An overview of (*some*) Bayesian computational frameworks for teaching

ISBA 2022, Montreal

bit.ly/rundel_isba2022

Colin Rundel

Duke University

Personal Context

What I teach:

- Sta 323 - Statistical Computing (`R`)
- Sta 523 - Programming for Statistical Science (`R`)
- STA 663 - Statistical Computing and Computation (`Python`)
- STA 344/444/644 - Spatio-temporal modeling (`R`)

Tools I use:

- The majority of my time is spent with R, with a bit of C++
- I use JAGS and Stan for applied modeling
- Recently, more teaching focused on the Python ecosystem

Bayesian computational frameworks?

A collection of tools that implement a domain specific language for expressing and implementing Bayesian statistical models.

For example,

- JAGS
- STAN
- pymc
- + many others

Some teaching considerations

- Ease of use (installation, syntax, debugging, etc.)
- Blackboxiness / High vs low level
- Generalizability
- Performance / Limitations
- Wider curriculum

Installation + basic usage

- All of the frameworks have external / system dependencies
    - e.g. libjags, Eigen, theano, etc.
- Generally easy to install binary packages are available
    - source installs can be challenging
- If things break it tends to be spectacular and difficult to troubleshoot
    - OS makes a difference
    - 🔥Burn it down🔥 as a path forward

Example - Bayesian SLR

```r
d = read.csv("data/lm.csv")
plot(d)
```

$$y_i | m, b, \sigma \sim N(m \cdot x_i + b, \sigma)$$

$$\begin{aligned} m &\sim N(0, 10) \\ n &\sim N(0, 10) \\ \sigma &\sim N(0, 5) \end{aligned}$$

SLR - JAGS

```r
model = "
model{
  m ~ dnorm(0, 1/100)
  b ~ dnorm(0, 1/100)

  sigma ~ dnorm(0, 1/25) T(0,)

  for(i in 1:length(y)) {
    mu[i] = m*x[i] + b
    y[i] ~ dnorm(mu[i], 1/(sigma^2))
  }
}
"
```

```r
jags_model = rjags::jags.model(
  textConnection(model), data = d, n.chains=4
)
```

```
## Compiling model graph
##    Resolving undeclared variables
##    Allocating nodes
## Graph information:
##    Observed stochastic nodes: 11
##    Unobserved stochastic nodes: 3
##    Total graph size: 56
##
## Initializing model
```

```r
update(jags_model, n.iter=1000, progress.bar="none")

post_jags = rjags::coda.samples(
  jags_model, variable.names=c("m","b"),
  n.iter=1000, progress.bar="none"
)
```

SLR - Stan

```
stan = "
data {
  int<lower=0> N;
  vector[N] x;
  vector[N] y;
}
parameters {
  real m;
  real b;
  real<lower=0> sigma;
}
transformed parameters {
  vector[N] mu = m*x + b;
}
model {
  m ~ normal(0, 10);
  b ~ normal(0, 10);
  sigma ~ normal(0, 5);
  y ~ normal(mu, sigma);
}
"
```

```
post_stan = rstan::stan(
  model_code = stan,
  data = list(x=d$x, y=d$y, N=nrow(d)),
  pars = c("m", "b", "sigma"),
  chains = 4, warmup = 1000, iter = 2000,
  refresh = 1000, verbose = FALSE,
)
```

```
##
## SAMPLING FOR MODEL 'anon_model' NOW (CHAIN 1).
## Chain 1:
## Chain 1: Gradient evaluation took 1.6e-05 seconds
## Chain 1: 1000 transitions using 10 leapfrog steps per transi
## Chain 1: Adjust your expectations accordingly!
## Chain 1:
## Chain 1:
## Chain 1: Iteration:    1 / 2000 [  0%]  (Warmup)
## Chain 1: Iteration: 1000 / 2000 [ 50%]  (Warmup)
## Chain 1: Iteration: 1001 / 2000 [ 50%]  (Sampling)
## Chain 1: Iteration: 2000 / 2000 [100%]  (Sampling)
## Chain 1:
## Chain 1:  Elapsed Time: 0.012 seconds (Warm-up)
## Chain 1:                0.012 seconds (Sampling)
## Chain 1:                0.024 seconds (Total)
## Chain 1:
##
## SAMPLING FOR MODEL 'anon_model' NOW (CHAIN 2).
## Chain 2:
## Chain 2: Gradient evaluation took 1e-06 seconds
## Chain 2: 1000 transitions using 10 leapfrog steps per transi
## Chain 2: Adjust your expectations accordingly!
```

