A brief introduction to econometrics in Stan

James Savage

2017-06-18

Contents

About

These notes are for a half-day short course in econometrics using Stan. The main reason to learn Stan is to fit models that are difficult to fit using other software. Such models might include models with high-dimensional random effects (about which we want to draw inference), models with complex or multi-stage likelihoods, or models with latent data structures. A second reason to learn Stan is that you want to conduct Bayesian analysis on workhorse models; perhaps you have good prior information, or are attracted to the possibility of making probabilistic statements about predictions and parameter estimates.

While this second reason is worthwhile, it is not the aim of this course. This course introduces a few workhorse models in order to give you the skills to build richer models that extract the most information from your data. There are three sessions:

1. An introduction to Bayesian reasoning, MCMC/HMC, and Stan.
2. An introduction to Modern Statistical Workflow, using a time-series model as the example.
3. Applying the workflow to a more complex model—in this case, aggregate random coefficients logit.

These notes have a few idiosyncracies:

> Tricks and shortcuts will look like this

The code examples live in the `models/` folder of the book's repository, (https://github.com/khakieconomics/shortcourse/models).

We use two computing languages in these notes. The first is Stan, a powerful modeling language that allows us to express and estimate probabilistic models with continuous parameter spaces. Stan programs are prefaced with their location in the `models/` folder, like so:

```
// models/model_1.stan
// ...  model code here
```

We also use the `R` language, for data preparation, calling Stan models, and visualising model results. R programs live in the `scripts/` folder; they typically read data from the `data/` folder, and liberally use `magrittr` syntax with `dplyr`. If this syntax is unfamiliar to you, it is worth taking a look at the excellent vignette to the `dplyr` package. Like the Stan models, all R code in the book is prefaced with its location in the book's directory.

```
# scripts/intro.R
# ... data work here
```

It is not necessary to be an R aficionado to make the most of these notes. Stan programs can be called from within Stata, Matlab, Mathematica, Julia and Python. If you are more comfortable using those languages than R for data preparation work, then you should be able to implement all the models in this book using those interfaces. Further documentation on calling Stan from other environments is available at http://mc-stan.org/interfaces/.

While Stan can be called quite easily from these other programming environments, the R implementation is more fully-fleshed—especially for model checking and post-processing. For this reason we use the R implementation of Stan, `rstan` in this book.

Chapter 1

An introduction to Stan

1.1 Why might you want to start learning Bayesian methods?

Learning Bayesian modeling does require a time investment. If you build and estimate statistical models for a living, it is probably an investment worth making, but we should be very explicit about the benefits (and costs) up-front. The benefits are many. Using Bayesian methods, we can take advantage of information that does not necessarily exist in our data (or model structure) in estimating our model. We can combine sources of information in a simple, coherent fashion. Uncertainty in our predictions automatically incorporates uncertainty in parameter estimates. We can define models that are arbitrarily rich—potentially with more parameters than we have data-points— and still expect coherent parameter estimates. We don't use tests; instead we just check the implied probabilities of outcomes of interest in our estimated model, whose parameters we can give probabilistic interpretations. And perhaps most importantly, using Bayesian methods forces us to understand our models far more deeply than with canned routines.

These benefits don't come free. The approach we advocate in this book—estimating models using full Markov Chain Monte Carlo—can appear slow. Learning Bayesian techniques, and a new programming language, is costly. And some fields have strong frequentist cultures, making communication of Bayesian results an important part of your work. We feel these costs are small relative to the potential gains.

Let's illustrate these benefits with examples. These examples are from real-world applied work by ourselves and our colleagues; hopefully you will see analogies with your own work.

Benefit 1: Incorporating knowledge from outside the data

When we run a field experiment, we typically want to evaluate the impact of some experimental *treatment* on an outcome (experiments are discussed in Chapter **??**). This impact is known as a *treatment effect*. Experiments can be costly do perform, limiting the number of observations that we can collect. Consequently, in these small-data studies it is common to have very imprecise estimates of the treatment effect.

