Bayesian Time Series with Stan

Alin Morariu

08/05/2020

Abstract

While time series analysis has typically been performed via the frequentist approach to statistics, Bayesian methods are now rising in popularity. Increases in computing power and more interpretable results have made fitting class statistical models with a Bayesian approach more attractive. In this document, we adapt 3 standard time series models to fit the Bayesian framework and perform parameter estimates with the help of R-Stan.

Contents

A Note to the Reader

This document is **not** a time series tutorial. The focus is not performing an in-depth and appropriate time series analysis but simply using Markov Chain Monte Carlo (MCMC) methods for the purpose of *parameter estimation*. As such, all things considering modeling checking and forecasting are omitted. All material below is to be viewed through a purely computational lens which shows how models are coded/structured in Stan and understanding the output.

Bayesian Framework

Unlike the frequentist approach where point estimates are computed for parameter values, the Bayesian framework allows us to specify a distribution for the parameter value *given* the data. This distribution is referred to as the posterior distribution of the parameter and is a function of the data.

$$\mathbb{P}(\theta|\text{data}) \propto \mathcal{L}(\text{data}|\theta) \cdot \mathbb{P}(\theta) \tag{1}$$

The quantities above will be referred to as:

- $\mathbb{P}(\theta|\text{data})$ - posterior distribution (function of the parameter)
- $\mathcal{L}(\text{data}|\theta)$ - likelihood (function of the data)
- $\mathbb{P}(\theta)$ - prior distribution (function of the parameter)

Parameter estimates correspond to the mode of the posterior distribution and this distribution is precisely what Stan explores with it's various MCMC algorithms.

Stan and it's Samplers

Stan is a language used for (amongst other things) full Bayesian statistical inference with MCMC sampling. It implements two MCMC methods; Hamiltonian Monte Carlo (HMC) and No U-Turn Sampling (NUTS). This section provides a high level summary of both algorithms. The key thing to keep in mind is that in order to have a "good" estimate of the model parameters we need to fully explore the posterior density of the parameters. These algorithms add certain contraints to the basic randomly generated steps of a vanilla Metropolis-Hastings (this is what we mean by MCMC algorithm) which makes them more efficient in exploring "new" parts of the space.

The set up of any MCMC method is that we want to have an estimate for a parameter vector θ. Based on our data, we compute the unnormalized posterior distribution, $\mathbb{P}(\theta|X)$, of the parameter vector. Now, we must sample from this distribution. MCMC work by creating a Markov chain which has the same distribution as posterior. Since this distribution will not always be known we take a second distribution $f(x)$ such taht $f(\theta) \propto k\mathbb{P}(\theta|X)$ where k is some constant. Now we initialize the algorithm at some point θ_0 then sample a potential next step θ' from some proposed density $g(\theta'|\theta_t)$ (note since we are taking samples of θ over time we create a Markov chain). We typically take g to be a Gaussian distribution centered at x_t as to keep the proposed steps relatively close to our intial point. Once we have a proposal, we verify that this is a "good" step. To do so, we compute $\alpha = \frac{f(\theta')}{f(\theta_t)} = \frac{\mathbb{P}(\theta'|X)}{\mathbb{P}(\theta_t|X)}$ and accept only if $\alpha \geq u \sim U(0,1)$. We will always accept if the next step is more probable (i.e. in the right direction) since that ratio will be above 1. Manipulating this error is precisely what HMC and NUTS attempt to do by exploiting some additional information about the posterior (namely gradients).

Hamiltonian Monte Carlo

HMC differs from the standard version of Metropolis-Hastings by introducing an additional variable referred to as the *auxiliary momentum*. We will denote this with ρ. Much like with everything else in the Bayesian world this will be specified by a probability distribution.

$$\mathbb{P}(\rho, \theta) = \mathbb{P}(\rho|\theta) \cdot \mathbb{P}(\theta) \tag{2}$$

$$\rho \sim MVN(0, \text{diag}(\Sigma_\theta)^{-1}) \tag{3}$$

The covariance matrix Σ_θ is computed during the warm up stage of the algorithm (think of this as a stage in the sampling where we go from no knowledge of the posterior to some knowledge of the posterior).

This auxiliary momentum is then used to compute the Hamiltonian of the posterior. The Hamiltonian is a matrix describing the level sets of the distribution in high dimensional space. Much like planets orbit in a set eliptical path, each level set of a distribution is a possible elipse for an object to move along and the collection of these level sets form a dynamical system.

$$H(\rho, \theta) = -\log \mathbb{P}(\rho, \theta) \tag{4}$$

$$= -\log \mathbb{P}(\rho|\theta) - \log \mathbb{P}(\theta) \tag{5}$$

where we denote the **kinetic energy** of the system as $T(\rho|\theta) = \log \mathbb{P}(\rho|\theta)$ and the **potential energy** as $V(\theta) = \log \mathbb{P}(\theta)$. The proposal θ' now turns into a 2 stage process.

1. Sample from $\rho \sim MVN(0, \text{diag}(\Sigma_\theta)^{-1})$
2. Solve the partial differential equation system

$$\begin{cases} \frac{d\theta}{dt} = +\frac{d \log \mathbb{P}(\rho|\theta)}{d\rho} \\ \frac{d\rho}{dt} = -\frac{d \log \mathbb{P}(\rho|\theta)}{d\theta} - \frac{d \log \mathbb{P}(\theta)}{d\theta} \end{cases}$$

$$= \begin{cases} \frac{d\theta}{dt} = +\frac{dT(\rho|\theta)}{d\rho} \\ \frac{d\rho}{dt} = -0 - \frac{dV(\theta)}{d\theta} \end{cases}$$

$$= \begin{cases} \frac{d\theta}{dt} = +\frac{dT}{d\rho} \\ \frac{d\rho}{dt} = -\frac{dV}{d\theta} \end{cases}$$

   with a leapfrog integrator

The estimate is then computed by drawing a new sample of ρ, taking a $\frac{1}{2}$ step update in the momentum, a full step in the parameter value and another $\frac{1}{2}$ step in the momentum before computing α.

$$\rho^* \leftarrow \rho - \frac{\epsilon}{2}\frac{dV}{d\theta} \tag{6}$$

$$\theta^* \leftarrow \theta + \epsilon \text{diag}(\Sigma_\theta)^{-1}\rho* \tag{7}$$

$$\rho^* \leftarrow \rho - \frac{\epsilon}{2}\frac{dV}{d\theta} \tag{8}$$

$$\Rightarrow (\rho^*, \theta^*) \tag{9}$$

Where the pair (ρ^*, θ^*) is the new proposal. Now we apply the Metroplois step and get an acceptance probability of $p_{\text{accept}} = \min 1, \exp\{H(p, \theta) - H(p^*, \theta^*)\}$

No U-Turn Sampler

One downfall of the HMC algorithm is that you may end up with a situation where you are going around in circles around the posterior and returning to where you started. This creates some inefficiencies that the

NUTS aims fix this by taking steps in both the positive and negative directions as dictated by the differential system. It continues to take steps until the gradients change direction. Once those points are detected, an average is choosen as the proposal.

This algorithm is much more complex and is thus omitted from this discussion. If you'd like more details, this paper by Michael Betancourt will provide more details.

Stan Code Structure

The way we use Stan is by encoding a model in Stan code (a .stan file) then running it through R. Models are laid out in an almost heirarchical structure. First is a data chunk listing all of the variables which will be inputted. Next is a parameters section which we use for specifying the quantities we want to estimate. Lastly, we need a section to specify the structure of the model, namely the priors and likelihood functions.

