flaxman, 4/18/2012, from http://www.openbugs.info/Examples/Blockers.html

Carlin (1992) considers a Bayesian approach to meta-analysis, and includes the following examples of 22 trials of beta-blockers to prevent mortality after myocardial infarction.

In a random effects meta-analysis we assume the true effect (on a log-odds scale) d_i in a trial i is drawn from some population distribution. Let r_i^C denote number of events in the control group in trial i, and r_i^T denote events under active treatment in trial i. Our model is:

$$
\begin{aligned}
r_i^C &\sim \text{Binomial}\left(p_i^C, n_i^C\right), \\
r_i^T &\sim \text{Binomial}\left(p_i^T, n_i^T\right), \\
\text{logit}\left(p_i^C\right) &= \mu_i, \\
\text{logit}\left(p_i^T\right) &= \mu_i + \delta_i, \\
\delta_i &\sim \text{Normal}(d, t).
\end{aligned}
$$

"Noninformative" priors are given for μ_i, t, and d. We want to make inferences about the population effect d, and the predictive distribution for the effect δ_{new} in a new trial. Empirical Bayes methods estimate d and t by maximum likelihood and use these estimates to form the predictive distribution $p(\delta_{\text{new}}|\hat{d},\hat{t})$. Full Bayes allows for the uncertainty concerning d and t .

In [1]:
```
import pylab as pl
import pymc as mc
```

In [2]:
```
### data
r_t_obs = [3, 7, 5, 102, 28, 4, 98, 60, 25, 138, 64, 45, 9, 57, 25, 33, 28, 8,
n_t_obs = [38, 114, 69, 1533, 355, 59, 945, 632, 278,1916, 873, 263, 291, 858,
r_c_obs = [3, 14, 11, 127, 27, 6, 152, 48, 37, 188, 52, 47, 16, 45, 31, 38, 12
n_c_obs = [39, 116, 93, 1520, 365, 52, 939, 471, 282, 1921, 583, 266, 293, 883
N = len(n_c_obs)
```

Here is the BUGS model:

```
model
{
   for( i in 1 : Num ) {
      rc[i] ~ dbin(pc[i], nc[i])
      rt[i] ~ dbin(pt[i], nt[i])
      logit(pc[i]) <- mu[i]
      logit(pt[i]) <- mu[i] + delta[i]
      mu[i] ~ dnorm(0.0,1.0E-5)
      delta[i] ~ dnorm(d, tau)
   }
   d ~ dnorm(0.0,1.0E-6)
   tau ~ dgamma(0.001,0.001)
   delta.new ~ dnorm(d, tau)
   sigma <- 1 / sqrt(tau)
}
```

In [3]:
```
### hyperpriors
d = mc.Normal('d', 0., 1.e-6, value=0.)
```

```
tau = mc.Gamma('tau', 1.e-3, 1.e-3, value=1.)

sigma = mc.Lambda('sigma', lambda tau=tau: tau**-.5)
delta_new = mc.Normal('delta_new', d, tau, value=0.)


### priors
mu = [mc.Normal('mu_%d'%i, 0., 1.e-5, value=0.) for i in range(N)]
delta = [mc.Normal('delta_%d'%i, d, tau, value=0.) for i in range(N)]

p_c = mc.Lambda('p_c', lambda mu=mu: mc.invlogit(mu))
p_t = mc.Lambda('p_t', lambda mu=mu, delta=delta: mc.invlogit(array(mu)+delta))


### likelihood
r_c = mc.Binomial('r_c', n_c_obs, p_c, value=r_c_obs, observed=True)
r_t = mc.Binomial('r_t', n_t_obs, p_t, value=r_t_obs, observed=True)
```

BUGS uses Gibbs steps automatically, so it only takes 10000 steps of MCMC after a 1000 step burn in for this model in their example.

