An Intro to Probabilistic Programming using JAGS

John Myles White

December 27, 2012

What I'll Assume for This Talk

- You know what Bayesian inference is:
  - Inference is the computation of a posterior distribution
  - All desired results are derived from the posterior
- You know what typical distributions look like:
  - Bernoulli / Binomial
  - Uniform
  - Beta
  - Normal
  - Exponential / Gamma
  - ...

What is Probabilistic Programming?

Probabilistic programming uses programming languages custom-designed for expressing probabilistic models

A Simple Example of JAGS Code

```
model
{
  alpha <- 1
  beta <- 1

  p ~ dbeta(alpha, beta)

  for (i in 1:N)
  {
    x[i] ~ dbern(p)
  }
}
```

Model Specification vs. Model Use

- Code in a probabilistic programming languages specifies a model, not a use case
- A single model specification can be reused in many contexts:
  - Monte Carlo simulations
  - Maximum A Posteriori (MAP) estimation
  - Sampling from a posterior distribution

Imperative versus Declarative Programming

- Describe a process for generating results:
  - C
  - Lisp
- Describe the structure of the results:
  - Prolog
  - SQL
  - Regular expressions

Existing Probabilistic Programming Systems

- The BUGS Family
  - WinBUGS
  - OpenBUGS
  - JAGS
  - Stan
- Infer.NET
- Church
- Factorie

JAGS Syntax: Model Blocks

```
model
{
   ...
}
```

JAGS Syntax: Deterministic Assignment

```
alpha <- 1
```

JAGS Syntax: Stochastic Assignment

```
p ~ dbeta(1, 1)
```

JAGS Semantics: Probability Distributions

- `dbern` / `dbinom`
- `dunif`
- `dbeta`
- `dnorm`
- `dexp` / `dgamma`
- `...`

JAGS Syntax: For Loops

```
for (i in 1:N)
{
  x[i] ~ dbern(p)
}
```

Putting It All Together

```
model
{
  alpha <- 1
  beta <- 1

  p ~ dbeta(alpha, beta)

  for (i in 1:N)
  {
    x[i] ~ dbern(p)
  }
}
```

An Equivalent Translation If N == 3

```
model
{
  p ~ dbeta(1, 1)

  x[1] ~ dbern(p)
  x[2] ~ dbern(p)
  x[3] ~ dbern(p)
}
```

For Loops are not Sequential

```
model
{
  p ~ dbeta(1, 1)

  x[2] ~ dbern(p)
  x[3] ~ dbern(p)
  x[1] ~ dbern(p)
}
```

For Loops do not Introduce a New Scope

```
model
{
  for (i in 1:N)
  {
    x[i] ~ dnorm(mu, 1)
    epsilon ~ dnorm(0, 1)
    x.pair[i] <- x[i] + epsilon
  }

  mu ~ dunif(-1000, 1000)
}
```

A Translation of a Broken For Loop

```
model
{
  x[1] ~ dnorm(mu, 1)
  epsilon ~ dnorm(0, 1)
  x.pair[1] <- x[1] + epsilon

  x[2] ~ dnorm(mu, 1)
  epsilon ~ dnorm(0, 1)
  x.pair[2] <- x[2] + epsilon

  mu ~ dunif(-1000, 1000)
}
```

Stochastic vs. Deterministic Nodes

- Observed data *must* correspond to stochastic nodes
- All constants like `N` must be known at compile-time
- Deterministic nodes are nothing more than shorthand
  - A deterministic node can always be optimized out!
- Stochastic nodes are the essence of the program

Observed Data Must Use Stochastic Nodes

Two valid mathematical formulations:

$$y_i \sim \mathcal{N}(ax_i + b, 1)$$

$$y_i = ax_i + b + \epsilon_i$$

where $\epsilon_i \sim \mathcal{N}(0, 1)$

Observed Data Must Use Stochastic Nodes

Valid jags code:

```
y[i] ~ dnorm(a * x[i] + b, 1)
```

Invalid jags code:

```
epsilon[i] ~ dnorm(0, 1)
y[i] <- a * x[i] + b + epsilon[i]
```

What's Badly Missing from JAGS Syntax?

- if / else
- Can sometimes get away with a `dsum()` function
- Some cases would require a non-existent `dprod()` function

This is *NOT* Valid JAGS Code

```
model
{
  p ~ dbeta(1, 1)

  for (i in 1:N)
  {
    exp[i] ~ dexp(1)
    norm[i] ~ dnorm(5, 1)
    alpha[i] ~ dbern(p)
    x[i] ~ dsum(alpha[i] * exp[i],
                (1 - alpha[i]) * norm[i])
  }
}
```

But Valid Code Does Exist for Many Important Models!

- Linear regression
- Logistic regression
- Hierarchical linear regression
- Mixtures of Gaussians

Linear Regression

```
model
{
  a ~ dnorm(0, 0.0001)
  b ~ dnorm(0, 0.0001)

  tau <- pow(sigma, -2)
  sigma ~ dunif(0, 100)

  for (i in 1:N)
  {
    mu[i] <- a * x[i] + b
    y[i] ~ dnorm(mu[i], tau)
  }
}
```

Logistic Regression

```
model
{
  a ~ dnorm(0, 0.0001)
  b ~ dnorm(0, 0.0001)

  for (i in 1:N)
  {
    y[i] ~ dbern(p[i])
    logit(p[i]) <- a * x[i] + b
  }
}
```

Hierarchical Linear Regression

```
model
{
  mu.a ~ dnorm(0, 0.0001)
  mu.b ~ dnorm(0, 0.0001)
  ...

  for (j in 1:K)
  {
    a[j] ~ dnorm(mu.a, tau.a)
    b[j] ~ dnorm(mu.b, tau.b)
  }
  for (i in 1:N)
  {
    mu[i] <- a[g[i]] * x[i] + b[g[i]]
    y[i] ~ dnorm(mu[i], tau)
  }
}
```

Clustering via Mixtures of Normals

```
model
{
  p ~ dbeta(1, 1)

  mu1 ~ dnorm(0, 0.0001)
  mu2 ~ dnorm(1, 0.0001)

  tau <- pow(sigma, -2)
  sigma ~ dunif(0, 100)

  for (i in 1:N)
  {
    z[i] ~ dbern(p)
    mu[i] <- z[i] * mu1 + (1 - z[i]) * mu2
    x[i] ~ dnorm(mu[i], tau)
  }
}
```

MCMC in 30 Seconds

- If we can write down a distribution, we can sample from it
- Every piece of JAGS code defines a probability distribution
- But we have to use Markov chains to draw samples
- May require hundreds of steps to produce one sample
- MCMC == Markov Chain Monte Carlo

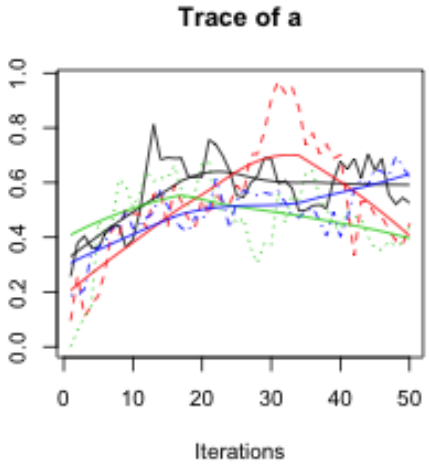
Using JAGS from R

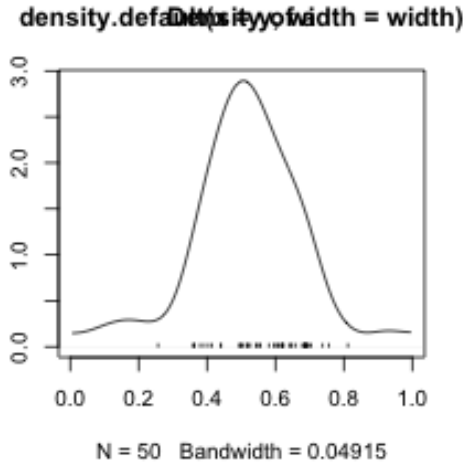
```
library("rjags")

jags <- jags.model("logit.bugs",
                   data = list("x" = x,
                               "N" = N,
                               "y" = y),
                   n.chains = 4,
                   n.adapt = 1000)

mcmc.samples <- coda.samples(jags, c("a", "b"), 5000)

summary(mcmc.samples)

plot(mcmc.samples)
```