Necessary Packages

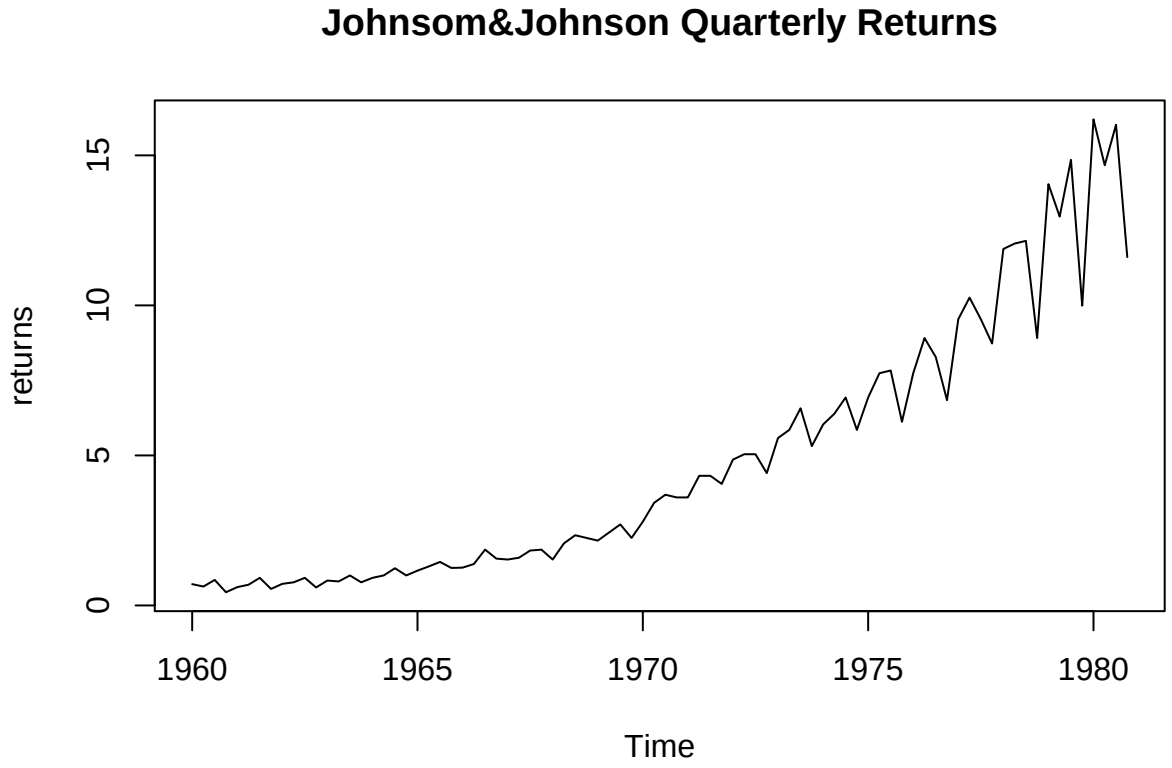
This section is simply for loading the base R packages we will be using to perform our Bayesian Inference.

```
### PACKAGES ####
library(tidyverse)            # data processing
library(rstan)                # R version of Stan
library(parallel)             # enable parallel computing

### OPTIONS ####
# Choose one of the below
options(mc.cores = 1)                              # all chains computed on 1 core
#options(mc.cores = parallel::detectCores())       # chains run in parallel

rstan_options(auto_write = TRUE)
```

Time Series Models

Now that we have an understanding of how Stan computes the estimates, let's write some Stan code. *I once again would like to emphasize that this section is simply fitting the various models to the data regardless of how poorly they may perform!* All of the models are assumed to have Gaussian noise components and will be fitted based on the JohnsonJohnson data set.

```
returns <- JohnsonJohnson

plot(returns,
     type = 'l',
     main ='Johnsom&Johnson Quarterly Returns')
```



Auto-regressive Model

Auto-regressive models of order p, denoted AR(p), take the form:

$$y_t = \phi_1 y_{t-1} + \ldots + \phi_p y_{t-p} + \epsilon_t \tag{10}$$

$$\epsilon_t \sim N(0, \sigma^2) \tag{11}$$

The parameters we need to estimate are $(\phi_1, \ldots, \phi_p, \sigma)$. We begin with a simple AR(1) model before generalizing the Stan code to allow for the user to input the order of the model as part of the data.

AR(1)

The Stan code is as follows:

```
data {
int<lower=0> N;              // length of TS
vector[N] returns;           // name of the vector/column we feed in
}
parameters {
real phi_0;                 // estimate the mean
real phi_1;                 // estimate the first order parameter
real<lower=0> sigma;         // estimate the std dev of the Gaussian noise
}
model {
// priors
phi_0 ~ normal(0,4);
phi_1 ~ normal(0,2);
```

```
returns[2:N] ~ normal(phi_0 + phi_1 * returns[1:(N-1)], sigma);
// note: improper/default priors are used here. We specify priors later
}
```

```
# Set up data like the data section of stan code
AR1_data <- list(N = length(returns),
                 returns = returns)

AR1_fit <- rstan::stan("AR1.stan",
                       data = AR1_data,
                       iter = 2000,
                       chains = 4,
                       refresh = 500)
```

```
##
## SAMPLING FOR MODEL 'AR1' NOW (CHAIN 1).
## Chain 1:
## Chain 1: Gradient evaluation took 1.3e-05 seconds
## Chain 1: 1000 transitions using 10 leapfrog steps per transition would take 0.13 seconds.
## Chain 1: Adjust your expectations accordingly!
## Chain 1:
## Chain 1:
## Chain 1: Iteration:    1 / 2000 [  0%]  (Warmup)
## Chain 1: Iteration:  500 / 2000 [ 25%]  (Warmup)
## Chain 1: Iteration: 1000 / 2000 [ 50%]  (Warmup)
## Chain 1: Iteration: 1001 / 2000 [ 50%]  (Sampling)
## Chain 1: Iteration: 1500 / 2000 [ 75%]  (Sampling)
## Chain 1: Iteration: 2000 / 2000 [100%]  (Sampling)
## Chain 1:
## Chain 1:  Elapsed Time: 0.041437 seconds (Warm-up)
## Chain 1:                0.040399 seconds (Sampling)
## Chain 1:                0.081836 seconds (Total)
## Chain 1:
##
## SAMPLING FOR MODEL 'AR1' NOW (CHAIN 2).
## Chain 2:
## Chain 2: Gradient evaluation took 6e-06 seconds
## Chain 2: 1000 transitions using 10 leapfrog steps per transition would take 0.06 seconds.
## Chain 2: Adjust your expectations accordingly!
## Chain 2:
## Chain 2:
## Chain 2: Iteration:    1 / 2000 [  0%]  (Warmup)
## Chain 2: Iteration:  500 / 2000 [ 25%]  (Warmup)
## Chain 2: Iteration: 1000 / 2000 [ 50%]  (Warmup)
## Chain 2: Iteration: 1001 / 2000 [ 50%]  (Sampling)
## Chain 2: Iteration: 1500 / 2000 [ 75%]  (Sampling)
## Chain 2: Iteration: 2000 / 2000 [100%]  (Sampling)
## Chain 2:
## Chain 2:  Elapsed Time: 0.038293 seconds (Warm-up)
## Chain 2:                0.039096 seconds (Sampling)
## Chain 2:                0.077389 seconds (Total)
## Chain 2:
##
## SAMPLING FOR MODEL 'AR1' NOW (CHAIN 3).
```

```
## Chain 3:
## Chain 3: Gradient evaluation took 7e-06 seconds
## Chain 3: 1000 transitions using 10 leapfrog steps per transition would take 0.07 seconds.
## Chain 3: Adjust your expectations accordingly!
## Chain 3:
## Chain 3:
## Chain 3: Iteration:    1 / 2000 [  0%]  (Warmup)
## Chain 3: Iteration:  500 / 2000 [ 25%]  (Warmup)
## Chain 3: Iteration: 1000 / 2000 [ 50%]  (Warmup)
## Chain 3: Iteration: 1001 / 2000 [ 50%]  (Sampling)
## Chain 3: Iteration: 1500 / 2000 [ 75%]  (Sampling)
## Chain 3: Iteration: 2000 / 2000 [100%]  (Sampling)
## Chain 3:
## Chain 3:  Elapsed Time: 0.043345 seconds (Warm-up)
## Chain 3:                0.036018 seconds (Sampling)
## Chain 3:                0.079363 seconds (Total)
## Chain 3:
##
## SAMPLING FOR MODEL 'AR1' NOW (CHAIN 4).
## Chain 4:
## Chain 4: Gradient evaluation took 6e-06 seconds
## Chain 4: 1000 transitions using 10 leapfrog steps per transition would take 0.06 seconds.
## Chain 4: Adjust your expectations accordingly!
## Chain 4:
## Chain 4:
## Chain 4: Iteration:    1 / 2000 [  0%]  (Warmup)
## Chain 4: Iteration:  500 / 2000 [ 25%]  (Warmup)
## Chain 4: Iteration: 1000 / 2000 [ 50%]  (Warmup)
## Chain 4: Iteration: 1001 / 2000 [ 50%]  (Sampling)
## Chain 4: Iteration: 1500 / 2000 [ 75%]  (Sampling)
## Chain 4: Iteration: 2000 / 2000 [100%]  (Sampling)
## Chain 4:
## Chain 4:  Elapsed Time: 0.043898 seconds (Warm-up)
## Chain 4:                0.04698 seconds (Sampling)
## Chain 4:                0.090878 seconds (Total)
## Chain 4:
```