Bayesian analysis of the experiment can help when there is more information available about the the treatment effect than exists the observations from our experiment, as would be the case of there having been previous studies of the same treatment effect. These previous studies' results can be incorporated into our study using what is known as a *hierarchical prior*, resulting in more precise estimates of the treatment effect. We provide a worked example of this in section **??**.

For example, imagine you are an education researcher evaluating the impact of a trendy new educational teaching method on test scores. You run a randomized experiment on seventy students at one school and learn that the intervention improved test scores by 34 points on a scale from to 800. Because of the fairly small sample size, this estimate has a standard error of 23.5, and is not "statistically significant" with a p-value of 0.15 (greater than the arbitrary 0.05 threshold used for declaring statistical significance in many

fields). This is consistent with a 95% confidence interval of the estimate of (-12.8 to 80.9). If you were to roll out the intervention on a large cohort of students, what would you expect the treatment effect to be? 0—our estimate is not statistically significant? 34? Some other number?

Now suppose that because this educational intervention is trendy, it is being experimented on by other researchers. You find that these researchers have achieved the following estimates of the treatment effect from the same treatment (this is Rubin's famous 8 Schools data (Rubin, 1981)):

After seeing these other study results of the same intervention, how might your expectations of a roll-out of the intervention change? It turns out that we can use these study results in re-analyzing our own data in order to get a more precise estimate of the treatment effect in our own study. Doing so reduces the 95% "credibility interval" to (). We work through this exercise in section **??**.

Benefit 2: Combining sources of information

A similar type of analysis to the is to use Bayesian methods to combine sources of information, as in a meta-analysis or political poll aggregation. The aim of this type of research is to estimate a statistic—for instance, the distribution of political preferences, or a treatment effect—you would expect to find in as-yet untested populations using previous studies as the data source, not the underlying data. The central notion is that these previous studies are noisy estimates of some underlying statistic that applies to the whole population. They are noisy partly because of sampling variability, but also because the studies might differ systematically from one another (political pollsters using different questions, for example). An important assumption is that together, these studies are not systematically biased once we account for observable differences between them.

A recent example of this type of analysis is in (Meager, 2016), who uses hierarchical Bayesian modeling to obtain estimates of the generalizable treatment effects (and quantile effects) of microcredit expansions on various household measures of financial success.

The study makes several contributions, but two highlight the power of a (hierarchical) Bayesian approach. The first is that a biproduct of the Bayesian aggregation procedure is an estimate of the generalizability of a given previous study. That is, the procedure tells us how much we can expect to learn about the impact of a yet-untried microcredit expansion from a given experiment. The second is that using a hierarchical Bayesian aggregation procedure gives us new estimates for the treatment effects in the previous studies. Remember: the estimates from those previous studies are noisy. The technique reduces the noise in these estimates by "borrowing power" from other studies. In the report, several of the "statistically significant" findings in the previous studies lose their "significance" once we adjust them for the fact that similar studies find much smaller (or zero) effects. In a world in which costly policy decisions might be influenced by false discoveries, this feature is appealing.

Benefit 3: Dealing with uncertainty consistently in model predictions

We often use models to generate predictions or forecasts. There are several types of uncertainty that we ought to be concerned with. The first is sampling variability: even if we have the "perfect model" (we don't believe such a thing exists) there will remain variation in what we are predicting, either because of pure randomness or measurement error. If we were to use the model for predictions, we should expect the model to be right *on average*, but not right in every instance.

The second source of uncertainty is uncertainty in the unknown parameters in our model. A model itself typically combines "knowns" and unknown variables, and the goal of estimating a model is to draw inference about the value of the unknowns. So long as we only have a limited number of observations, we will be unsure of the precise values of the unknowns; this contributes to uncertainty in our predictions.

The third source of uncertainty is uncertainty about whether the model is the right model for the job— is it correctly specified, and are we modeling a stationary (or non-changing) set of relationships? If the model is improperly specified or if the fundamental relationships of the system are changing, our model will not

perform well. Importantly, this type of uncertainty is not represented by predictive intervals—it is therefore prudent to treat predictive intervals as the *minimum* amount of uncertainty that we should have over the outcome.

