Introdução ao Stan como ferramenta de inferência bayesiana

Versão 1.4.3

Verifique a existência de uma versão mais recente em:
`https://marcoinacio.com/port`

Marco Henrique de Almeida Inácio *

*Universidade de São Paulo/UFSCar.

Conteúdo

1 Introdução

Esta apostila tem por objetivo ensinar por meio de exemplos práticos, o público alvo são pessoas com conhecimento básicos ou avançados de inferência bayesiana e `R` ou `Python`.

O autor tem a intenção de manter esta apostila em constante atualização e melhora e para isso é importante o *feedback* e as críticas construtivas por parte de seus leitores. Como a apostila está em constante melhora, é recomendável verificar periodicamente no site (`http://marcoinacio.com/stan`) a disponibilidade de uma nova versão. No mesmo site você encontra também o código-fonte em `R` e `Python` dos modelos utilizados aqui.

A apostila tenta ser modular a medida do possível, assim sendo, após concluir a seção de introdução e a seguinte com modelos básicos, o leitor pode partir de imediato para qualquer um dos demais assuntos da apostilas ou para o apêndice (em especial, para seção de perguntas frequentes).

Ressaltamos que grande parte dos modelos aqui apresentados são inspirados nos presentes no manual oficial do software [1], o qual recomendamos fortemente a leitura após o termino deste apostila.

1.1 Stan e seus algoritmos

`Stan` é um software cuja função primordial é amostrar um modelo estatístico bayesiano. Adicionalmente, o software também é capaz de realizar otimização (útil para máxima verossimilhança) e inferência variacional (`ADVI`).

Aqui focaremos apenas na primeira função. Nessa função, o software amostra[1] um modelo utilizando o algoritmo `NUTS`, (ou ainda utilizando `HMC`, caso o usuário explicitamente o solicite, vide apêndice A.1 e A.2), o qual não necessita de nenhuma configuração previa do usuário além de informar o modelo estatístico na linguagem `Stan`, como veremos a seguir. Veja mais detalhes no apêndice B de Almeida Inácio, Izbicki e Salasar [2].

1.2 RStan e PyStan

Para realizar a amostragem no `Stan`, o usuário descreve o modelo na linguagem probabilística própria do `Stan`, feito isso, o software traduz o modelo para C++, compila, executa, e devolve ao usuário uma amostra `MCMC`.

Todo o procedimento pode ser feito diretamente no `R` utilizando-se o pacote `RStan` disponível nos repositórios do `CRAN`. Também é possível utilizar o `Stan` diretamente no `Python`, utilizando o `PyStan`, disponível via `pip`. Além disso, também existem interfaces para outros linguagens como `Matlab` e `Stata`. Caso ainda não tenha feito, recomendados fortemente que instale o `RStan` ou o `PyStan`, ou ainda o pacote corresponde ao software de seu interesse.

1.3 A linguagem do Stan

Aqui temos uma rápida descrição da linguagem do `Stan`. Não se preocupe caso não entenda muito bem: com os exemplos que virão na próxima seção, isso ficará mais claro.

No `Stan`, um modelo é composto por blocos, sendo 3 os mais básicos:

[1] como estamos focando no primeira função apenas, amostrar um modelo e executar um modelo serão usados praticamente como sinônimos.

data onde são declarados os dados que serão enviados (pelo usuário) ao `Stan` sempre que se for executar o modelo.

parameters onde são declarados os parâmetros que serão amostrados e entregues (pelo `Stan`) aos usuário ao final da execução do modelo.

model onde o usuário descreve o modelo que deseja que o Stan amostre (i.e.: especifica a função de verossimilhança e as prioris. No `Stan`, como de praxe em inferência bayesiana, somente precisamos informar a posteriori a menos de uma constante de normalização.

Além disso, existem várias maneiras de armazenar a informação, tanto dos parâmetros, quanto dos dados, sendo as mais básicas:

int um unico número inteiro.

real um unico número real.

int[] uma "coleção" (array) de números inteiros.

real[] uma "coleção" (array) de números reais.

vector um vetor coluna de números reais.

row_vector um vetor linha de números reais.

matrix uma matriz de números reais.

1.4 Citando este material

O seguinte template bibtex pode ser usado para citar este material:

```
@misc{apostilaStan,
author={Marco Henrique de Almeida Inácio},
title={Introdução ao Stan como ferramenta de inferência bayesiana},
url={https://marcoinacio.com/stan}
}
```

Ou ainda o artigo relacionado:

```
@article{deAlmeidaIncio2018,
  doi = {10.1080/03610918.2018.1484480},
  url = {https://doi.org/10.1080/03610918.2018.1484480},
  year = {2018},
  month = nov,
  publisher = {Informa {UK} Limited},
  volume = {49},
  number = {1},
  pages = {261--282},
  author = {Marco Henrique de Almeida In{\'{a}}cio and
  Rafael Izbicki and Luis Ernesto Salasar},
  title = {Comparing two populations using Bayesian
  Fourier series density estimation},
  journal = {Communications in Statistics - Simulation
  and Computation}
}
```

2 Exemplos de modelos básicos

Um modelo no `Stan` é composto por blocos, sendo 3 os mais básicos:

2.1 Primeiro modelo: média de uma normal

Vejamos um modelo simples que amostra o parâmetro de média de uma normal:

```
data {
  int n;
  real y[n];
}
parameters {
  real mu;
}
model {
  for (i in 1:n) {
    y[i] ~ normal(mu, 1);
  }
}
```

A primeira coisa a ressaltar é que não é necessário o loop pois o `Stan` vetoriza automaticamente a array *y*.

Assim sendo, podemos escrever o bloco *model* como simplesmente:

```
model {
  y ~ normal(mu, 1);
}
```

Agora rode no `R` o arquivo *1_gaussian_mean.R*.