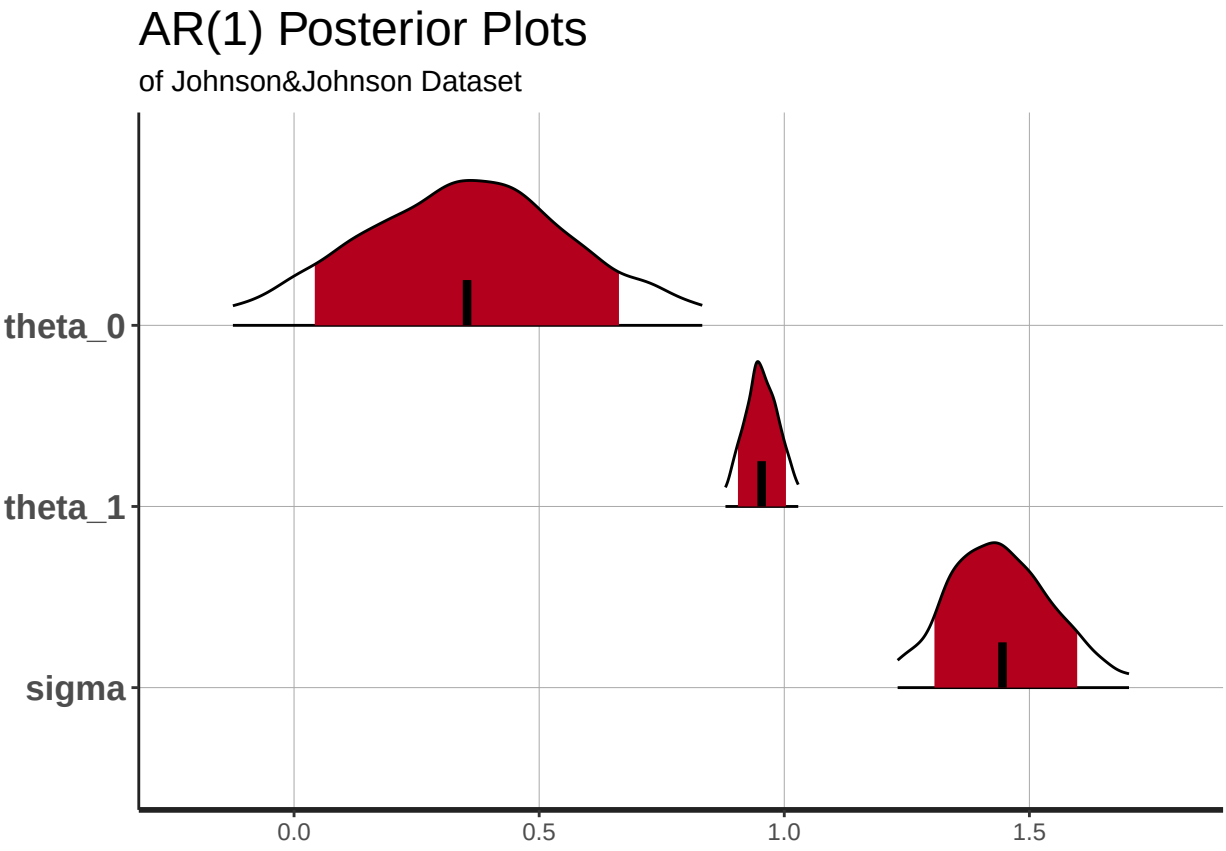
```
# the model
AR1_fit
```

```
## Inference for Stan model: AR1.
## 4 chains, each with iter=2000; warmup=1000; thin=1;
## post-warmup draws per chain=1000, total post-warmup draws=4000.
##
##           mean se_mean   sd   2.5%    25%    50%    75%  97.5% n_eff Rhat
## theta_0   0.35    0.01 0.24  -0.12   0.19   0.35   0.51   0.83  1993    1
## theta_1   0.95    0.00 0.04   0.88   0.93   0.95   0.98   1.03  1930    1
## sigma     1.45    0.00 0.12   1.23   1.36   1.44   1.52   1.70  2420    1
## lp__    -70.84    0.03 1.34 -74.37 -71.39 -70.47 -69.88 -69.37  1512    1
##
## Samples were drawn using NUTS(diag_e) at Sat May  9 02:31:26 2020.
## 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).
```

```r
# posterior plots
stan_plot(AR1_fit,
          point_est = "mean",
          show_density = TRUE) +
     labs(title = 'AR(1) Posterior Plots',
          subtitle = 'of Johnson&Johnson Dataset')
```

```
## ci_level: 0.8 (80% intervals)
```

```
## outer_level: 0.95 (95% intervals)
```



AR(p)