By using Bayesian methods, we automatically deal with the first and second sources of uncertainty, without resorting to workarounds. Non-Bayesians can do this as well (for example, by boostrapping), but it is not an inherent part of the procedure. But neither Bayesian nor non-Bayesian techniques deal well with the problem of poorly-specificed models and "model non-stationarity". So what do we do? Using a well thought-through workflow and "generative reasoning" (see **??**) can help us iterate towards a specification that describes our data on hand well. Unfortunately, model non-stationarity is a difficult, philosophical problem. There are no quick fixes.

Benefit 4: Regularizing richly parameterized models

Many important predictive/control variables have high dimensionsionality. For example, imagine you are building a model to predict whether the a package deliverd from town A to B will be delivered late. An important predictive variable will be the origin and destination towns. In most countries, there are a very large number of post-codes, whose values have no intrinsic meaning (so we cannot use these as a numerical control variable). How can we use these values to generate better predictions? And how should we learn from towns with very few observations vis-à-vis towns with many?

How do we deal with this problem without using Bayesian methods? One popular technique is to "one-hot" encode the categorical predictors, so that our data go from

to

And then continue apace. The problem with such an approach is that our parameter space becomes extremely large, and we might end up doing "noise-mining"—discovering relationships where there are none, simply through bad luck. A trick known as "regularization" can help prevent this, and is considered good practice when you have a model with many variables. It does so by "penalizing" parameter estimates that are not estimated precisely—for instance, those zip codes with very few observations—"shrinking" the parameter estimates towards zero (or some other value). Regularization like this is widely used in machine learning. A difficulty is that most canned routines that implement regularization do not allow for easy incorporation of uncertainty in the parameter estimates.

Bayesian methods also employ regularization, through the use of priors. This is because our parameter estimates will always be between the likelihood estimate (typically, the estimates you'll get from a non-Bayesian approach) and our priors. Strictly, priors should encode the information that we have about parameter estimates before we estimate a model. One valuable type of prior information that we have is "regularization works!" or "mean reversion happens!" These partly motivate the approaches we advocate for modeling hierarchical and panel data in chapter **??**, especially the use of hierarchical priors.

Benefit 5: Doing away with tests

In frequentist statistics, inference is performed through the use of tests. Testing is normally the process of combining model and data to come up with a *test statistic*, which will have some large-sample limiting distribution *under the assumption that the null hypothesis is correct*. We then compare the test statistic to that limiting distribution to determine whether there exists sufficient evidence to reject the null hypothesis.

Done well, testing can yield useful insights and help guide modeling. But as an approach to workflow and science, testing is difficult to learn and remember, easy to abuse, and with limited data, can result in many erroneous conclusions.

In Bayesian analysis, we do not use tests. All inference is conducted by analysing our fitted model (the *posterior*), which is typically a matrix of draws from the joint distribution of all parameters and predictions. The posterior has a probabilistic interpretation, and consequently if we want to make a probabilistic statement

about our fitted model, we only need to count the number of draws from the posterior for which a condition holds, and divide it by the number of draws.

This is a far more intuitive and easy-to-use way of conducting inference.

1.2 Models and inference

The type of modeling we are doing here is known as *generative modeling*, typically with fairly strong assumptions about the parametric sampling distributions that give rise to our data. For instance, we might have a normal linear regression model

$$y_i = X_i\beta + \epsilon \text{ where } \epsilon \sim \text{Normal}(0,\, \sigma)$$

Which for many scale-location type distributions, can be written as

$$y_i \sim \text{Normal}(X_i\beta,\, \sigma)$$

Which simply says that y_i is generated according to a normal distribution with a location $X_i\beta$, and with residuals of average size σ. In the case of the normal distribution, the location is the mean or expected value, and the scale of the residuals is the residual standard deviation. This is the *generative model*—the model which we say gives rise to_generates_ the data y. (Nb. in this case we refer to X as being *conditioning information*, ie data that is not generated within the model; if we do not know its value out of sample we should be modeling it too!) In this course we'll restrict ourselves to generative models where the generative distribution has a known parametric form.