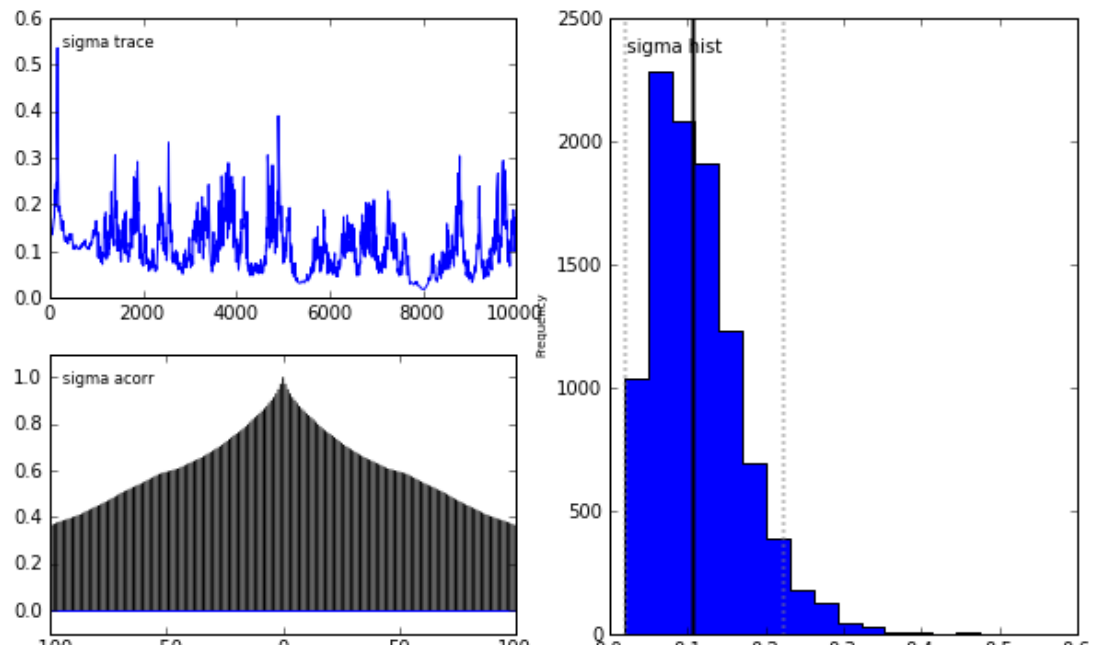
PyMC only uses Gibbs steps if you set them up yourself, and it uses Metropolis steps by default. So 10000 steps go by more quickly, but the chain takes longer to converge to the stationary distribution.