The Stan code is as follows:

```
data {
int<lower=0> P;                         // order of the model
int<lower=0> N;                         // length of TS
vector[N] returns;                        // name of TS vector
}

parameters {
real mu;                                  // mean of TS
real phi[P];                            // real valued theta vector of length p
```

```
real <lower = 0> sigma;                      // std dev of Gaussian noise
}

model {
// specify priors
mu ~ normal(0,4);
phi ~ normal(0, 2);
sigma ~ exponential(2);

// use for loops to set up the likelihood

for (n in (P+1):N){
real average = mu;

for (p in 1:P)
average += phi[p] * returns[n - p];

returns[n] ~ normal(average, sigma);    // mean is the combo of previous P steps in the process
}
}
```

```
# Set up data like the data section of stan code
model_order <- 3
ARp_data <- list(P = model_order, # order of AR model
                 N = length(returns), # length of TS
                 returns = returns # the data
)

ARp_fit <- rstan::stan("ARp.stan",
                       data = ARp_data,
                       iter = 2000,
                       chains = 2,
                       refresh = 500)
```

```
## recompiling to avoid crashing R session

## Trying to compile a simple C file

## Running /Library/Frameworks/R.framework/Resources/bin/R CMD SHLIB foo.c
## clang -mmacosx-version-min=10.13 -I"/Library/Frameworks/R.framework/Resources/include" -DNDEBUG   -I
## In file included from <built-in>:1:
## In file included from /Library/Frameworks/R.framework/Versions/4.0/Resources/library/StanHeaders/incl
## In file included from /Library/Frameworks/R.framework/Versions/4.0/Resources/library/RcppEigen/inclu
## In file included from /Library/Frameworks/R.framework/Versions/4.0/Resources/library/RcppEigen/inclu
## /Library/Frameworks/R.framework/Versions/4.0/Resources/library/RcppEigen/include/Eigen/src/Core/util/
## namespace Eigen {
## ^
## /Library/Frameworks/R.framework/Versions/4.0/Resources/library/RcppEigen/include/Eigen/src/Core/util/
## namespace Eigen {
##                 ^
##                 ;
## In file included from <built-in>:1:
## In file included from /Library/Frameworks/R.framework/Versions/4.0/Resources/library/StanHeaders/incl
```

```
## In file included from /Library/Frameworks/R.framework/Versions/4.0/Resources/library/RcppEigen/inclu
## /Library/Frameworks/R.framework/Versions/4.0/Resources/library/RcppEigen/include/Eigen/Core:96:10: fa
## #include <complex>
##          ^~~~~~~~~
## 3 errors generated.
## make: *** [foo.o] Error 1
##
## SAMPLING FOR MODEL 'ARp' NOW (CHAIN 1).
## Chain 1:
## Chain 1: Gradient evaluation took 3.2e-05 seconds
## Chain 1: 1000 transitions using 10 leapfrog steps per transition would take 0.32 seconds.
## Chain 1: Adjust your expectations accordingly!
## Chain 1:
## Chain 1:
## Chain 1: Iteration:    1 / 2000 [  0%]  (Warmup)
## Chain 1: Iteration:  500 / 2000 [ 25%]  (Warmup)
## Chain 1: Iteration: 1000 / 2000 [ 50%]  (Warmup)
## Chain 1: Iteration: 1001 / 2000 [ 50%]  (Sampling)
## Chain 1: Iteration: 1500 / 2000 [ 75%]  (Sampling)
## Chain 1: Iteration: 2000 / 2000 [100%]  (Sampling)
## Chain 1:
## Chain 1:  Elapsed Time: 0.346231 seconds (Warm-up)
## Chain 1:                0.355777 seconds (Sampling)
## Chain 1:                0.702008 seconds (Total)
## Chain 1:
##
## SAMPLING FOR MODEL 'ARp' NOW (CHAIN 2).
## Chain 2:
## Chain 2: Gradient evaluation took 1.2e-05 seconds
## Chain 2: 1000 transitions using 10 leapfrog steps per transition would take 0.12 seconds.
## Chain 2: Adjust your expectations accordingly!
## Chain 2:
## Chain 2:
## Chain 2: Iteration:    1 / 2000 [  0%]  (Warmup)
## Chain 2: Iteration:  500 / 2000 [ 25%]  (Warmup)
## Chain 2: Iteration: 1000 / 2000 [ 50%]  (Warmup)
## Chain 2: Iteration: 1001 / 2000 [ 50%]  (Sampling)
## Chain 2: Iteration: 1500 / 2000 [ 75%]  (Sampling)
## Chain 2: Iteration: 2000 / 2000 [100%]  (Sampling)
## Chain 2:
## Chain 2:  Elapsed Time: 0.34805 seconds (Warm-up)
## Chain 2:                0.324377 seconds (Sampling)
## Chain 2:                0.672427 seconds (Total)
## Chain 2:
```

```
# the model
ARp_fit
```

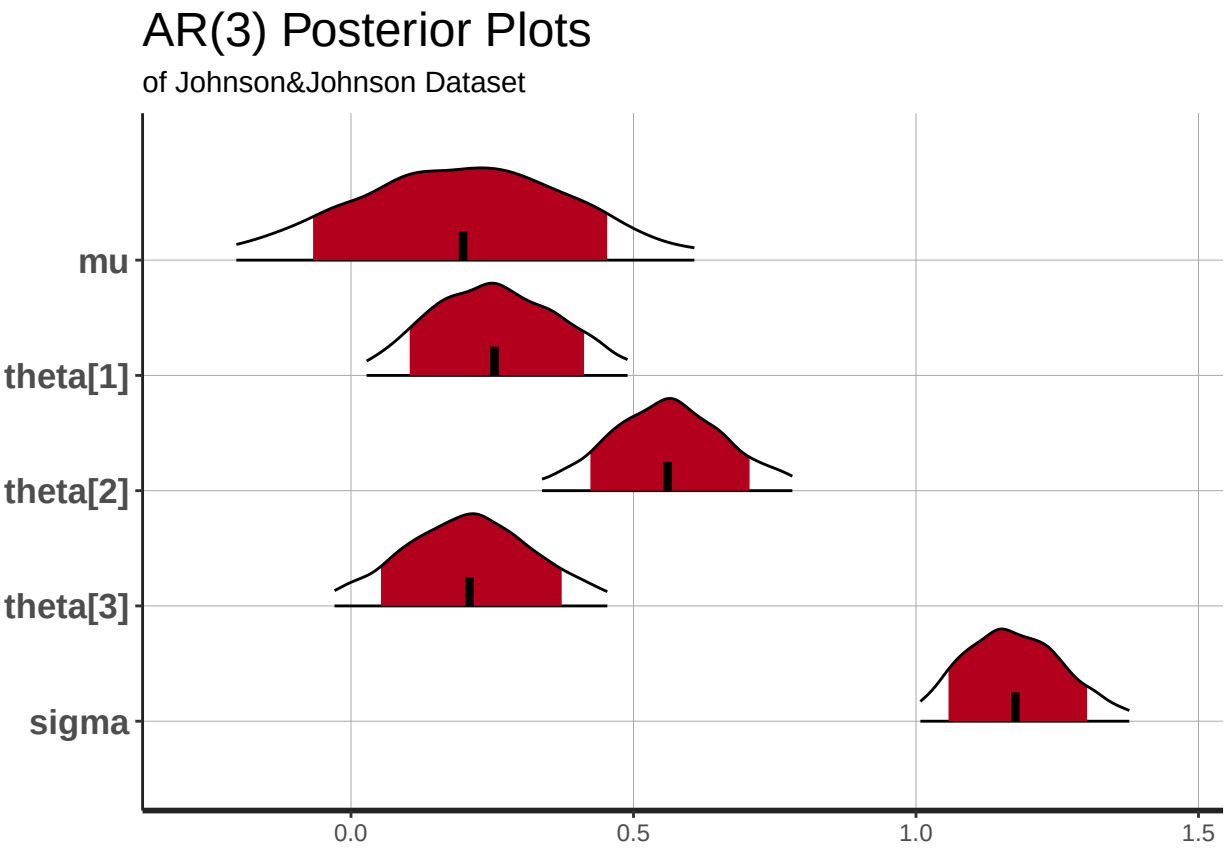
```
## Inference for Stan model: ARp.
## 2 chains, each with iter=2000; warmup=1000; thin=1;
## post-warmup draws per chain=1000, total post-warmup draws=2000.
##
##            mean se_mean   sd   2.5%    25%    50%    75%  97.5% n_eff Rhat
## mu         0.20    0.01 0.20  -0.20   0.06   0.20   0.34   0.61  1557    1
```

```
## theta[1]   0.25    0.00 0.12   0.03   0.17   0.25   0.34   0.49  1093    1
## theta[2]   0.56    0.00 0.11   0.34   0.49   0.56   0.63   0.78  1145    1
## theta[3]   0.21    0.00 0.12  -0.03   0.13   0.21   0.29   0.45  1124    1
## sigma      1.18    0.00 0.10   1.01   1.11   1.17   1.24   1.38  1449    1
## lp__     -56.07    0.08 1.76 -60.37 -56.90 -55.67 -54.86 -53.89   487    1
##
## Samples were drawn using NUTS(diag_e) at Sat May  9 02:32:39 2020.
## 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).
```

```
# posterior plots
stan_plot(ARp_fit,
          point_est = "mean",
          show_density = TRUE) +
     labs(title = 'AR(3) Posterior Plots',
          subtitle = 'of Johnson&Johnson Dataset')
```

```
## ci_level: 0.8 (80% intervals)
```

```
## outer_level: 0.95 (95% intervals)
```