Don't be put off by the choice of a normal distribution. If we thought that large deviations from the expected value were not unexpected, we might use a fat-tailed distribution, like

$$y_i \sim \text{Student's t}(\nu, X_i\beta,\, \sigma)$$

or

$$y_i \sim \text{Cauchy}(X_i\beta,\, \sigma)$$

In any case, the above are *models*. The models themselves have *fixed* unknown parameters and, possibly, *fixed* latent variables, about which we want to conduct probabilistic inference. That is, we want to be able to make probabilistic statements about the true (out of sample) values of the the fixed values of unknown parameters and latent variables.

It is common to use different techniques to do this, with the technique depending on the model. For example,

1.3 Why use Stan?

Once you've accepted that you should use Bayesian methods to fit your models, you're left with a choice of how to do it. Roughly, you have a few options:

1.4 Bayes rule, likelihood and priors

1.5 A quick introduction to Hamiltonian Monte Carlo

1.6 A tour of a Stan program

Chapter 2

Modern Statistical Workflow

This session introduces the process I recommend for model building, which I call "Modern Statistical Workflow".

2.1 Modern Statistical Workflow

The workflow described here is a template for all the models that will be discussed during the course. If you work by it, you will learn models more thoroughly, spot errors more swiftly, and build a much better understanding of economics and statistics than you would under a less rigorous workflow.

The workflow is iterative. Typically we start with the simplest possible model, working through each step in the process. Only once we have done each step do we add richness to the model. Building models up like this in an iterative way will mean that you always have a working version of a model to fall back on. The process is:

1. Write out a full probability model. This involves specifying the joint distribution for your parameters/latent variables and the conditional distribution for the outcome data.
2. Simulate some data from the model with assumed values for the parameters (these might be quite different from the "true" parameter values).
3. Estimate the model using the simulated data. Check that your model can recover the known parameters used to simulate the data.
4. Estimate the model parameters conditioned on real data.
5. Check that the estimation has run properly.
6. Run posterior predictive checking/time series cross validation to evaluate model fit.
7. Perform predictive inference.

Iterate the entire process to improve the model! Compare models—which model are the observed outcomes more plausibly drawn from?

2.1.1 Example: A model of wages

Before building any model, it is always worth writing down the questions that we might want to ask. Sometimes, the questions will be relativey simple, like "what is the difference in average wages between men and women?" Yet for most large-scale modeling tasks we want to build models capable of answering many questions. In the case of wages, they may be questions like:

- If I know someone is male and lives in the South what should I expect their wages to be, holding other personal characteristics constant?
- How much does education affect wages?

- Workers with more work experience tend to earn higher wages. How does this effect vary across demographic groups?
- Does variance in wages differ across demographic groups?

As a good rule of thumb, the more questions you want a model to be able to answer, the more complex the model will have to be. The first question above might be answered with a simple linear regression model, the second, a more elaborate model that allows the relationship between experience and wages to vary across demographic groups; the final question might involve modeling the variance of the wage distribution, not just its mean.

The example given below introduces a simple linear model of wages given demographic characteristics, with the intent of introducing instrumental variables—the first trick up our sleeve for the day. We'll introduce two competing instrumental variables models: the first assuming independence between the first and second stage regressions and the second modeling them jointly.