SLR - pymc3

```python
import pymc3 as pm
import arviz as az
```

```python
with pm.Model() as lm:
  m = pm.Normal('m', mu=0, sd=10)
  b = pm.Normal('b', mu=0, sd=10)

  mu = m * d.x + b
  sigma = pm.HalfNormal('sigma', sd=5)

  y = pm.Normal('y', mu=mu, sd=sigma, observed=d.y)
```

```python
with lm:
  post_pymc = pm.sample(return_inferencedata=True, random_seed=
```

```
## █
## Auto-assigning NUTS sampler...
## Initializing NUTS using jitter+adapt_diag...
## Multiprocess sampling (4 chains in 4 jobs)
## NUTS: [sigma, b, m]
## Sampling 4 chains for 1_000 tune and 1_000 draw iterations (
## There were 6 divergences after tuning. Increase `target_accep
```

Modelling results

- JAGS models return a coda `mcmc.list`
    - basic tabular structure that is easy to work with
- Stan models return a `stanfit` S4 object
    - less directly accessible but provides important basic summaries (e.g. `n_eff`, `Rhat`, etc.)
    - easily convertible to coda (`As.mcmc.list()`)
- pymc3 models return* ArviZ `InferenceData` objects (xarray/NetCDF based)
    - complex schema (everything and the kitchen sink approach)
    - less tabular friendly

- All frameworks support quick basic visualizations of results (trace, density, caterpillar, etc.)

Error reporting

- As pymc models are Python code, any syntax errors are reported as Python syntax errors
- JAGS and Stan implement their own parsers which have generally helpful error messages with the former tending to be terser / less detailed,

```
rjags::jags.model(
  textConnection("
  model{
    m ~ dnorm(0, 1/100
  }
  ")
)
```

```
## Error in rjags::jags.model(textConnection("\n  model{\n    m
## Error parsing model file:
## syntax error on line 4 near "}"
```

```
rstan::stan(
  model_code = "
  model {
    m ~ normal(0, 10;
  }
  "
)
```

```
## Error in stanc(file = file, model_code = model_code, model_na
## 0
##
## Syntax error in 'string', line 2, column 20 to column 21,
##   parsing error:
##
## Ill-formed phrase. Found an expression. This can be followed
##   by a ",", a "}", a ")", a "[", a "]" or an infix or postfi
##   operator.
```

- Runtime errors are a mixed bag

Posterior predictive checks

- Possible with all three frameworks, JAGS and Stan require that extra parameters be included in the model:

JAGS:

```
for(i in 1:length(y)) {
  y_tilde[i] ~ dnorm(mu[i], 1/(sigma^2))
}
```

Stan:

```
generated quantities {
  real y_tilde[N] = normal_rng(mu, sigma);
}
```

- pymc allows for the PPD to be sampled from an existing model result,

```
with lm:
    y_tilde = pm.sample_posterior_predictive(
        post_pymc, var_names=["y"], random_seed=1234
    )
```

- Similarly, the prior predictive samples can be generated without rewriting the model

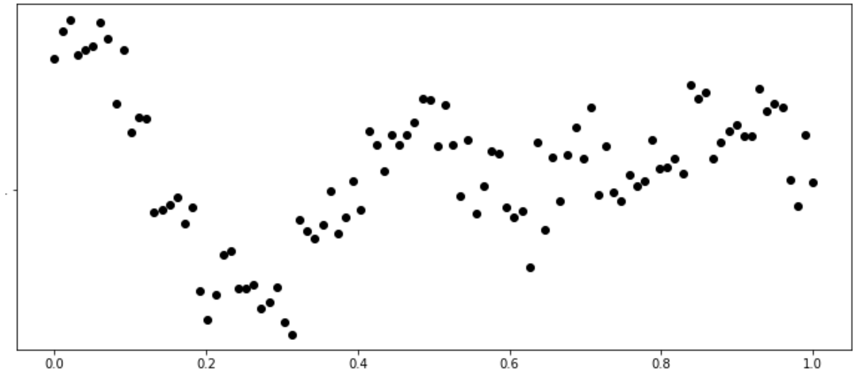
Limitations - GP Reg

```
d = pd.read_csv("data/gp.csv")
d.shape
```

```
## (100, 3)
```

```
D = np.array([ np.abs(xi - d.x) for xi in d.x])
I = (D == 0).astype("double")

fig = plt.figure(figsize=(12, 5))
plt.plot(d.x, d.y, "ok", ".")
plt.show()
```




```
with pm.Model() as gp:
  nugget = pm.HalfCauchy("nugget", beta=5)
  sigma2 = pm.HalfCauchy("sigma2", beta=5)
  ls     = pm.HalfCauchy("ls",     beta=5)

  Sigma = I * nugget + sigma2 * np.exp(-0.5 * D**2 * ls**2)

  y = pm.MvNormal(
    "y",
    mu=np.zeros(d.shape[0]),
    cov=Sigma, observed=d.y
  )
```