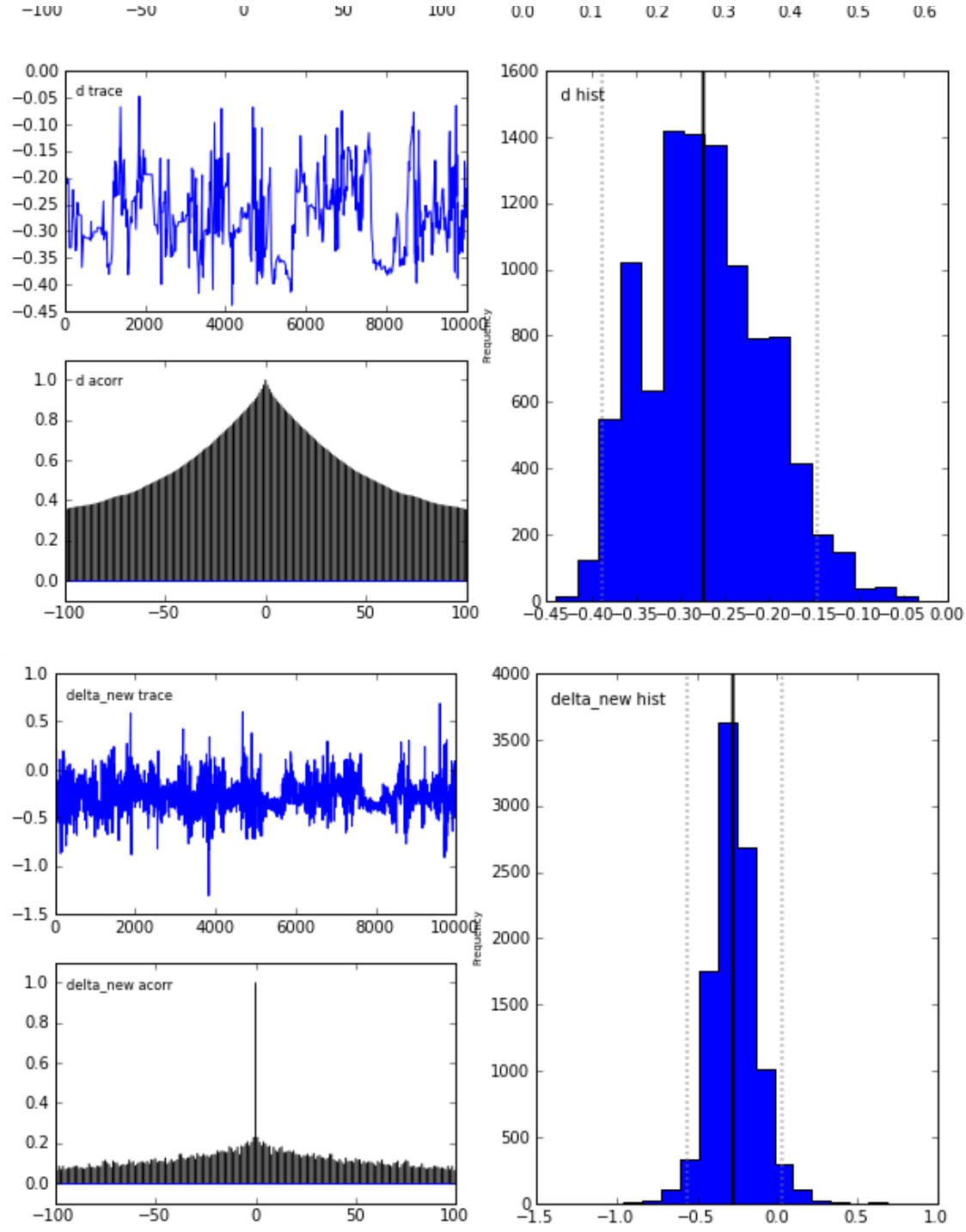
In [4]:
```
m = mc.MCMC([d, tau, sigma, delta_new, mu, delta, p_c, p_t, r_c, r_t])

# not long enough to mix
%time m.sample(11000, 1000, progress_bar=False)
```

```
CPU times: user 104.11 s, sys: 1.21 s, total: 105.31 s
Wall time: 123.44 s
```

In [5]:
```
mc.Matplot.plot(sigma)
mc.Matplot.plot(d)
mc.Matplot.plot(delta_new)
```

```
Plotting sigma
Plotting d
Plotting delta_new
```





Using some fancy step methods, and thinning the chain a little gives more reliable results