Let's walk through each step of the workflow, gradually introducing Stan along the way. While we're not going to estimate the model on real data, we want to make sure that the model we build is sane. As such we'll look at the characteristics of wages for some real data. This data comes from some wage and demographics data from the 1988 Current Population Survey, which comes in R's `AER` package. This dataset contains the weekly wage for around 28,000 working men in 1988; prices are in 1992 dollars. You can load the dataset into your R workspace like so:

```
library(AER)
```

```
## Loading required package: car
```

```
## Loading required package: lmtest
```

```
## Loading required package: zoo
```

```
##
## Attaching package: 'zoo'
```

```
## The following objects are masked from 'package:base':
##
##     as.Date, as.Date.numeric
```

```
## Loading required package: sandwich
```

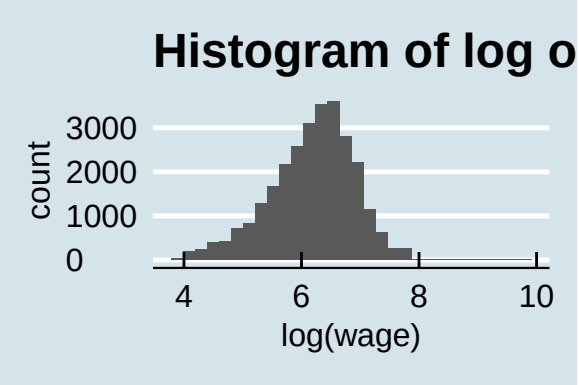
```
## Loading required package: survival
```

```
data("CPS1988")
```

2.1.2 Step 1: Writing out the probability model

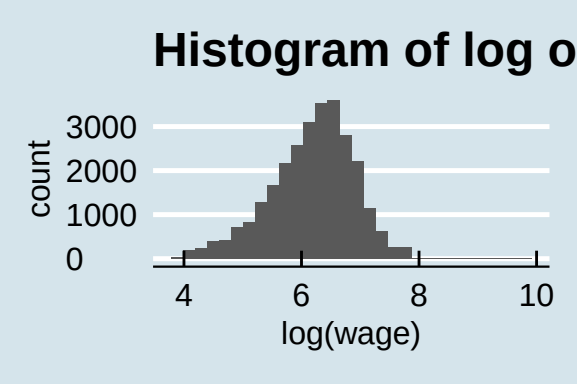
The first step of of our workflow is to propose an underlying generative model. It's helpful to think of a generative model as being a structured random number generator, which when simulated, generates outcomes with a distribution that looks like the distribution of the outcome variable. Once we have decided on the generative model, we then get into the specifics of endogeneity issues etc. In deciding the choice of distribution to use, you should plot a histogram or density of the outcome. For example, we could generate a histogram of wages like so:

```
library(ggplot2)
ggplot(CPS1988, aes(x = log(wage))) +
  geom_histogram() +
  ggthemes::theme_economist(base_size = 12) +
  ggtitle("Histogram of log of wages")
```



As we can see, the distribution of wages is quite skewed, and so we might need to choose a distribution capable of generating highly skewed outcomes. Another approach is to transform the data. In this case, because all wages are positive, we could take their natural log. The distribution of log wages appears to be far more normal than the initial distribution, and it possible that the non-normality is explainable using demographic characteristics.

```
ggplot(CPS1988, aes(x = log(wage))) +
  geom_histogram() +
  ggthemes::theme_economist(base_size = 12) +
  ggtitle("Histogram of log of wages")
```



If we decide to choose a normal density as the data-generating process, and assume that the conditional distribution of one person's wage does not depend on the conditional distribution of another person's, we can write it out like so:

$$\log(\text{wage})_i \sim \text{Normal}(\mu_i, \sigma_i)$$

which says that a person i's wage is distributed according to a normal distribution with *location* μ_i and *scale* σ_i. In the case of a normal density, the location is the mean, and the scale is the standard deviation. We prefer to use "location" and "scale" rather than "mean" and "standard deviation" because the terminology can carry across to other densities whose location and scale parameters don't correspond to the mean or standard deviation.

Let's be clear about what this means. This generative model says that each individual's (log) wage is not completely determined—it involves some amount of luck. So while on average it will be μ_i, luck will result in differences from this average, and these differences have a standard deviation of σ_i.