Autoregressive Conditional Heteroskedasticty Model

This section is limited to ARCH(1) models but this can be extended to an ARCH(p) model with a for loop on the variance parameter in the likelihood. The Stan code will be as follows.

```
data{
int<lower = 0> N;                           // length of time series
vector[N] returns;                          // the time series
}

parameters{
real mu;                                    // a constant - average return
real<lower = 0> alpha_0;                    // parameter of ARCH portion - intercept of noise
real<lower = 0, upper = 1> alpha_1;         // another parameter of ARCH - slope of noise
}

model{
// priors
mu ~ normal(0,2);
alpha_0 ~ normal(0, 2);
alpha_1 ~ normal(0,1);

// likelihood - can be vectorized instead of a loop
for(n in 2:N)
returns[n] ~ normal(mu, sqrt(alpha_0 + alpha_1 * pow(returns[n-1] - mu,2)));
}
```

```
# Set up data like the data section of stan code
ARCH1_data <- list(N = length(returns),
                   returns = returns)

ARCH1_fit <- rstan::stan("ARCH1.stan",
                         data = ARCH1_data,
                         iter = 2000,
                         chains = 2,
                         refresh = 500)
```

```
## recompiling to avoid crashing R session

## Trying to compile a simple C file

## Running /Library/Frameworks/R.framework/Resources/bin/R CMD SHLIB foo.c
## clang -mmacosx-version-min=10.13 -I"/Library/Frameworks/R.framework/Resources/include" -DNDEBUG   -I
## In file included from <built-in>:1:
## In file included from /Library/Frameworks/R.framework/Versions/4.0/Resources/library/StanHeaders/incl
## In file included from /Library/Frameworks/R.framework/Versions/4.0/Resources/library/RcppEigen/inclu
## In file included from /Library/Frameworks/R.framework/Versions/4.0/Resources/library/RcppEigen/inclu
## /Library/Frameworks/R.framework/Versions/4.0/Resources/library/RcppEigen/include/Eigen/src/Core/util/
## namespace Eigen {
## ^
## /Library/Frameworks/R.framework/Versions/4.0/Resources/library/RcppEigen/include/Eigen/src/Core/util/
## namespace Eigen {
##                ^
##                ;
## In file included from <built-in>:1:
## In file included from /Library/Frameworks/R.framework/Versions/4.0/Resources/library/StanHeaders/incl
## In file included from /Library/Frameworks/R.framework/Versions/4.0/Resources/library/RcppEigen/inclu
## /Library/Frameworks/R.framework/Versions/4.0/Resources/library/RcppEigen/include/Eigen/Core:96:10: fa
```

```
## #include <complex>
##          ^~~~~~~~
## 3 errors generated.
## make: *** [foo.o] Error 1
##
## SAMPLING FOR MODEL 'ARCH1' NOW (CHAIN 1).
## Chain 1:
## Chain 1: Gradient evaluation took 4e-05 seconds
## Chain 1: 1000 transitions using 10 leapfrog steps per transition would take 0.4 seconds.
## Chain 1: Adjust your expectations accordingly!
## Chain 1:
## Chain 1:
## Chain 1: Iteration:    1 / 2000 [  0%]  (Warmup)
## Chain 1: Iteration:  500 / 2000 [ 25%]  (Warmup)
## Chain 1: Iteration: 1000 / 2000 [ 50%]  (Warmup)
## Chain 1: Iteration: 1001 / 2000 [ 50%]  (Sampling)
## Chain 1: Iteration: 1500 / 2000 [ 75%]  (Sampling)
## Chain 1: Iteration: 2000 / 2000 [100%]  (Sampling)
## Chain 1:
## Chain 1:  Elapsed Time: 0.093177 seconds (Warm-up)
## Chain 1:                0.106443 seconds (Sampling)
## Chain 1:                0.19962 seconds (Total)
## Chain 1:
##
## SAMPLING FOR MODEL 'ARCH1' NOW (CHAIN 2).
## Chain 2:
## Chain 2: Gradient evaluation took 5.6e-05 seconds
## Chain 2: 1000 transitions using 10 leapfrog steps per transition would take 0.56 seconds.
## Chain 2: Adjust your expectations accordingly!
## Chain 2:
## Chain 2:
## Chain 2: Iteration:    1 / 2000 [  0%]  (Warmup)
## Chain 2: Iteration:  500 / 2000 [ 25%]  (Warmup)
## Chain 2: Iteration: 1000 / 2000 [ 50%]  (Warmup)
## Chain 2: Iteration: 1001 / 2000 [ 50%]  (Sampling)
## Chain 2: Iteration: 1500 / 2000 [ 75%]  (Sampling)
## Chain 2: Iteration: 2000 / 2000 [100%]  (Sampling)
## Chain 2:
## Chain 2:  Elapsed Time: 0.090865 seconds (Warm-up)
## Chain 2:                0.093219 seconds (Sampling)
## Chain 2:                0.184084 seconds (Total)
## Chain 2:
```

```
# the model
ARCH1_fit
```

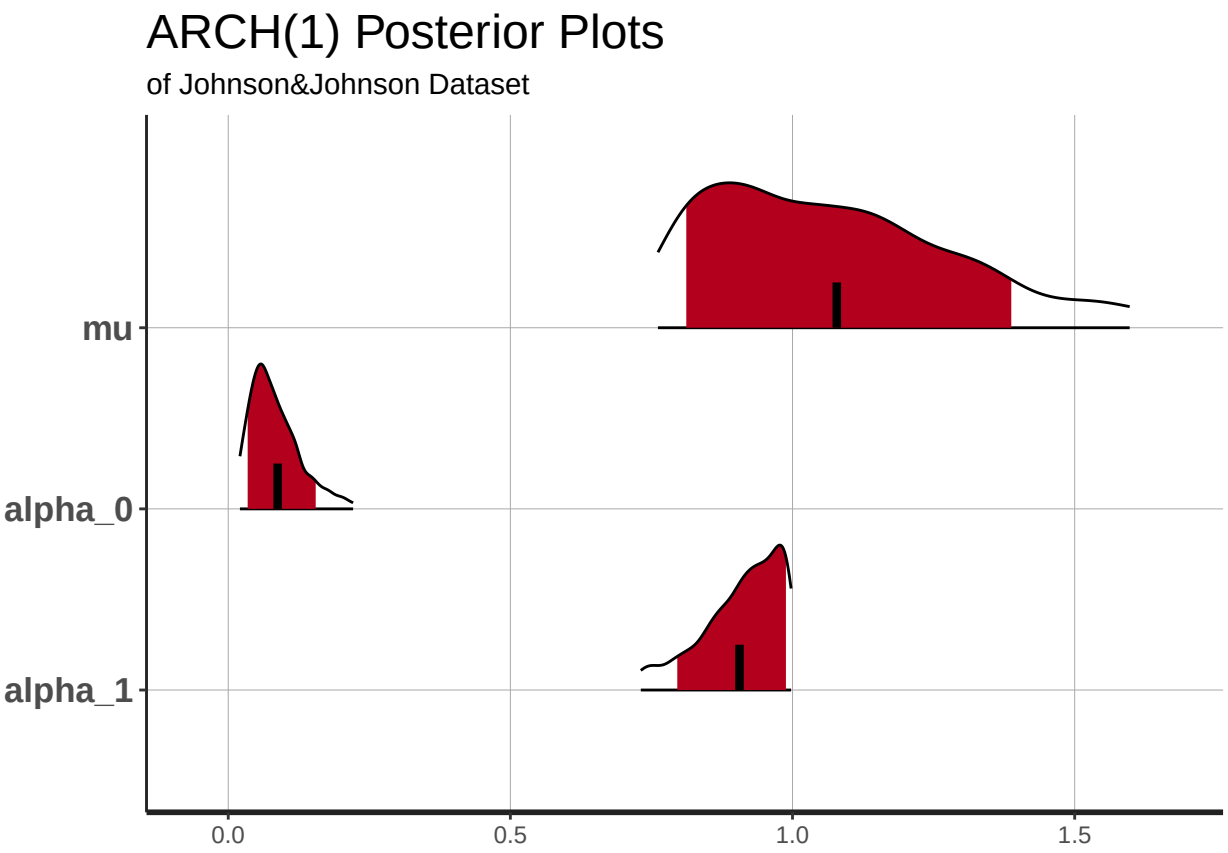
```
## Inference for Stan model: ARCH1.
## 2 chains, each with iter=2000; warmup=1000; thin=1;
## post-warmup draws per chain=1000, total post-warmup draws=2000.
##
##            mean se_mean   sd    2.5%     25%     50%     75%   97.5% n_eff Rhat
## mu         1.08    0.01 0.23    0.76    0.89    1.05    1.22    1.60   444    1
## alpha_0    0.09    0.00 0.05    0.02    0.05    0.08    0.11    0.22   615    1
## alpha_1    0.91    0.00 0.07    0.73    0.86    0.92    0.97    1.00  1063    1
```

```
## lp__    -102.48    0.06 1.42 -106.40 -103.08 -102.14 -101.48 -100.84   481    1
##
## Samples were drawn using NUTS(diag_e) at Sat May  9 02:33:53 2020.
## 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).
```

```
# posterior plots
stan_plot(ARCH1_fit,
          point_est = "mean",
          show_density = TRUE) +
     labs(title = 'ARCH(1) Posterior Plots',
          subtitle = 'of Johnson&Johnson Dataset')
```

```
## ci_level: 0.8 (80% intervals)
```

```
## outer_level: 0.95 (95% intervals)
```



Moving Average Model

Moving average models of order q, denoted MA(q), take the form:

$$y_t = \theta_1 y\epsilon_{t-1} + \ldots + \theta_p epsilon_{t-q} + \epsilon_t \tag{12}$$

$$\epsilon_t \sim N(0, \sigma^2) \tag{13}$$

The parameters we need to estimate are $(\theta_1, \ldots, \theta_p, \sigma)$. The Stan code is as follows.

```
data {
int<lower = 0> Q;                                   // order of model
int<lower = 1> T;                                   // length of time series
vector[T] returns;                                  // time series vector
}
parameters {
real mu;                                            // mean of the time series
real theta[Q];                                      // the coefficients
real<lower = 0> sigma;                              // noise scale parameter
}
model {
// useful parameters
vector[T] previous_steps;                           // prediction at time t (the moving average)
vector[T] error;                                    // noise at each step

// priors
mu ~ cauchy(0, 2.5);
theta ~ normal(0, 2);
sigma ~ exponential(2.01);

// likelihood
for(t in 1:T){
previous_steps[t] = mu;
error[t] = returns[t] - mu;

for(q in 1:min(t-1, Q)){
previous_steps[t] = previous_steps[t] + theta[q] * error[t-q];
error[t] = error[t] - theta[q] * error[t-q];
}
}

returns ~ normal(previous_steps, sigma);
}
```

```
# Set up data like the data section of stan code
model_order = 4
MAq_data <- list(Q = model_order, # order of AR model
                 T = length(returns), # length of TS
                 returns = returns # the data
)

MAq_fit <- rstan::stan("MAq.stan",
                       data = MAq_data,
                       iter = 2000,
                       chains = 2,
                       refresh = 500,
                       pars = c("mu", "theta","sigma"))
```

```
## recompiling to avoid crashing R session

## Trying to compile a simple C file

## Running /Library/Frameworks/R.framework/Resources/bin/R CMD SHLIB foo.c
```

```
## clang -mmacosx-version-min=10.13 -I"/Library/Frameworks/R.framework/Resources/include" -DNDEBUG   -I
## In file included from <built-in>:1:
## In file included from /Library/Frameworks/R.framework/Versions/4.0/Resources/library/StanHeaders/incl
## In file included from /Library/Frameworks/R.framework/Versions/4.0/Resources/library/RcppEigen/inclu
## In file included from /Library/Frameworks/R.framework/Versions/4.0/Resources/library/RcppEigen/inclu
## /Library/Frameworks/R.framework/Versions/4.0/Resources/library/RcppEigen/include/Eigen/src/Core/util/
## namespace Eigen {
## ^
## /Library/Frameworks/R.framework/Versions/4.0/Resources/library/RcppEigen/include/Eigen/src/Core/util/
## namespace Eigen {
##                ^
##                ;
## In file included from <built-in>:1:
## In file included from /Library/Frameworks/R.framework/Versions/4.0/Resources/library/StanHeaders/incl
## In file included from /Library/Frameworks/R.framework/Versions/4.0/Resources/library/RcppEigen/inclu
## /Library/Frameworks/R.framework/Versions/4.0/Resources/library/RcppEigen/include/Eigen/Core:96:10: fa
## #include <complex>
##          ^~~~~~~~
## 3 errors generated.
## make: *** [foo.o] Error 1
##
## SAMPLING FOR MODEL 'MAq' NOW (CHAIN 1).
## Chain 1:
## Chain 1: Gradient evaluation took 6.2e-05 seconds
## Chain 1: 1000 transitions using 10 leapfrog steps per transition would take 0.62 seconds.
## Chain 1: Adjust your expectations accordingly!
## Chain 1:
## Chain 1:
## Chain 1: Iteration:    1 / 2000 [  0%]  (Warmup)
## Chain 1: Iteration:  500 / 2000 [ 25%]  (Warmup)
## Chain 1: Iteration: 1000 / 2000 [ 50%]  (Warmup)
## Chain 1: Iteration: 1001 / 2000 [ 50%]  (Sampling)
## Chain 1: Iteration: 1500 / 2000 [ 75%]  (Sampling)
## Chain 1: Iteration: 2000 / 2000 [100%]  (Sampling)
## Chain 1:
## Chain 1:  Elapsed Time: 0.594555 seconds (Warm-up)
## Chain 1:                0.278093 seconds (Sampling)
## Chain 1:                0.872648 seconds (Total)
## Chain 1:
##
## SAMPLING FOR MODEL 'MAq' NOW (CHAIN 2).
## Chain 2:
## Chain 2: Gradient evaluation took 2.3e-05 seconds
## Chain 2: 1000 transitions using 10 leapfrog steps per transition would take 0.23 seconds.
## Chain 2: Adjust your expectations accordingly!
## Chain 2:
## Chain 2:
## Chain 2: Iteration:    1 / 2000 [  0%]  (Warmup)
## Chain 2: Iteration:  500 / 2000 [ 25%]  (Warmup)
## Chain 2: Iteration: 1000 / 2000 [ 50%]  (Warmup)
## Chain 2: Iteration: 1001 / 2000 [ 50%]  (Sampling)
## Chain 2: Iteration: 1500 / 2000 [ 75%]  (Sampling)
## Chain 2: Iteration: 2000 / 2000 [100%]  (Sampling)
## Chain 2:
```

```
## Chain 2:  Elapsed Time: 0.126988 seconds (Warm-up)
## Chain 2:                2.93926 seconds (Sampling)
## Chain 2:                3.06625 seconds (Total)
## Chain 2:
```