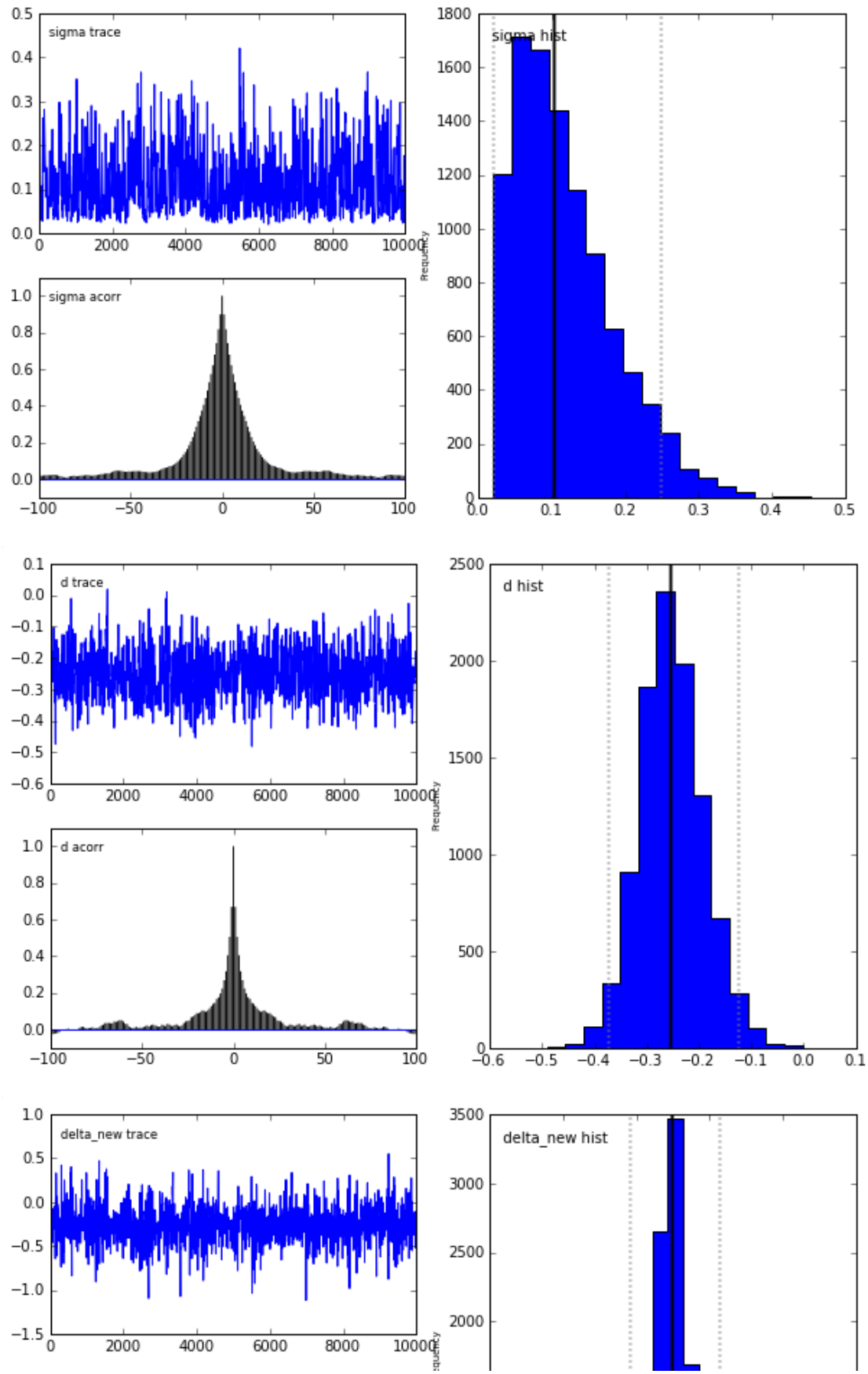
In [6]:
```
%time mc.MAP([d, tau, sigma, delta_new, mu, delta, p_c, p_t, r_c, r_t]).fit(me
m = mc.MCMC([d, tau, sigma, delta_new, mu, delta, p_c, p_t, r_c, r_t])

%time m.sample(110000, 10000, 10, progress_bar=False)
```