Notice that both parameters μ_i and σ_i vary across each individual. One of the main challenges of building a good model is to come up with functional forms for μ_i and σ_i, taking into account the information available to us. For instance, the (normal) linear regression model uses a (row) vector of individual characteristics

$X_i = (\text{education}_i, \text{experience}_i, \dots)$, along with a set of parameters that are common to all individuals (an intercept α, coefficients β and a scale parameter σ). The generative model is then:

$$\log(\text{wage})_i \sim \text{Normal}(\alpha + X_i\beta, \sigma)$$

which is the same as saying:

$$\log(\text{wage})_i = \alpha + X_i\beta + \epsilon_i \text{ with } \epsilon_i \sim \text{N}(0, \sigma)$$

Note that we've made "modeling assumptions" $\mu_i = \alpha + X_i\beta$ and $\sigma_i = \sigma$. The parameters of the generative model are both "true" and unknown. The entire point is to peform inference in order to get probabilistic estimates of the "true" parameters.

2.1.2.1 Choosing the right generative model

Above, we picked out a normal density for log wages (which corresponds to a lognormal density for wages) as a reasonable first step in modeling our wage series. How did we get to this choice? The choice of distribution to use should depend on the nature of your outcome variables. Two good rules of thumb are:

1. The chosen distribution should not give positive probability to impossible outcomes. For example, wages can't be negative, and so if we were to use a normal density (which gives positive probability to all outcomes) to model wages, we would be committing an error. If an outcome is outcome is binary or count data, the model should not give weight to non-integer outcomes. And so on.
2. The chosen distribution should give positive weight to plausible outcomes.

2.1.2.2 Choosing priors

To complete our probability model, we need to specify priors for the parameters β and σ. Again, these priors should place positive probabilistic weight over values of the parameters that we consider possible, and zero weight on impossible values (like a negative scale σ). In this case, it is common to assume normal priors for regression coefficients and half-Cauchy or half-Student-t priors on scales.

A great discussion of choosing priors is available here.

2.1.2.3 Thinking ahead: are our data endogenous? Instrumental variables

As you will see in the generative model above, ϵ are as though they've been drawn from a (normal) random number generator, and have no systematic relationship to the variables in X. Now what is the economic meaning of ϵ? The way I prefer to think about it is as a catch-all containing the unobserved information that is relevant to the outcome.

We need to think ahead: is there unobserved information that will be systematically correlated with X? Can we tell a story that there are things that cause both some change in one of our X variables and also our observed wages? If such information exists, then at the model estimation stage we will have an unobserved confounder problem, and we need to consider it in our probability model. A common way of achiving this is to use instrumental variables.

An instrumental variable is one that introduces plausible exogenous variation into our endogenous regressor. For example, if we have years of education on the right hand side, we might be concerned that the same sorts of unobserved factors—family and peer pressure, IQ etc.—that lead to high levels of education might also lead to high wages (even in absense of high levels of education). In this case we would want to "instrument" education, ideally with an experimental treatment that randomly assigned some people to higher rates of education and others to less. In reality, such an experiment might not be possible to run, but we might find "natural experiments" that result in the same variation. The most famous case of such an instrument is the Vietnam war draft (Angrist and Kreuger, 1992).

There are a few ways of incorporating instrumental variables. The first is so-called "two stage least squares" in which we first regress the endogenous regressor on the exogenous regressors ($X_{edu,i}$) plus an instrument or instruments Z_i. In the second stage we replace the actual values of education with the fitted values from the first stage.

Stage one:

$$\text{education}_i \sim \text{Normal}(\alpha_{s1} + X_{-edu,i}\gamma + Z_i\delta, \sigma_{s1})$$

Stage two:

$$\log(wage_i) \sim \text{Normal}(\alpha_{s2} + X_{-edu,i}\beta + (\alpha_{s1} + X_{-edu,i}\gamma + Z_i\delta)\tau, \sigma_{s2})$$

(In the second stage, we only estimate α_{s2}, β, τ and σ_{s2}; the other parameters' values are from the first stage).

If we treat the uncertainty of the first model approproately in the second (as is automatic in Bayes), then two stage least squares yields an consistent estimate of the treatment effect τ (that is, as the number of observations grows, we get less bias). But it may be inefficient in the case when the residuals of the data generating processes in stage one and stage two are correlated.