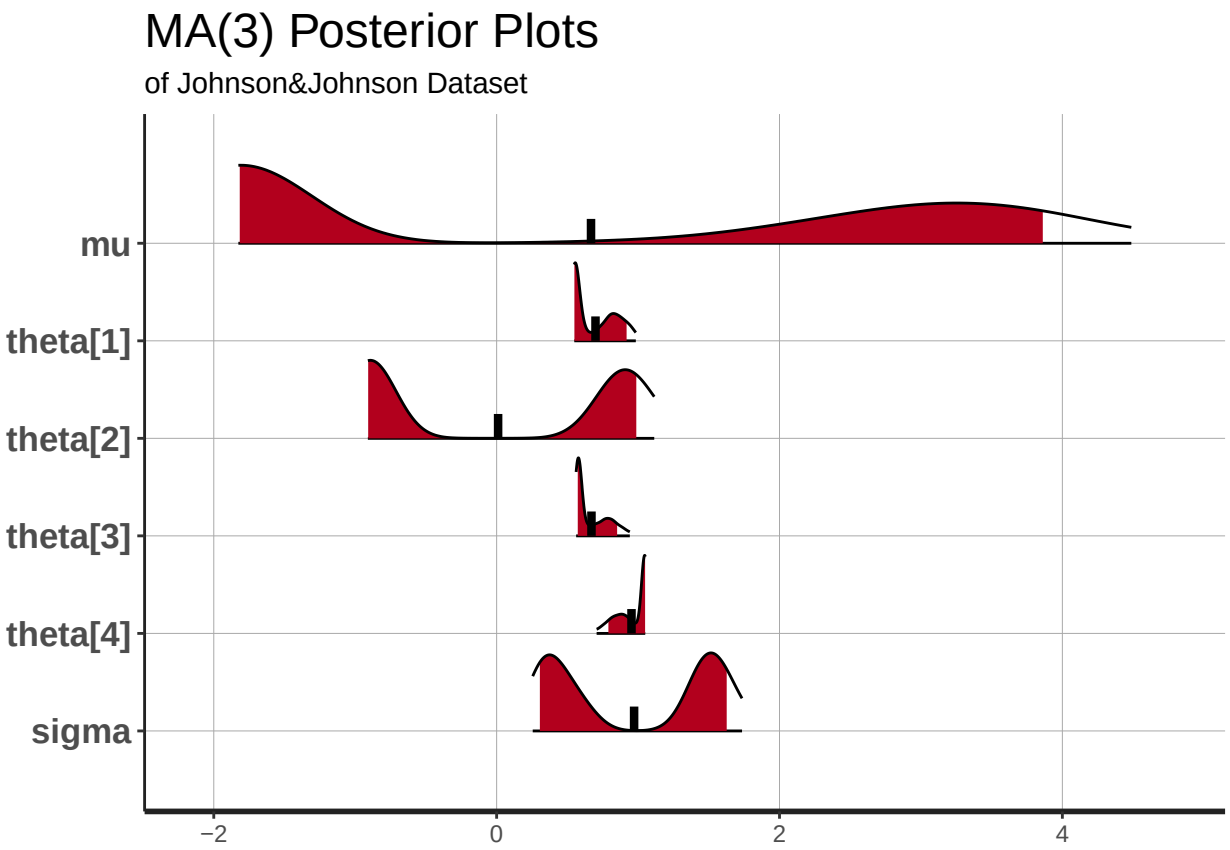
```
# the model
MAq_fit
```

```
## Inference for Stan model: MAq.
## 2 chains, each with iter=2000; warmup=1000; thin=1;
## post-warmup draws per chain=1000, total post-warmup draws=2000.
##
##               mean  se_mean       sd      2.5%       25%      50%    75%  97.5%
## mu            0.67     2.50     2.55     -1.83     -1.81    -0.63   3.19   4.49
## theta[1]      0.70     0.15     0.16      0.55      0.55     0.63   0.84   0.98
## theta[2]      0.01     0.90     0.91     -0.91     -0.91    -0.09   0.91   1.11
## theta[3]      0.67     0.10     0.12      0.56      0.58     0.59   0.77   0.94
## theta[4]      0.95     0.10     0.11      0.71      0.87     1.03   1.05   1.05
## sigma         0.97     0.56     0.57      0.26      0.40     0.97   1.52   1.73
## lp__     -22218.17 22029.30 27733.59 -96492.26 -37765.56 -5873.52 -78.27 -76.07
##          n_eff  Rhat
## mu           1  4.89
## theta[1]     1  2.48
## theta[2]     1 13.68
## theta[3]     1  1.68
## theta[4]     1  1.96
## sigma        1  6.82
## lp__         2  3.44
##
## Samples were drawn using NUTS(diag_e) at Sat May  9 02:35:15 2020.
## 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).
```

```
# posterior plots
stan_plot(MAq_fit,
          point_est = "mean",
          show_density = TRUE) +
     labs(title = 'MA(3) Posterior Plots',
          subtitle = 'of Johnson&Johnson Dataset')
```

```
## ci_level: 0.8 (80% intervals)
```

```
## outer_level: 0.95 (95% intervals)
```



The posterior plots look like houses from a Dr. Seuss book which means that this is a multimodal posterior distribution.

Auto-regressive Moving Average Model

Auto-regressuve Moving average models of order (p,q), denoted ARMA(p,q), take the form:

$$y_t = \phi_1 y_{t-1} + \ldots + \phi_p y_{t-p} + \theta_1 y \epsilon_{t-1} + \ldots + \theta_p epsilon_{t-q} + \epsilon_t \tag{14}$$