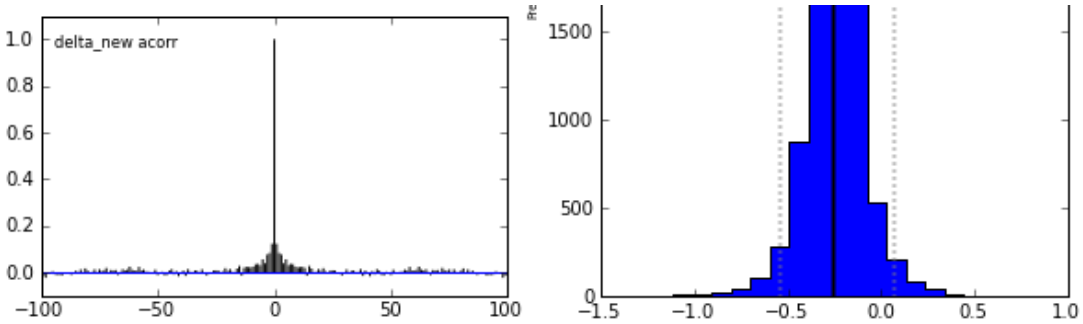
```
CPU times: user 3.36 s, sys: 0.03 s, total: 3.40 s
Wall time: 3.86 s
CPU times: user 1005.07 s, sys: 10.50 s, total: 1015.58 s
Wall time: 1165.24 s
```

In [7]:
```
mc.Matplot.plot(sigma)
```

```
mc.Matplot.plot(d)
mc.Matplot.plot(delta_new)
```

```
Plotting sigma
Plotting d
Plotting delta_new
```





Results

The OpenBUGS results table shows the following:

```
               mean     sd
d             -0.2492   0.06422
delta_new     -0.2499   0.1509
sigma          0.1189   0.07
```

To be compared with the following:

In [8]:
```
for node in [d, delta_new, sigma]:
    stats = node.stats()
    print '%10s\t%1.3f \t%.3f' % (node.__name__, stats['mean'], stats['standar
```

```
         d      -0.253  0.062
 delta_new      -0.254  0.146
     sigma      0.116   0.067
```

In [9]:
```
# use the following to generate plots of all stochastics
# (good for checking convergence in detail)
# mc.Matplot.plot(m)
```

Extensions

In some circumstances it might be reasonable to assume that the population distribution has heavier tails, for example a t-distribution with low degrees of freedom. This is easily accomplished in BUGS by using the `dt` distribution function instead of `dnorm` for d and d_{new}.

In PyMC, the relevant stochastic is `mc.NoncentralT`

In [10]:
```
### hyperpriors
d = mc.Normal('d', 0., 1.e-6, value=0.)
tau = mc.Gamma('tau', 1.e-3, 1.e-3, value=1.)

sigma = mc.Lambda('sigma', lambda tau=tau: tau**-.5)
delta_new = mc.NoncentralT('delta_new', d, tau, 4., value=0.)


### priors
mu = [mc.Normal('mu_%d'%i, 0., 1.e-5, value=0.) for i in range(N)]
delta = [mc.NoncentralT('delta_%d'%i, d, tau, 4., value=0.) for i in range(N)]

p_c = mc.Lambda('p_c', lambda mu=mu: mc.invlogit(mu))
```

```
p_t = mc.Lambda('p_t', lambda mu=mu, delta=delta: mc.invlogit(array(mu)+delta)


### likelihood
r_c = mc.Binomial('r_c', n_c_obs, p_c, value=r_c_obs, observed=True)
r_t = mc.Binomial('r_t', n_t_obs, p_t, value=r_t_obs, observed=True)
```

Our estimates are lower and with tighter precision - in fact similar to the values obtained by Carlin for the empirical Bayes estimator. The discrepancy appears to be due to Carlin's use of a uniform prior for s 2 in his analysis, which will lead to increased posterior mean and standard deviation for d, as compared to our (approximate) use of p(s 2) ~ 1 / s 2 (see his Figure 1).

It is easy to use a uniform prior on s^2 to compare results.

And how can we use PyMC to do empirical bayes? That requires going to Carlin's paper to see what is done there precisely.

In [ ]: