Bayesian Statistics I

Lecture 10 - Probabilistic programming for Bayesian inference

Mattias Villani

Department of Statistics
Stockholm University

Department of Computer and Information Science
Linköping University

mattiasvillani.com @matvil mattiasvillani

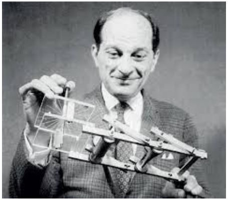
Lecture overview

- **Stan**
- **Turing.jl**

Stan

- Stan is a probabilistic programming language based on HMC.
- Allows for Bayesian inference in many models with automatic implementation of the MCMC sampler.
- Named after Stanislaw Ulam (1909-1984), co-inventor of the Monte Carlo algorithm.
- Written in C++ but can be run from R using the package `rstan`

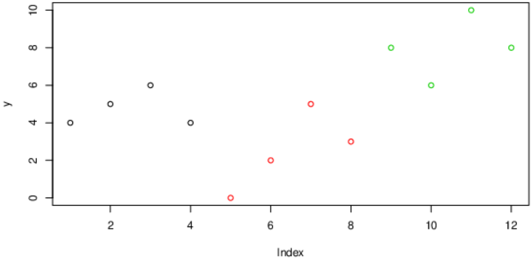
Stan logo



Stanislaw Ulam

Stan - toy example: three plants

- Three plants were observed for four months, measuring the number of flowers



Stan Model 1: iid Normal

$$y_i \overset{iid}{\sim} N\left(\mu, \sigma^2\right)$$

```
library(rstan)
y = c(4,5,6,4,0,2,5,3,8,6,10,8)
N = length(y)

StanModel = '
data {
  int<lower=0> N; // Number of observations
  int<lower=0> y[N]; // Number of flowers
}
parameters {
  real mu;
  real<lower=0> sigma2;
}
model {
  mu ~ normal(0,100); // Normal with mean 0, st.dev. 100
  sigma2 ~ scaled_inv_chi_square(1,2); // Scaled-inv-chi2 with nu 1, sigma 2
  for(i in 1:N)
    y[i] ~ normal(mu,sqrt(sigma2));
}'
```

Stan Model 2: multilevel normal

$$y_{i,p} \sim N\left(\mu_p, \sigma_p^2\right), \quad \mu_p \sim N\left(\mu, \sigma^2\right)$$

```
StanModel = '
data {
  int<lower=0> N; // Number of observations
  int<lower=0> y[N]; // Number of flowers
  int<lower=0> P; // Number of plants
}
transformed data {
  int<lower=0> M; // Number of months
  M = N / P;
}
parameters {
  real mu;
  real<lower=0> sigma2;
  real mup[P];
  real sigmap2[P];
}
model {
  mu ~ normal(0,100); // Normal with mean 0, st.dev. 100
  sigma2 ~ scaled_inv_chi_square(1,2); // Scaled-inv-chi2 with nu 1, sigma 2
  for(p in 1:P){
    mup[p] ~ normal(mu,sqrt(sigma2));
    for(m in 1:M)
      y[M*(p-1)+m] ~ normal(mup[p],sqrt(sigmap2[p]));
  }
}'
```

Stan Model 3: multilevel Poisson

$$y_{i,p} \sim Poisson\left(\mu_p\right), \quad \mu_p \sim logN\left(\mu, \sigma^2\right)$$

```
StanModel = '
data {
  int<lower=0> N; // Number of observations
  int<lower=0> y[N]; // Number of flowers
  int<lower=0> P; // Number of plants
}
transformed data {
  int<lower=0> M; // Number of months
  M = N / P;
}
parameters {
  real mu;
  real<lower=0> sigma2;
  real mup[P];
}
model {
  mu ~ normal(0,100); // Normal with mean 0, st.dev. 100
  sigma2 ~ scaled_inv_chi_square(1,2); // Scaled-inv-chi2 with nu 1, sigma 2
  for(p in 1:P){
    mup[p] ~ lognormal(mu,sqrt(sigma2)); // Log-normal
    for(m in 1:M)
      y[M*(p-1)+m] ~ poisson(mup[p]); // Poisson
  }
}'
```