Ao "abrir" o objeto *fit* onde foi salvo o resultado de sua inferência, você deverá receber como resultado algo como:

```
> fit
Inference for Stan model: b47ef5f8d84d13d75baf0b47a5ed7514.
2 chains, each with iter=10000; warmup=5000; thin=1;
post-warmup draws per chain=5000, total post-warmup draws=10000.

        mean se_mean   sd     2.5%      25%      50%      75%    97.5% n_eff Rhat
mu      2.79    0.00 0.10     2.59     2.72     2.79     2.86     2.99  3393    1
lp__ -137.69    0.01 0.72 -139.74 -137.84 -137.42 -137.23 -137.18  4046    1

Samples were drawn using NUTS(diag_e) at Fri Mar 11 22:22:22 2016.
For each parameter, n_eff is a crude measure of effective sample size,
and Rhat is the potential scale reduction factor on split chains (at
convergence, Rhat=1).
```

Este é um resumo da posteriori que foi amostrada, vemos para cada parâmetro (neste modelo temos apenas o parâmetro μ) as informações de média, desvio padrão, erro padrão médio, alguns quantiles, o indicador de convergência *Rhat* (*Rhat* "próximo" de 1 é **condição necessária** para convergência do amostrador) e o pouco conhecido, mas muito útil *n_eff* que nos dá uma **estimativa** do número efetivo de amostras: o número de amostras equivalentes de um amostrador Monte Carlo (amostras i.i.d. da posteriori).

Podemos extrair as amostras do objeto fit utilizando as funções *as.matrix*, *as.array* ou *extract*.

Além disso, é exibido também um resumo para *lp__* que é a soma dos todos logaritmos das probabilidades do modelo no bloco *model* (ou seja, é o logaritmo da posteriori).

2.2 Segundo modelo: desvio padrão de uma normal

```
data {
  int n;
  real y[n];
}
parameters {
  real mu;
  real <lower=0> sigma;
}
model {
  y ~ normal(mu, sigma);
  sigma ~ cauchy(0, 2.5);
}
```

O que podemos aprender com o exemplo:

- Limite inferior no sigma (<lower=0>): sempre que o espaço paramétrico for restrito (i.e.: no caso de parâmetro unidimensionais, sempre que o espaço paramétrico **não** for igual a $\mathbb{R}$), você deve declarar isso explicitamente no `Stan`.
- A distribuição normal é parametrizada pela média e pelo desvio padrão (não é pela precisão nem pela variância).
- Cauchy como priori "padrão" ("pouco-informativa") recomendada. Para mais detalhes, veja [3].
- Não há qualquer necessidade de utilizar conjugadas no `Stan` (o fato de serem conjugadas em nada o auxiliam computacionalmente).
- A priori do parâmetro μ não é apresentada, portanto é proporcional a uma constante (i.e.: é a uniforme imprópria pois o espaço do paramétrico de μ é $\mathbb{R}$).

Agora rode no `R` o arquivo *2_gaussian_std_deviation.R*.

2.3 Terceiro modelo: uma regressão linear

```
data {
  int n;
  vector[n] x1;
  vector[n] x2;
  real y[n];
}
parameters {
  real beta0;
  real beta1;
  real beta2;
  real <lower=0> sigma;
}
model {
  y ~ normal(beta0 + beta1 * x1 + beta2 * x2, sigma);
  sigma ~ cauchy(0, 2.5);
}
```

Perceba que modelo de regressão linear temos as seguintes operações:

- Multiplicação de um vetor por um escalar, obtendo novamente um vetor de mesma dimensão, o qual é a multiplicação de cada componente do vetor e o escalar.
- A soma um escalar e um vetor, obtendo novamente um vetor de mesma dimensão, o qual a é a adição de cada componente do vetor e o escalar.
- A soma de vetores, obviamente obtendo novamente um vetor de mesma dimensão.

Outra coisa a se notar é que y é uma array, o parâmetro de media é um vetor e o do desvio padrão é um escalar real e que tudo funciona normalmente como esperado, de maneira equivalente (porém mais eficiente que):

```
for (i in 1:n)
  y[i] ~ normal((beta0 + beta1 * x1 + beta2 * x2)[i], sigma);
```

Agora rode no `R` o arquivo *3_linear_regression.R*.

Observação: a partir do modelo 3, a descrições dos modelos na linguagem Stan foram colocadas em arquivos separados (com extensão `.stan`). Portanto, para rodar um modelo, é necessário abrir o `R` (ou usar `setwd`) ou o `Python` na mesma pasta onde se encontra o arquivo `.stan` do modelo.

2.4 Quarto modelo: *yet another* regressão linear

```
data {
  int n;
  int m;
  matrix[n, m] x;
  real y[n];
}
parameters {
  vector[m] beta;
  real <lower=0> sigma;
}
model {
  y ~ normal(x * beta, sigma);
  sigma ~ cauchy(0, 2.5);
}
```

Aqui temos novamente uma regressão linear, porém neste caso com uma operação de multiplicação entre uma matriz de n linhas e m colunas e um vetor coluna de m linhas, o que nós dá como resultado, portanto, um vetor coluna de n linhas.

Agora rode no `R` o arquivo *4_linear_regression_matrix.R*.

2.5 Quinto modelo: regressão linear com previsão

```
data {
  int n;
  vector[n] x1;
  vector[n] y;
}
transformed data {
  real media_x1;
  media_x1 = mean(x1);
```

```
}
parameters {
  real beta0;
  real beta1;
  real <lower=0> sigma;
}
model {
  y ~ normal(beta0 + beta1 * x1, sigma);
  sigma ~ cauchy(0, 2.5);
}
generated quantities {
  real y_predito;
  y_predito = normal_rng(beta0 + beta1 * media_x1, sigma);
}
```

Vemos neste modelo dois outros blocos da linguagem do `Stan`:

transformed data neste podemos criar novos dados a partir de dados que foram informados ao `Stan` no bloco *data*.

generated quantities aqui podemos criar novos "parâmetros" com bases nos que o Stan já amostrou (os definidos no bloco *parameters*). Os parâmetros criados no bloco "generated quantities" são entregues ao usuário ao final da execução do modelo da mesma maneira que o parâmetros convencionais criados no bloco *parameters*.

Além fazemos aqui previsões a partir da posteriori para o valores médios do regressores de modelo linear simples.

Agora rode no `R` o arquivo *5_linear_regression_with_prediction.R*.

2.6 Sexto modelo: Bernoulli simples

Vejamos agora um modelo com dados dicotômicos:

```
data {
  int n;
  int y[n];
}
parameters {
  real <lower=0, upper=1> p;
}
model {
  y ~ bernoulli(p);
  p ~ beta(1, 1);
}
```

Aqui ressaltamos a importância do limite inferior e superior do parâmetro p!

Agora rode no `R` o arquivo *6_bernoulli.R*.

2.7 Sétimo modelo: regressão logistica

```
data {
  int n;
  int y[n];
```

```
  vector[n] x;
}
parameters {
  real beta0;
  real beta1;
}
transformed parameters {
  real <lower=0, upper=1> mu[n];
  for (i in 1:n)
    mu[i] = inv_logit(beta0 + beta1 * x[i]);
}
model {
  y ~ bernoulli(mu);
}
```

Aqui temos um exemplo de uso de um outro bloco do `Stan`: o *transformed parameters*: nele podemos criar novos parâmetros com base em outros já definidos anteriormente (e também com base em variáveis definidas nos blocos *data* e *transformed data*).

Para parâmetros declarados em *transformed parameters* não é necessário que sejam informados seus limites, embora sempre que possível é recomendável por ajudar a detectar erros de programação[2].

Um detalhe talvez pouco conhecido é que se algum dos parâmetros definido em *transformed parameters* não for utilizado em *model* (ou seja, você definiu ele não por necessidade do amostrador, mas apenas para tê-lo na saída do `Stan`), então você pode defini-lo no *generated quantities*[3].

Vale também ressaltar que se as suas prioris (no bloco *model*) estiverem definidas para um parâmetro definido em *transformed parameters*, então você está efetuando uma transformação de variáveis e deve portanto adicionar (em *model*) o logaritmo do valor absoluto da jacobiana desta transformação (veja [1] para mais detalhes).

Agora rode no `R` o arquivo *7_logistic_regression.R*.

2.8 Oitavo modelo: regressão logistica revisitada

```
data {
  int n;
  int y[n];
  vector[n] x;
}
parameters {
  real beta0;
  real beta1;
}
model {
  y ~ bernoulli_logit(beta0 + beta1 * x);
}
```

Aqui temos mais uma vez a regressão logística, mas neste caso, a distribuição *bernoulli_logit* trabalha diretamente na função de ligação logit, tornando desnecessário (e incorreto) usarmos a função *inv_logit* (logit inversa). Cabe ressaltar que esta maneira de amostrar a regressão logística não só é mais simples de escrever mas também é computacionalmente mais eficiente.

[2] O `Stan` retornará uma mensagem de erro sempre que manipulações dentro do bloco *transformed parameter* levarem um parametro lá definido para fora dos limites que o usuário imputou ao mesmo.

[3] E nesse caso, usar o bloco *generated quantities* é mais eficiente, veja pergunta corresponde na subseção A.3 do apêndice

Agora rode no R o arquivo *8_improved_logistic_regression.R*.

3 Sobrevivência

De acordo com [4], a análise de sobrevivência é, de maneira geral, o conjunto de procedimentos estatísticos para análise de dados para os quais a variável de interesse é o tempo transcorrido até que um determinado evento ocorra.

Nesta seção demonstramos a realização de alguns destes procedimentos no `Stan`.

3.1 Modelo simples

O caso mais simples (e não muito emocionante) é aquele em que observamos a sobrevivência de um indivíduo sem nenhuma censura ou truncamento:

$$P(\lambda|T_{obs}) = \frac{P(T_{obs}|\lambda)P(\lambda)}{P(T_{obs})} \tag{1}$$

Nos modelos aqui apresentados usaremos:

- $T_{obs}|\lambda \sim$ exponencial(λ)
- $\lambda \sim$ Cauchy$(0, 2.5)$

Onde $E(T_{obs}|\lambda) = 1/\lambda$. Assim, temos o seguinte modelo no `Stan`:

```
data {
  int n_obs;
  real t_obs[n_obs];
}
parameters {
  real<lower=0> lambda;
}
model {
  t_obs ~ exponential(lambda);
  lambda ~ cauchy(0, 2.5);
}
```

Agora rode no `R` o arquivo *survival_simple.R*

3.2 Modelo com truncamento

No caso de truncamento, sabemos que se o tempo de sobrevivência de algum indivíduo estiver fora de um determinado intervalo, ele não é observado, **e não sabemos quantos indivíduos não foram observados**.

Assim sendo, o que acontece nesse caso é que estamos recebendo efetivamente uma amostra de uma distribuição (exponencial) truncada. Para facilitar o entendimento, dividimos aqui o truncamento nos dois casos a seguir.

3.2.1 Pontos de truncamento conhecidos

Se os pontos do intervalo de truncamento são conhecidos, então basta adicionarmos os pontos de truncamento à verossimilhança:

```
data {
  int n_obs;
  real t_obs[n_obs];
  real<upper=min(t_obs)> L;
  real<lower=max(t_obs)> U;
}
parameters {
  real<lower=0> lambda;
}
model {
  for (i in 1:n_obs)
    t_obs[i] ~ exponential(lambda) T[L, U];
  lambda ~ cauchy(0, 2.5);
}
```

Repare que todavia tivemos que informar a verossimilhança ponto-a-ponto usando um loop. O motivo disso é que o `Stan` não aceita vetorização quando a distribuição é truncada.

3.2.2 Pontos de truncamento desconhecidos

Se os pontos de truncamento são desconhecidos, temos que incluir e tratar os pontos de truncamento como parâmetros e dar a eles uma priori, preferencialmente, bastante informativa.

Temos então o seguinte modelo no `Stan`:

```
data {
  int n_obs;
  real t_obs[n_obs];
}
transformed data {
  real min_t_obs;
  real max_t_obs;
  min_t_obs = min(t_obs);
  max_t_obs = max(t_obs);
}
parameters {
  real<lower=0> lambda;
  real<upper=min_t_obs> L;
  real<lower=max_t_obs> U;
}
model {
  for (i in 1:n_obs)
    t_obs[i] ~ exponential(lambda) T[L, U];
  L ~ normal(1/lambda - .3, 1);
  U ~ normal(1/lambda + .3, 1);
  lambda ~ cauchy(0, 2.5);
}
```

Agora rode no `R` o arquivo *survival_truncation.R*

3.3 Modelo com censura

No caso de censura à direita, temos que, para alguns indivíduos, o qual chamamos de censurados, sabemos **apenas** que seu tempo de sobrevivência foi superior a um determinado valor.

O caso de censura à esquerda é analogo, sabemos apenas que se o tempo de sobrevivência de algum indivíduos foi inferior a determinado valor.

3.3.1 Um único ponto de censura

No caso mais simples, a censura ocorre no mesmo ponto para todos os indivíduos censurados. Nesse caso, suponhamos que uma censura à direita ocorreu em um U (conhecido).

Suponhamos que tenhamos observado k indivíduos não censurados, e $n-k$ censurados, de modo que:

$$(t_1, t_2, ..., t_k) = t_{1:k} = t_{obs}$$
$$(t_{k+1}, t_{k+2}, ..., t_n) = t_{(k+1):n} = t_{cens}$$

E que:

- $T|\lambda \sim$ exponencial(λ)
- $\lambda \sim$ Cauchy$(0, 2.5)$

Assim teremos (se $x_i > U$):

$$\begin{aligned} & P(\lambda, T_{cens} = x | T_{obs} = t_{obs}, T_{cens} > U) \\ &= \frac{P(T_{obs} = t_{obs}, T_{cens} > U | \lambda, T_{cens} = x) P(T_{cens} = x|\lambda) P(\lambda)}{P(T_{obs} = t_{obs}, T_{cens} > U)} \\ &= \frac{P(T_{obs} = t_{obs} | \lambda, T_{cens} = x) P(T_{cens} = x|\lambda) P(\lambda)}{P(T_{obs} = t_{obs}, T_{cens} > U)} \\ &= \frac{P(T_{obs} = t_{obs} | \lambda) P(T_{cens} = x|\lambda) P(\lambda)}{P(T_{obs} = t_{obs}, T_{cens} > U)} \end{aligned} \tag{2}$$

Temos então o seguinte modelo no `Stan`:

```
data {
  int n_obs;
  int n_cen;
  real t_obs[n_obs];
  real<lower=max(t_obs)> u;
}
parameters {
  real<lower=0> lambda;
  vector<lower=u>[n_cen] t_cen;
}
model {
  t_obs ~ exponential(lambda);
  t_cen ~ exponential(lambda);
  lambda ~ cauchy(0, 2.5);
}
```

Agora rode no `R` o arquivo *survival_censoring.R*

3.3.2 Modelo com censura marginalizado

Há uma outra forma de modelarmos a censura que é "marginalizando fora"[4] o *y_cen*:

$$
\begin{aligned}
& P(\lambda|T_{obs} = t_{obs}, T_{cens} > U) \\
&= \int_{-\infty}^{\infty} P(\lambda, T_{cens} = x|T_{obs} = t_{obs}, T_{cens} > U)\mathrm{d}x \\
&= \int_{U}^{\infty} P(\lambda, T_{cens} = x|T_{obs} = t_{obs}, T_{cens} > U)\mathrm{d}x \\
&= \int_{U}^{\infty} \frac{P(T_{obs} = t_{obs}, T_{cens} > U|\lambda, T_{cens} = x)P(T_{cens} = x|\lambda)P(\lambda)}{P(T_{obs} = t_{obs}, T_{cens} > U)}\mathrm{d}x \\
&= \int_{U}^{\infty} \frac{P(T_{obs} = t_{obs}|\lambda, T_{cens} = x)P(T_{cens} = x|\lambda)P(\lambda)}{P(T_{obs} = t_{obs}, T_{cens} > U)}\mathrm{d}x \\
&= \int_{U}^{\infty} \frac{P(T_{obs} = t_{obs}|\lambda)P(T_{cens} = x|\lambda)P(\lambda)}{P(T_{obs} = t_{obs}, T_{cens} > U)}\mathrm{d}x \\
&= \frac{P(T_{obs} = t_{obs}|\lambda)P(\lambda)}{P(T_{obs} = t_{obs}, T_{cens} > U)} \int_{U}^{\infty} P(T_{cens} = x|\lambda)\mathrm{d}x
\end{aligned}
\tag{3}
$$

Isso é feito da seguinte maneira no `Stan`:

```
data {
  int n_obs;
  int n_cen;
  real t_obs[n_obs];
  real<lower=max(t_obs)> u;
}
parameters {
  real<lower=0> lambda;
}
model {
  t_obs ~ exponential(lambda);
  target += n_cen * exponential_lccdf(u | lambda);
  lambda ~ cauchy(0, 2.5);
}
```

Agora rode no `R` o arquivo *survival_censoring_marginalized.R*

3.3.3 Modelo com multiplas censuras

Não é difícil, entretanto, estendermos esse modelo marginalizado para o caso de indivíduos tendo múltiplos pontos de censura. No `Stan` teríamos:

```
data {
  int n_obs;
  int n_cen;
  real t_obs[n_obs];
  real t_cen[n_cen];
}
parameters {
  real<lower=0> lambda;
```

[4] Tradução livre de *marginalizing out*.

```
}
model {
  t_obs ~ exponential(lambda);
  target += exponential_lccdf(t_cen | lambda);
  lambda ~ cauchy(0, 2.5);
}
generated quantities {
  real t_obs_new;
  t_obs_new = exponential_rng(lambda);
}
```

Agora rode no `R` o arquivo *survival_censoring_multiple.R*

Note que, além de estendermos o modelo para múltiplos pontos de censura, incluímos, no bloco *generated quantities*, um previsor para uma nova observação, o qual chamamos de *t_obs_new*.

Bônus: o arquivo *survival_censoring_multiple_weilbull.R* contém um exemplo análogo ao anterior, porém utilizando a distribuição Weilbull (ao invés de exponencial).

3.3.4 Regressão com censuras

Poderíamos ainda estender esse modelo de modo a incluir covariáveis e realizar assim uma regressão no `Stan`:

```
data {
  int n_obs;
  int n_cen;
  real t_obs[n_obs];
  real x_obs[n_obs];
  real t_cen[n_cen];
  real x_cen[n_cen];
}
parameters {
  real beta0;
  real beta1;
}
transformed parameters {
  real lambda_obs[n_obs];
  real lambda_cen[n_cen];
  for (i in 1:n_obs)
    lambda_obs[i] = 1 / exp(beta0 + beta1 * x_obs[i]);
  for (i in 1:n_cen)
    lambda_cen[i] = 1 / exp(beta0 + beta1 * x_cen[i]);
}
model {
  t_obs ~ exponential(lambda_obs);
  target += exponential_lccdf(t_cen | lambda_cen);
}
```

Note que utilizamos uma função de ligação log na nossa regressão. Agora rode no `R` o arquivo *survival_censoring_regression.R*

4 Variáveis discretas

O `Stan` não suporta amostrar variáveis discretas nos modelos haja visto que as mesmas não possuem derivada e portanto não podem ser amostradas usando o `HMC` ou o `NUTS`.

Porém podemos marginalizá-las para fora do modelo: suponha que tenhamos um parâmetro discreto $H \in \mathbb{H}$ e uma variável contínua $\mu \in \Theta$. Podemos obter a verossimilhança (marginal) de μ ao ponderá-la por todos os possíveis valores de H:

$$P(Y|\mu) = \sum_{h \in \mathbb{H}} P(Y|\mu, H = h)P(H = h|\mu) \tag{4}$$

Para mais detalhes sobre o procedimento, veja o apêndice B de Almeida Inácio, Izbicki e Salasar [2].

4.1 Mistura de modelos

Suponhamos que queremos obter a posteriori seguinte do seguinte modelo (o qual é uma mistura de modelos):

$$\begin{array}{l}(Y|\mu, H = 1) \sim \text{Poisson}(\mu) \\ (Y|\mu, H = 2) \sim \text{binomial negativa}(\mu, 30)\end{array} \tag{5}$$

Note que aqui a binomial negativa é parametrizada pela média μ e um parâmetro de precisão ϕ conforme a função *neg_binomial_2* do `Stan`. Nessa parametrização, a distribuição binomial negativa é equivalente a de Poisson (com mesmo μ) no limite em que $\phi \to \infty$.

Além disso, suponhamos as seguintes distribuições a priori:

- $\mu \sim \text{normal}(2.1, 1)$
- $P(H = 1) = P(H = 2) = 0.5$

Assim sendo, geremos dados a partir do seguinte modelo verdadeiro: $Y \sim \text{binomial negativa}(2, 10)$.

Temos então o seguinte modelo no `Stan`:

```
data {
  int n;
  int y[n];
}
parameters {
  real<lower=0> mu;
}
transformed parameters {
  vector[2] log_pesos;
  log_pesos[1] = poisson_lpmf(y | mu);
  log_pesos[2] = neg_binomial_2_lpmf(y | mu, 30);
}
model {
  target += log_sum_exp(log_pesos);
  mu ~ normal(2.1, 1);
```

```
}
generated quantities {
  int<lower=1, upper=2> H;
  vector[2] probs;
  probs = softmax(log_pesos);
  H = categorical_rng(probs);
}
```

Agora rode no R o arquivo *model_averaging.R*

Temos a seguinte saída no R:

```
> fit
Inference for Stan model: 8d3279a72b399237b4454f2a7b0999d2.
2 chains, each with iter=10000; warmup=5000; thin=1;
post-warmup draws per chain=5000, total post-warmup draws=10000.
```

	mean	se_mean	sd	2.5%	25%	50%	75%	97.5%	n_eff
mu	2.16	0.00	0.16	1.85	2.05	2.16	2.27	2.49	3139
log_pesos[1]	−74.75	0.00	0.34	−75.74	−74.82	−74.61	−74.53	−74.51	4636
log_pesos[2]	−74.29	0.00	0.32	−75.20	−74.36	−74.16	−74.09	−74.06	4629
H	1.61	0.00	0.49	1.00	1.00	2.00	2.00	2.00	9873
probs[1]	0.39	0.00	0.01	0.37	0.39	0.39	0.39	0.39	4721
probs[2]	0.61	0.00	0.01	0.61	0.61	0.61	0.61	0.63	4721
lp__	−148.28	0.01	0.68	−150.23	−148.44	−148.01	−147.85	−147.81	4595

Note que para o modelo 2, que está mais "próximo" do modelo verdadeiro, foi atribuída a probabilidade a posteriori $P(H = 2|D) \approx 0,61$.

Obs.: este exemplo visa apenas fins didáticos para exemplificar a mistura (discreta) de modelos no Stan. Na realidade, para um caso deste tipo onde existe um contínuo de modelos binomiais negativos (um para cada valor de ϕ), um procedimento mais adequado seria atribuir uma priori (contínua) para ϕ e (assim ter uma mistura contínua de modelos). Além disso, de maneira geral, a comparação de modelos é bastante sensível a escolha da priori (vide [5] e [6]).

4.2 Misturas finitas

A técnica de marginalização de variáveis discretas também pode ser utilizada para modelar misturas finitas no Stan.

Suponha que tenhamos observações que sabemos que vieram de 3 normais com mesma variância, mas médias diferentes. Porém não sabemos quais são as médias nem qual observação veio de qual média.

Temos então a seguinte verossimilhança:

$$\begin{aligned}(Y_i|\mu,\sigma,H_i=1) &\sim \text{Normal}(\mu_1,\sigma)\\ (Y_i|\mu,\sigma,H_i=2) &\sim \text{Normal}(\mu_2,\sigma)\\ (Y_i|\mu,\sigma,H_i=3) &\sim \text{Normal}(\mu_3,\sigma)\end{aligned} \tag{6}$$

Veja que no caso anterior (de mistura de modelos), assumíamos que ou todas as observações pertenciam a um modelo ou todas pertenciam a outro. Aqui, pelo contrário, assumimos que algumas observações podem estar em um modelo e outras podem estar em outro[5].

Utilizemos as seguintes prioris:

[5]Sugestão: reflita sobre a importância dessa distinção e como ela leva a modelagens e inferências distintas

- $\mu_i \sim \text{normal}(0, 5)$
- $\sigma \sim \text{Cauchy}(0, 2.5)$
- $H|\lambda \sim \text{categorica}(\lambda)$
- $\lambda \sim \text{Dirichlet}(\frac{2}{3}, \frac{2}{3}, \frac{2}{3})$

Temos assim o seguinte modelo no `Stan`:

```
data {
  int<lower=1> h; // number of mixture components
  int<lower=1> n; // number of data points
  real y[n]; // observations
}
parameters {
  simplex[h] lambda; // mixing proportions
  real mu[h]; // locations of mixture components
  real<lower=0> sigma;
}
model {
  real ps[h];
  mu ~ normal(0, 5);
  lambda ~ dirichlet(rep_vector(2.0/h, h));

  for (i in 1:n) {
    for (j in 1:h)
      ps[j] = log(lambda[j]) + normal_lpdf(y[i] | mu[j], sigma);
    target += log_sum_exp(ps);
  }
  sigma ~ cauchy(0, 2.5);
}
generated quantities {
  real mu_ordered[h];
  mu_ordered = sort_asc(mu);
}
```

Agora rode no `R` o arquivo *finite_mixture.R*

5 Séries Temporais

5.1 Modelo ARMA

Idealmente apresentaríamos primeiro os modelos `AR` e `MA` isoladamente, mas por questão de brevidade tratemos diretamente do modelo `ARMA`:

$$\begin{aligned}(y_t|y_{t-1}) &\sim \text{normal}(\mu_t, \sigma^2) \\ \mu_t &= c + \phi\, y_{t-1} + \theta\,(y_{t-1} - \mu_{t-1})\end{aligned} \tag{7}$$

Temos então o seguinte modelo no `Stan`:

```
data {
  int<lower=1> t_;
  real y[t_];
}
parameters {
  real c;
  real<lower = -1, upper = 1> phi;
  real<lower = -1, upper = 1> theta;
  real<lower=0> sigma;
}
model {
  vector[t_] mu;
  real err;
  mu[1] = c + phi * c;

  err = y[1] - mu[1];
  for (t in 2:t_) {
    mu[t] = c + phi * y[t-1] + theta * err;
    err = y[t] - mu[t];
  }

  c ~ normal(0, 10);
  phi ~ normal(0, 2);
  theta ~ normal(0, 2);
  sigma ~ cauchy(0, 5);
  y ~ normal(mu, sigma);
}
```

Agora rode no `R` o arquivo *time_series_arma.R*

5.2 Modelo GARCH

Caso queiramos modelar a volatilidade da série podemos valer-nos do modelo `GARCH`:

$$\begin{aligned}(y_t|y_{t-1}) &\sim \text{normal}(\mu, \sigma_t^2) \\ \sigma_t^2 &= \alpha + \beta\, \sigma_{t-1}^2 + \gamma\,(y_t - \mu)^2\end{aligned} \tag{8}$$

Temos então o seguinte modelo no `Stan`:

```
data {
  int<lower=0> t_;
  real y[t_];
}
parameters {
  real mu;
  real<lower=0> sigma_t_eq_1;
  real<lower=0> garch_const;
  real<lower=0> arch0;
  real<lower=0> garch0;
}
model {
  real error2[t_];
  real sigma[t_];
  real sigma2[t_];

  sigma[1] = sigma_t_eq_1;
  sigma2[1] = pow(sigma_t_eq_1, 2);
  error2[1] = pow(y[1] - mu, 2);

  for (t in 2:t_) {
    error2[t] = pow(y[t] - mu, 2);
    sigma2[t] = garch_const + garch0 * sigma2[t-1] + arch0 * error2[t-1];
    sigma[t] = sqrt(sigma2[t]);
  }

  y ~ normal(mu, sigma);
  sigma_t_eq_1 ~ cauchy(0, 2.5);
}
```

Agora rode no R o arquivo *time_series_garch.R*

5.3 Modelo ARMA-GARCH

Combinando os dois modelos anteriores, temos o modelo ARMA-GARCH:

$$\begin{aligned}(y_t|y_{t-1}) &\sim \text{normal}(\mu_t, \sigma_t^2)\\ \mu_t &= c + \phi\, y_{t-1} + \theta\,(y_{t-1} - \mu_{t-1})\\ \sigma_t^2 &= \alpha + \beta\,\sigma_{t-1}^2 + \gamma\,(y_t - \mu_t)^2\end{aligned} \tag{9}$$

Com o seguinte modelo no Stan:

```
data {
  int<lower=0> t_;
  real y[t_];
}
parameters {
  real arma_const;
  real phi;
  real theta;
  real<lower=0> sigma_t_eq_1;
  real<lower=0> garch_const;
  real<lower=0> arch0;
```

```
  real<lower=0> garch0;
}
model {
  real mu[t_];
  real error;
  real error2;
  real sigma[t_];
  real sigma2[t_];

  mu[1] = arma_const + phi * arma_const / (1 - phi);
  error = y[1] - mu[1];
  error2 = pow(error, 2);
  sigma2[1] = pow(sigma_t_eq_1, 2);
  sigma[1] = sigma_t_eq_1;

  for (t in 2:t_) {
    mu[t] = arma_const + phi * y[t-1] + theta * error;
    sigma2[t] = garch_const + garch0 * sigma2[t-1] + arch0 * error2;
    error = y[t] - mu[t];
    error2 = pow(error, 2);
    sigma[t] = sqrt(sigma2[t]);
  }

  y ~ normal(mu, sigma);
  sigma_t_eq_1 ~ cauchy(0, 2.5);
}
```

Agora rode no R o arquivo *time_series_arma_garch.R*

A Apêndice

Aqui incluímos, dentre outras coisas, algumas seções que poderiam ter sido incluídas no texto principal, mas não o foram de modo a não desmotivar o leitor com excessiva teoria.

A.1 Algoritmo HMC

`HMC` (*Halmitonian Monte Carlo* ou *Hybrid Monte Carlo*) é um algoritmo que utiliza do sistema de equações hamiltonianas da Física.

Esse sistema de equações descreve aquele famoso problema do ensino médio onde uma bola é colocada em uma rampa sem atrito e conforme ela desce vai convertendo energia potencial gravitacional em energia cinética até chegar no ponto mais baixo da rampa onde passa a subir e converter energia cinética em energia potencial, e como não há atrito ou outras perdas no sistema, a soma da energia potencial com a energia cinética é constante.

O algoritmo funciona da seguinte maneira:

Tratamos o vetor de variáveis que queremos amostrar q como variáveis de posição na equação hamiltoniana e a energia potencial como sendo proporcional a $-\log P(q)$ (o oposto da log-probabilidade, que no caso é o oposto da log-posteriori).

Criamos um vetor de variáveis adicionais p (no caso mais simples, independentes e com distribuição normal padrão), de mesma dimensão e independentes de q, e as tratamos como variáveis de momento no sistema de equações hamiltoniano. Da mesma forma a energia cinética é $-\log P(p)$.

O algoritmo funciona em dois estágios. No primeiro, amostramos p de sua distribuição (normal multivariada) e mantemos q inalterado. No segundo estagio levamos (p, q) de t_0 para t_s segundo o sistema de equações hamiltonianas, o segundo estagio é determinístico.

Esse sistema de equações tem propriedades interessantes como reversibilidade e preservação da soma das energia (logo $P(p, q)$ é mantido constante).

Um dos problemas do `HMC` é que é necessário que o usuário configure o dois parâmetros do amostrador: o número de saltos *leapfrog* e o tamanho de cada salto. Essa configuração é uma tarefa difícil de ser efetuada e requer conhecimento especializado.

A.2 Algoritmo NUTS

O algoritmo `NUTS` (*No-U-Turn Sampler*) é uma extensão do algoritmo `HMC` e recebe esse nome pois visa parar no momento em que a trajetória começa a refazer os seus passos.

No algoritmo, cuja descrição pode ser encontrada em descrito em [7], ao contrário do `HMC`, o usuário não necessita configurar a quantidade de saltos *leapfrog* e adicionalmente os autores do mesmo propuseram métodos para escolher o tamanho do salto *leapfrog* automaticamente. Isso foi incorporado ao `Stan` de modo que o usuário não necessita configurar nenhum parâmetro do amostrador para efetuar sua inferência. *It just works out of the box!*. Veja mais detalhes no apêndice B de Almeida Inácio, Izbicki e Salasar [2].

A.3 Perguntas frequentes

Perguntas frequentes[6] sobre esta apostila e o `Stan`:

1. Para que serve o *int* se eu posso declarar inteiros como *real*? Do ponto de vista de interface do usuário o real não é aceito em todos os lugares que o int é. Por exemplo, no número de observações da Poisson.

2. Qual a diferença entre *vector* e *real[]* e entre *matrix* e *real[,]*? Para o usuário a grande diferença é que vetores e matrizes permitem operações de álgebra linear como multiplicação de matrizes, enquanto que uma array de números reais (*real[]*) não o permite. Entretanto, as arrays são objetos mais gerais, podendo se estender a mais de 2 dimensões, e podendo existir até mesmo arrays de vetores ou de matrizes!

   Internamente, *vector*, *row_vector* e *matrix* são objetos de uma biblioteca (em `C++`) de álgebra linear chamada `Eigen`, enquanto arrays são objetos `std::vector`.

   Além disso, uma diferença que vale ressaltar é que as arrays de duas ou mais dimensões são armazenas em ordem "row-major", enquanto matrizes são armazenas em ordem "column-major".

3. Qual a diferença entre o bloco *transformed parameters* e o bloco *generated quantities*?

   Do ponto de vista de interface do usuário existem duas diferenças: a primeira é que o que é definido no bloco *transformed parameters* pode ser utilizado no bloco *model*, já o que é definido no bloco *generated quantities* não pode. A segunda diferença é que no bloco *generated quantities* e apenas nele, podemos utilizar as funções geradoras de números aleatórios do Stan (aquelas que terminam com *_rng* como por exemplo, a *normal_rng*).

   Do ponto de vista interno do `Stan`, entretanto, há vantagem em definir uma variável no bloco *generated quantities* ao invés do bloco *transformed parameters* quando isso é possível (ou seja, quando você não vai precisar dessa variável no bloco *model*): o bloco *generated quantities* não cria internamente objetos que fazem diferenciação automática (i.e.: calculam gradiente necessário para o algoritmo `NUTS` e `HMC`), pois não é mais necessário já que os parâmetros já foram amostrados, ao inves disso, o Stan trabalha ali com tipos mais primitivos do C++ (tais como *double* e *std::vector<double>*) que são computacionalmente mais rápidos.

4. Quão pequeno deve ser o *Rhat* para que eu tenha certeza que o modelo convergiu?

   Nenhum valor de *Rhat* é suficiente para se saber que o modelo convergiu, a relação de causalidade é contrária: a convergência da modelo implica *Rhat* igual a 1 (no limite). Vide [8] para mais detalhes.

   Vale lembrar ainda que o *n_eff* é também uma estimativa e que portanto está passível de erro, inclusive assim como *Rhat*, ele não necessariamente é capaz de saber se modelo convergiu corretamente.

   Vejamos um exemplo mental de *Rhat* e *n_eff* enganadores: pode ter acontecido de o amostrador ter ficado "travado" num submodelo de tal modo que aparente convergência (pois "passeou" bastante dentro do espaço amostral do submodelo), mas a convergência não ocorreu para o modelo "global".

   Algo que pode ser pode ser feito para dar um pouco mais de segurança na convergência é amostrar em várias cadeias cada uma com inicialização em pontos distintos do espaço amostral. Infelizmente esse tarefa nem sempre é fácil pois certos pontos do espaço amostral podem ser problemáticos para a inicialização devido a baixa probabilidade nesses pontos e geometrias complexas da posteriori.

5. O `Stan` amostra mais devagar que o software X...

   Se o software X funciona bem para sua necessidade, então tudo bem, porém deve-se ressaltar uma coisa: o que importa não é o número de amostras total, mas o número de amostras

[6] Não que o autor já tenha recebido dúzias de perguntas, porém aqui tenta antecipar algumas com base na intuição e experiência.

efetivo, um amostrador "rápido" porém que produz amostras extremamente correlacionadas pode ser pior que um "lento" mas que se aproxima da eficiência de um Monte Carlo.

A.4 Recursos adicionais

- Manual de referência do software [1].
- Artigo Almeida Inácio, Izbicki e Salasar [2].
- Site do `Stan`: `http://mc-stan.org/`.
- Fórum de ajuda: `discourse.mc-stan.org`.
- *Mailing list*: `https://goo.gl/MY8TDt`.
- Github do `Stan`: `https://github.com/stan-dev`.
- Livro *Bayesian data analysis* [9].
- Website do autor desta apostila: `https://marcoinacio.com`.
- Tutorial de Python voltado para estatisticos, com seção sobre PyStan e machine learning: `https://pytutorial.marcoinacio.com/`.

Referências

[1] Stan Development Team, *Stan Modeling Language Users Guide and Reference Manual, Version 2.14.0*, 2016.

[2] M. H. de Almeida Inácio, R. Izbicki e L. E. Salasar, "Comparing two populations using Bayesian Fourier series density estimation", *Communications in Statistics - Simulation and Computation*, vol. 49, n.º 1, pp. 261–282, nov. de 2018. DOI: 10.1080/03610918.2018.1484480.

[3] A. Gelman, A. Jakulin, M. G. Pittau e Y.-S. Su, "A weakly informative default prior distribution for logistic and other regression models", *Ann. Appl. Stat.*, vol. 2, n.º 4, pp. 1360–1383, dez. de 2008. DOI: 10.1214/08-AOAS191.

[4] D. G. Kleinbaum, *Survival Analysis (Statistics for Biology and Health)*. Springer, 2013, ISBN: 978-1441966452.

[5] J. Kruschke, *Doing Bayesian data analysis: a tutorial with R, JAGS, and Stan*. Boston: Academic Press, 2015, ISBN: 978-0124058880.

[6] R. E. Kass e A. E. Raftery, "Bayes Factors", *Journal of the American Statistical Association*, vol. 90, n.º 430, pp. 773–795, 1995.

[7] M. D. Hoffman e A. Gelman, "The No-U-Turn Sampler: Adaptively Setting Path Lengths in Hamiltonian Monte Carlo", *Journal of Machine Learning Research*, vol. 15, pp. 1593–1623, 2014.

[8] *Rhat for lp___ - Google Groups*, goo.gl/Il9bJd, Acessado: 2017-03-24.

[9] A. Gelman, J. B. Carlin, H. S. Stern, D. B. Dunson, A. Vehtari e D. B. Rubin, *Bayesian data analysis*, Third. CRC Press, 2014, ISBN: 978-143984095-5.