The second method of implementing instrumental variables is as a simultaneous equations model. Under this framework, the generative model is

$$(\log(wage_i), \text{edu}_i)' \sim \text{Multi normal}\left((\mu_{1,i}, \mu_{2,i})', \Sigma\right)$$

where

$$\mu_{1,i} = \alpha_{s2} + X_{-edu,i}\beta + (\alpha_{s1} + X_{-edu,i}\gamma + Z_i\delta)\tau$$

and

$$\mu_{2,i} = \alpha_{s1} + X_{-edu,i}\gamma + Z_i\delta$$

You will see: this is the same as the two stage least squares model above, exept we have allowed the errors to be correlated across equations (this information is in the covariance matrix Σ). Nobody really understands raw numbers from Covariance matrices, so we typically decompose covariance into the more interpretable scale vector σ and correlation matrix Ω such that $\Sigma = \text{diag}(\sigma)\Omega\text{diag}(\sigma)$. This decomposition also allows us to use more interpretable priors.

We now have two possible models. What we'll do below is simulate data from the second model with known parmaters. Then we'll code up both models and estimate each, allowing us to perform model comparison.

2.1.3 Step 2: Simulating the model with known parameters

We have now specified two probability models. What we will do next is simulate some data from the second (more complex model), and then check to see if we can recover the (known) model parameters by estimating both the correctly specified and incorrectly specified models above. Simulating and recovering known parameters is an important checking procedure in model building; it often helps catch errors in the model and clarifies the model in the mind of the modeler.

Now that we have written out the data generating model, let's generate some known parameters and covariates and simulate the model. First: generate some values for the data and paramaters.

```
# Generate a matrix of random numbers, and values for beta, nu and sigma

set.seed(48) # Set the random number generator seed so that we get the same parameters
N <- 500 # Number of observations
P <- 5 # Number of covariates
X <- matrix(rnorm(N*P), N, P) # generate an N*P covariate matrix of random data
```

```
Z <- rnorm(N) # an instrument

# The parameters governing the residuals
sigma <- c(1, 2)
Omega <- matrix(c(1, .5, .5, 1), 2, 2)

# Generate some residuals
resid <- MASS::mvrnorm(N, mu = c(0, 0), Sigma = diag(sigma)%*% Omega %*% diag(sigma))

# Now the parameters of our model
beta <- rnorm(P)
tau <- 1 # This is the treatment effect we're looking to recover
alpha_1 <- rnorm(1)
alpha_2 <- rnorm(1)
gamma <- rnorm(P)
delta <- rnorm(1)

mu_2 <- alpha_1 + X%*%gamma + Z*delta
mu_1 <- alpha_2 + X%*%beta + mu_2*tau

Y <- as.numeric(mu_1 + resid[,1])
endog_regressor <- as.numeric(mu_2 + resid[,2])

# And let's check we can't recapture with simple OLS:

lm(Y ~ . + endog_regressor, data = as.data.frame(X))
```

2.1.4 Writing out the Stan model to recover known parameters

A Stan model is comprised of code blocks. Each block is a place for a certain task. The bold blocks below must be present in all Stan programs (even if they contain no arguments):

1. `functions`, where we define functions to be used in the blocks below. This is where we will write out a random number generator that gives us draws from our assumed model.
2. `data`, declares the data to be used for the model
3. `transformed data`, makes transformations of the data passed in above
4. `parameters`, defines the unknowns to be estimated, including any restrictions on their values.
5. `transformed parameters`, often it is preferable to work with transformations of the parameters and data declared above; in this case we define them here.
6. `model`, where the full probability model is defined.
7. `generated quantities`, generates a range of outputs from the model (posterior predictions, forecasts, values of loss functions, etc.).

```
# In R:
# Load necessary libraries and set up multi-core processing for Stan
options(warn=-1, message =-1)
library(dplyr); library(ggplot2); library(rstan); library(reshape2)
options(mc.cores = parallel::detectCores())
```

Now we have y and X, and we want to estimate β, σ and, depending on the model, ν. We have two candidate probability models that we want to estimate and check which one is a more plausible model of the data. To do this, we need to specify both models in Stan and then estimate them.