$$\epsilon_t \sim N(0, \sigma^2) \tag{15}$$

The parameters we need to estimate are $(\phi_1, \ldots, \phi_p, \theta_1, \ldots, \theta_p, \sigma)$. The Stan code is as follows.

```
data {
int<lower = 0> P;                       // AR Order
int<lower = 0> Q;                       // MA Order
int<lower=1> T;                         // num observations
real y[T];                              // observed outputs
}
parameters {
real mu;                                // mean coeff
vector[P] phi;                          // autoregression coeff
vector[Q] theta;                        // moving avg coeff
real<lower=0> sigma;                    // noise scale
}
model {
vector[T] nu;                           // prediction for time t
```

```
vector[T] err;                          // error for time t

// initialization
for(t in 1:max(P,Q)){
nu[t] = mu + phi[t] * mu;           // assume err[0] == 0
err[t] = y[t] - nu[t];
}

for (t in (max(P,Q)+1):T) {
nu[t] = mu;
//
// AR component
for(p in 1:P){
nu[t] = nu[t] + phi[p] * y[t-p];

}
// MA component
for(q in 1:Q){
nu[t] = nu[t] + theta[q] * y[t-q];
}
//
// error computation
err[t] = y[t] - nu[t];
}

// priors
mu ~ normal(0, 10);
phi ~ normal(0, 2);
theta ~ normal(0, 2);
sigma ~ cauchy(0, 5);

// likelihood
err ~ normal(0, sigma);
}
```

```
# Set up data like the data section of stan code
AR_order = 2
MA_order = 2

ARMApq_data <- list(P = AR_order,
                    Q = MA_order,
                    T = length(returns),
                    y = returns)

ARMApq_fit <- rstan::stan('~/Documents/Github/RStan/ARMAV2.stan',
                          data = ARMApq_data,
                          iter = 4000,
                          chains = 1,
                          thin = 1,
                          algorithm = 'HMC')
```

```
## Trying to compile a simple C file

## Running /Library/Frameworks/R.framework/Resources/bin/R CMD SHLIB foo.c
```

```
## clang -mmacosx-version-min=10.13 -I"/Library/Frameworks/R.framework/Resources/include" -DNDEBUG   -I
## In file included from <built-in>:1:
## In file included from /Library/Frameworks/R.framework/Versions/4.0/Resources/library/StanHeaders/incl
## In file included from /Library/Frameworks/R.framework/Versions/4.0/Resources/library/RcppEigen/inclu
## In file included from /Library/Frameworks/R.framework/Versions/4.0/Resources/library/RcppEigen/inclu
## /Library/Frameworks/R.framework/Versions/4.0/Resources/library/RcppEigen/include/Eigen/src/Core/util/
## namespace Eigen {
## ^
## /Library/Frameworks/R.framework/Versions/4.0/Resources/library/RcppEigen/include/Eigen/src/Core/util/
## namespace Eigen {
##                ^
##                ;
## In file included from <built-in>:1:
## In file included from /Library/Frameworks/R.framework/Versions/4.0/Resources/library/StanHeaders/incl
## In file included from /Library/Frameworks/R.framework/Versions/4.0/Resources/library/RcppEigen/inclu
## /Library/Frameworks/R.framework/Versions/4.0/Resources/library/RcppEigen/include/Eigen/Core:96:10: fa
## #include <complex>
##          ^~~~~~~~
## 3 errors generated.
## make: *** [foo.o] Error 1
##
## SAMPLING FOR MODEL 'ARMAV2' NOW (CHAIN 1).
## Chain 1:
## Chain 1: Gradient evaluation took 3.5e-05 seconds
## Chain 1: 1000 transitions using 10 leapfrog steps per transition would take 0.35 seconds.
## Chain 1: Adjust your expectations accordingly!
## Chain 1:
## Chain 1:
## Chain 1: Iteration:    1 / 4000 [  0%]  (Warmup)
## Chain 1: Iteration:  400 / 4000 [ 10%]  (Warmup)
## Chain 1: Iteration:  800 / 4000 [ 20%]  (Warmup)
## Chain 1: Iteration: 1200 / 4000 [ 30%]  (Warmup)
## Chain 1: Iteration: 1600 / 4000 [ 40%]  (Warmup)
## Chain 1: Iteration: 2000 / 4000 [ 50%]  (Warmup)
## Chain 1: Iteration: 2001 / 4000 [ 50%]  (Sampling)
## Chain 1: Iteration: 2400 / 4000 [ 60%]  (Sampling)
## Chain 1: Iteration: 2800 / 4000 [ 70%]  (Sampling)
## Chain 1: Iteration: 3200 / 4000 [ 80%]  (Sampling)
## Chain 1: Iteration: 3600 / 4000 [ 90%]  (Sampling)
## Chain 1: Iteration: 4000 / 4000 [100%]  (Sampling)
## Chain 1:
## Chain 1:  Elapsed Time: 14.1972 seconds (Warm-up)
## Chain 1:                8.65214 seconds (Sampling)
## Chain 1:                22.8494 seconds (Total)
## Chain 1:
```

```
# the model
ARMApq_fit
```

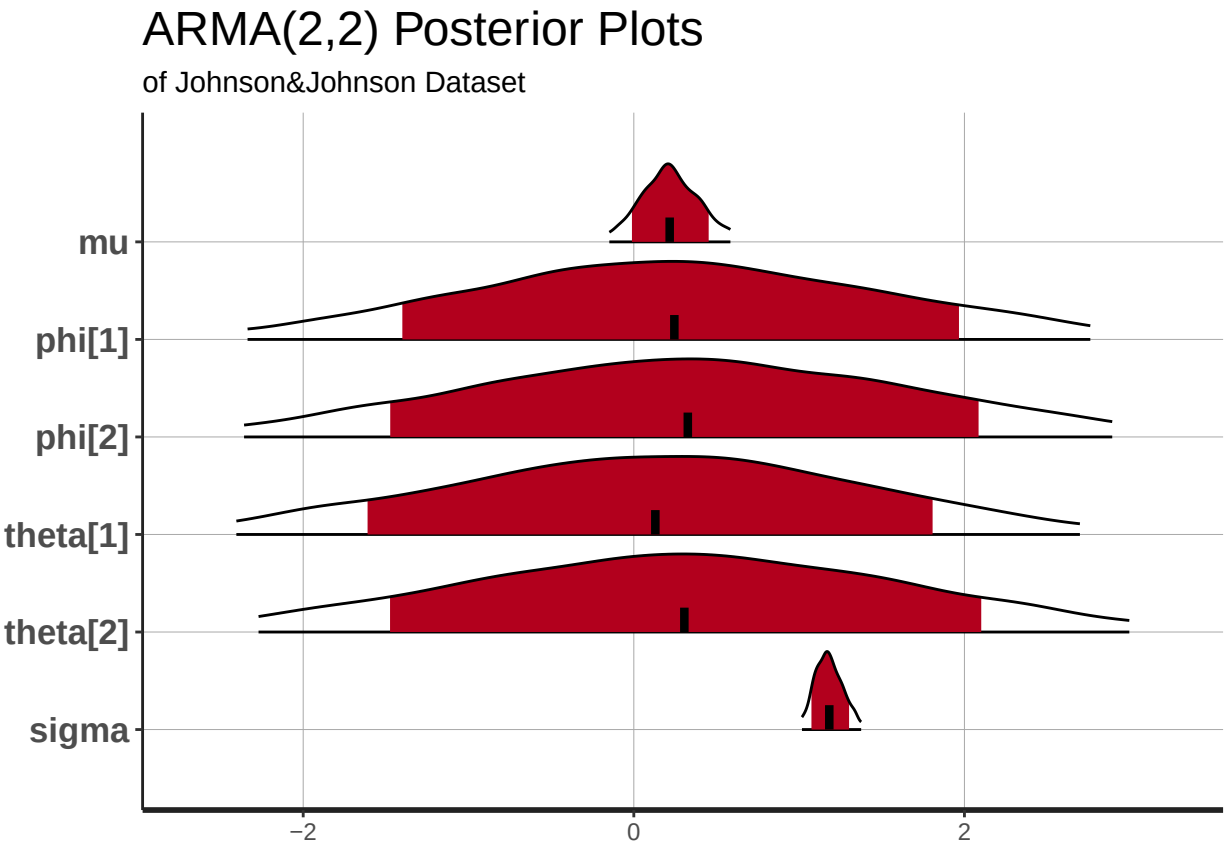
```
## Inference for Stan model: ARMAV2.
## 1 chains, each with iter=4000; warmup=2000; thin=1;
## post-warmup draws per chain=2000, total post-warmup draws=2000.
##
##            mean se_mean   sd   2.5%    25%    50%    75%  97.5% n_eff Rhat
```

```
## mu         0.22    0.00 0.18  -0.15   0.09   0.21   0.34   0.58  6602    1
## phi[1]     0.24    0.02 1.31  -2.34  -0.61   0.24   1.14   2.76  6602    1
## phi[2]     0.33    0.02 1.37  -2.36  -0.63   0.33   1.28   2.91  5233    1
## theta[1]   0.13    0.02 1.31  -2.40  -0.76   0.14   1.00   2.70  6602    1
## theta[2]   0.30    0.02 1.37  -2.28  -0.64   0.31   1.24   3.00  5252    1
## sigma      1.18    0.00 0.09   1.02   1.12   1.18   1.24   1.38  6602    1
## lp__     -56.06    0.08 1.71 -60.18 -57.04 -55.72 -54.81 -53.71   430    1
##
## Samples were drawn using HMC(diag_e) at Sat May  9 02:36:48 2020.
## 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).
```

```
# posterior plots
stan_plot(ARMApq_fit,
          point_est = "mean",
          show_density = TRUE) +
     labs(title = 'ARMA(2,2) Posterior Plots',
          subtitle = 'of Johnson&Johnson Dataset')
```

```
## ci_level: 0.8 (80% intervals)
```

```
## outer_level: 0.95 (95% intervals)
```



Acknowledgments

The material for this page was sourced from the (Stan User's Guide)[https://mc-stan.org/docs/2_23/stan-users-guide/index.html].