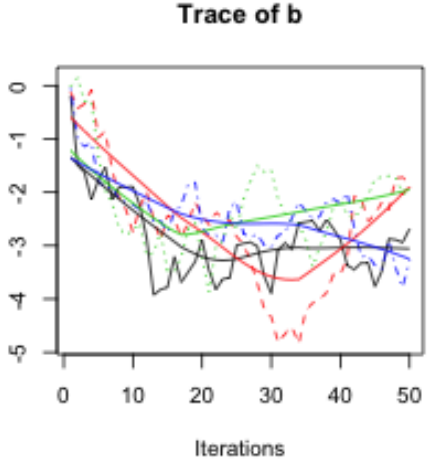
Summarizing the Results

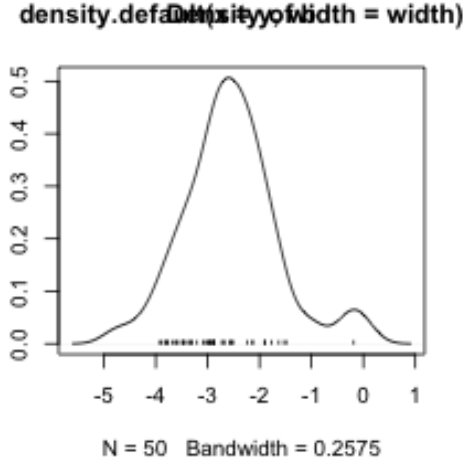
```
> summary(mcmc.samples)

Iterations = 1:50
Thinning interval = 1
Number of chains = 4
Sample size per chain = 50

1. Empirical mean and standard deviation for each variable,
   plus standard error of the mean:

     Mean     SD Naive SE Time-series SE
a  0.5504 0.1404 0.009929        0.01868
b -2.7125 0.8504 0.060134        0.10579

2. Quantiles for each variable:

     2.5%     25%     50%     75%   97.5%
a  0.2669  0.4669  0.5414  0.6144  0.8234
b -4.3907 -3.1135 -2.6842 -2.3118 -0.3475
```

Plotting the Samples









Burn-In

- Markov chains do not start in the right position
- Need time to reach and then settle down near MAP values
- The initial sampling period is called burn-in
- We discard all of these samples

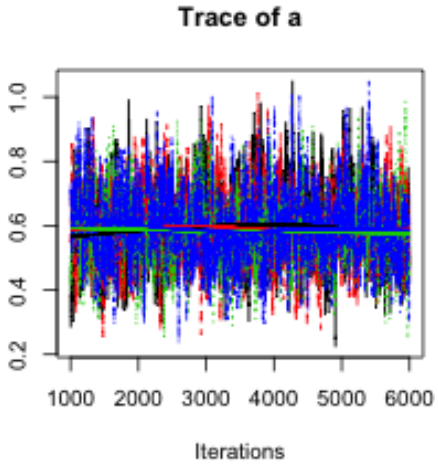
Adaptive Phase

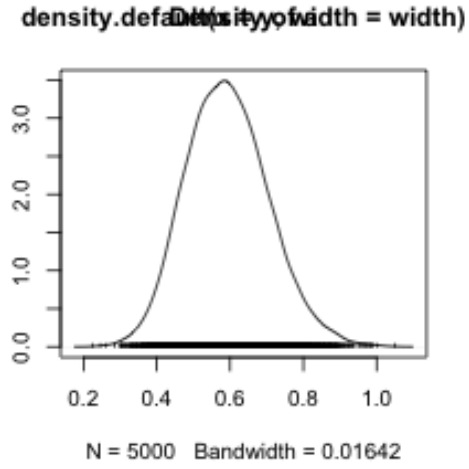
- JAGS has tunable parameters that need to adapt to the data
- Controlled by `n.adapt` parameter
- JAGS allows you to treat adaptive phase as part of burn-in
- For simple models, adaptation may not be necessary

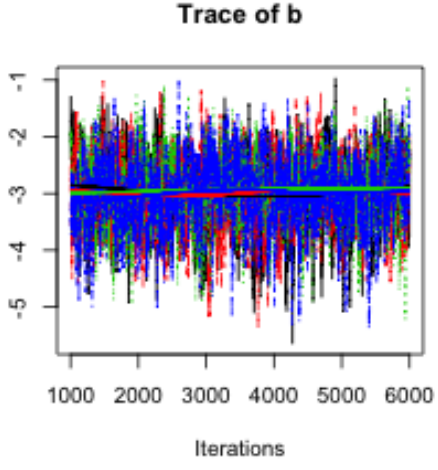
Mixing

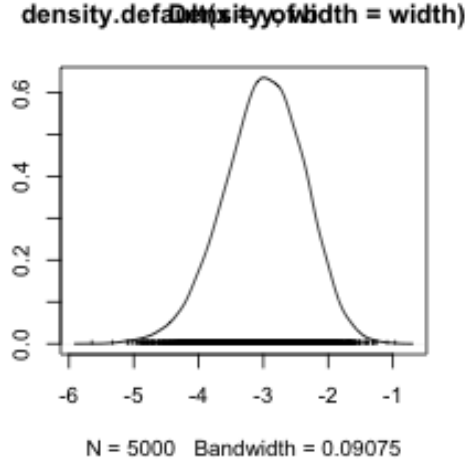
- How do we know that burn-in is complete?
- Run multiple chains
- Test that they all produce similar results
- All test for mixing are heuristic methods

Plotting the Samples after Burn-In









Other Practical Issues

- Autocorrelation between samples
- Thinning
- Initializing parameters
- Numeric stability

Additional References

- The BUGS Book
- The JAGS Manual
- My GitHub Repo of JAGS Examples
- 2012 NIPS Workshop on Probabilistic Programming

Appendix 1: Basic Sampler Design

- Slice sampler
- Metropolis-Hastings sampler
- Adaptive rejection sampling
- CDF inversion

Appendix 2: Gibbs Sampling

- Conjugacy
- Any closed form posterior