NUTS

```
with gp:
    post_nuts = pm.sample(
        return_inferencedata = True,
        chains = 2
    )
```

```
## █
## Multiprocess sampling (2 chains in 4 jobs)
## NUTS: [ls, sigma2, nugget]
## Sampling 2 chains for 1_000 tune and 1_000 draw iterations (2_000 + 2_000 draws total) took 240 seconds.
```

```
az.summary(post)
```

```
##            mean     sd  hdi_3%  hdi_97%  ...  mcse_sd  ess_bulk  ess_tail  r_hat
## nugget   0.541  0.087   0.397    0.715  ...    0.002    1754.0    1292.0    1.0
## sigma2   4.096  2.557   1.262    8.273  ...    0.060    1067.0    1004.0    1.0
## ls      10.756  2.383   6.593   15.267  ...    0.049    1068.0    1109.0    1.0
##
## [3 rows x 9 columns]
```

Slice steps

```python
with gp:
    step = pm.Slice([nugget, sigma2, ls])
    post_slice = pm.sample(
        return_inferencedata = True,
        chains = 2,
        step = step
    )
```

```
## █
## Multiprocess sampling (2 chains in 4 jobs)
## CompoundStep
## >Slice: [ls]
## >Slice: [sigma2]
## >Slice: [nugget]
## Sampling 2 chains for 1_000 tune and 1_000 draw iterations (2_000 + 2_000 draws total) took 24 seconds.
```

```python
az.summary(post_slice)
```

```
##            mean     sd  hdi_3%  hdi_97%  ...  mcse_sd  ess_bulk  ess_tail  r_hat
## nugget    0.542  0.085   0.399    0.705  ...    0.002    1573.0    1510.0    1.0
## sigma2    4.557  3.551   1.082   10.070  ...    0.087     915.0     842.0    1.0
## ls       10.526  2.466   5.815   14.552  ...    0.055     989.0     967.0    1.0
##
## [3 rows x 9 columns]
```

Metropolis-Hastings steps

```python
with gp:
    step = pm.Metropolis([nugget, sigma2, ls])
    post_mh = pm.sample(
        return_inferencedata = True,
        chains = 2,
        step = step
    )
```

```
## █
## Multiprocess sampling (2 chains in 4 jobs)
## CompoundStep
## >Metropolis: [ls]
## >Metropolis: [sigma2]
## >Metropolis: [nugget]
## Sampling 2 chains for 1_000 tune and 1_000 draw iterations (2_000 + 2_000 draws total) took 9 seconds.
## The estimated number of effective samples is smaller than 200 for some parameters.
```

```python
az.summary(post_mh)
```

```
##            mean     sd  hdi_3%  hdi_97%  ...  mcse_sd  ess_bulk  ess_tail  r_hat
## nugget    0.546  0.096   0.373    0.722  ...    0.004     321.0     351.0   1.01
## sigma2    4.535  3.522   1.081    9.282  ...    0.155     231.0     273.0   1.03
## ls       10.518  2.314   6.730   14.987  ...    0.118     188.0     220.0   1.03
##
## [3 rows x 9 columns]
```

Concluding thoughts

- All of these frameworks are a reasonable choice
  - Many different axes to optimize over
  - More excellent choices than ever before
- Feeling the grass is greener is real
- Personal choice for Fall 2022 (Sta 344)?
  - Probably BRMS -> Stan

Thank you!

Questions or Comments?

	rundel.github.io
	rundel
	rundel
	rundel@gmail.com colin.rundel@duke.edu
	bit.ly/rundel_isba2022