Stan: fit model and analyze output

```r
data = list(N=N, y=y, P=P)
burnin = 1000
niter = 2000
fit = stan(model_code=StanModel,data=data,
           warmup=burnin,iter=niter,chains=4)

# Print the fitted model
print(fit,digits_summary=3)

# Extract posterior samples
postDraws <- extract(fit)

# Do traceplots of the first chain
par(mfrow = c(1,1))
plot(postDraws$mu[1:(niter-burnin)],type="l",ylab="mu",main="Traceplot")

# Do automatic traceplots of all chains
traceplot(fit)

# Bivariate posterior plots
pairs(fit)
```

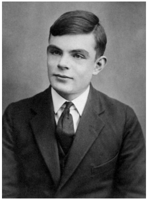
Stan - useful links

- Getting started with RStan
- RStan vignette
- Stan Modeling Language User's Guide and Reference Manual
- Stan Case Studies

Turing.jl

- Turing is a probabilistic programming language in Julia.
- Similar to Stan, but takes advantage of Julia's features such a metaprogramming.
- Named after Alan Turing (1912-1954).
- Written in Julia, which is fast natively.

Turing.jl logo



Alan Turing

Turing.jl for Bernoulli model

```
using Turing, StatsPlots, Random
# Declare the Turing model:
@model function iidbern(y, α, β)
    θ ~ Beta(α,β)  # prior
    N = length(y)  # number of observations
    for n in 1:N
        y[n] ~ Bernoulli(θ) # model
    end
end

# Set up the observed data
data = [0,1,1,0,0,1,1,0,1,1]

# Settings for the Hamiltonian Monte Carlo (HMC) sampler.
niter = 10000
nburn = 1000
ϵ = 0.1
τ = 10

# Sample the posterior using HMC
postdraws = sample(iidbern(data, 1, 2), HMC(ϵ, τ), niter,
    discard_initial = nburn) plot(postdraws)

# Print and plot results display(postdraws)
plot(postdraws)
```

Turing.jl for normal model

```
using Turing, StatsPlots, Random
ScaledInverseChiSq(ν,τ²) = InverseGamma(ν/2,ν*τ²/2) # Inv-χ² distribution

# Setting up the Turing model:
@model function iidnormal(x, μ₀, κ₀, ν₀, σ²₀)
    σ² ~ ScaledInverseChiSq(ν₀, σ²₀)
    θ ~ Normal(μ₀,σ²/κ₀)  # prior
    n = length(x)  # number of observations
    for i in 1:n
        x[i] ~ Normal(θ, √σ²) # model
    end
end

# Set up the observed data
x = [15.77,20.5,8.26,14.37,21.09]

# Set up the prior
μ₀ = 20; κ₀ = 1; ν₀ = 5; σ²₀ = 5^2

# Settings of the Hamiltonian Monte Carlo (HMC) sampler.
niter = 10000
nburn = 1000
α = 0.65 # target acceptance probability in No U-Turn sampler

# Sample the posterior using HMC
postdraws = sample(iidnormal(x, μ₀, κ₀, ν₀, σ²₀), NUTS(α), niter, discard_initial = nburn)

# Print an plot results
display(postdraws)
plot(postdraws)
```

Stan for normal model

```
library(rstan)
# Define the Stan model
stanModelNormal = '
// The input data is a vector y of length N.
data {
  // data
  int<lower=0> N;
  vector[N] y;
  // prior
  real mu0;
  real<lower=0> kappa0;
  real<lower=0> nu0;
  real<lower=0> sigma20;
}

// The parameters in the model
parameters {
  real theta;
  real<lower=0> sigma2;
}

model {
  sigma2 ~ scaled_inv_chi_square(nu0, sqrt(sigma20));
  theta ~ normal(mu0,sqrt(sigma2/kappa0));
  y ~ normal(theta, sqrt(sigma2));
}
'

# Set up the observed data
data <- list(N = 5, y = c(15.77, 20.5, 8.26, 14.37, 21.09))

# Set up the prior
prior <- list(mu0 = 20, kappa0 = 1, nu0 = 5, sigma20 = 5^2)

# Sample from posterior using HMC
fit <- stan(model_code = stanModelNormal, data = c(data,prior), iter = 10000 )

# print and plot results
print(fit, pars = c("theta","sigma2"), probs=c(.1,.5,.9))
pairs(fit)
traceplot(fit, pars = c("theta", "sigma2"), nrow = 2)
```