Let's jump straight in and define the incorrectly specified model. It is incorrect in that we haven't properly accounted for the mutual information in first and second stage regressions.

```
// saved as models/independent_iv.stan
// saved as models/independent_iv.stan
data {
  int N; // number of observations
  int P; // number of covariates
  matrix[N, P] X; //covariate matrix
  vector[N] Y; //outcome vector
  vector[N] endog_regressor; // the endogenous regressor
  vector[N] Z; // the instrument (which we'll assume is a vector)
}
parameters {
  vector[P] beta; // the regression coefficients
  vector[P] gamma;
  real tau;
  real delta;
  real alpha_1;
  real alpha_2;
  vector<lower = 0>[2] sigma; // the residual standard deviation
  corr_matrix[2] Omega;
}
transformed parameters {
  matrix[N, 2] mu;

  for(i in 1:N) {
    mu[i,2] = alpha_1 + X[i]*gamma + Z[i]*delta;
    mu[i,1] = alpha_2 + X[i]*beta + mu[i,2]*tau;
  }
}
model {
  // Define the priors
  beta ~ normal(0, 1);
  gamma ~ normal(0, 1);
  tau ~ normal(0, 1);
  sigma ~ cauchy(0, 1);
  delta ~ normal(0, 1);
  alpha_1 ~ normal(0, 1);
  alpha_2 ~ normal(0, 2);
  Omega ~ lkj_corr(5);

  // The likelihood
  for(i in 1:N) {
    Y[i]~ normal(mu[i], sigma[1]);
    endog_regressor[i]~ normal(mu[2], sigma[2]);
  }


}
generated quantities {
  // For model comparison, we'll want to keep the likelihood
  // contribution of each point

  vector[N] log_lik;
  for(i in 1:N) {
   log_lik[i] = normal_lpdf(Y[i] | alpha_1  + X[i,] * beta + endog_regressor[i]*tau, sigma[1]);
```

```
  }
}
```

Now we define the correctly specified model. It is the same as above, but with a couple of changes:

```
// saved as models/joint_iv.stan
// saved as models/joint_iv.stan
data {
  int N; // number of observations
  int P; // number of covariates
  matrix[N, P] X; //covariate matrix
  vector[N] Y; //outcome vector
  vector[N] endog_regressor; // the endogenous regressor
  vector[N] Z; // the instrument (which we'll assume is a vector)
}
parameters {
  vector[P] beta; // the regression coefficients
  vector[P] gamma;
  real tau;
  real delta;
  real alpha_1;
  real alpha_2;
  vector<lower = 0>[2] sigma; // the residual standard deviation
  corr_matrix[2] Omega;
}
transformed parameters {
  matrix[N, 2] mu;

  for(i in 1:N) {
    mu[i,2] = alpha_1 + X[i]*gamma + Z[i]*delta;
    mu[i,1] = alpha_2 + X[i]*beta + mu[i,2]*tau;
  }
}
model {
  // Define the priors
  beta ~ normal(0, 1);
  gamma ~ normal(0, 1);
  tau ~ normal(0, 1);
  sigma ~ cauchy(0, 1);
  delta ~ normal(0, 1);
  alpha_1 ~ normal(0, 1);
  alpha_2 ~ normal(0, 2);
  Omega ~ lkj_corr(5);

  // The likelihood
  {
    matrix[N, 2] Y2;
    Y2 = append_col(Y, endog_regressor);
    for(i in 1:N) {
      Y2[i]~ multi_normal(mu[i], diag_matrix(sigma)*Omega*diag_matrix(sigma));
    }
  }

}
generated quantities {
```

```
  // For model comparison, we'll want to keep the likelihood
  // contribution of each point

  vector[N] log_lik;
  for(i in 1:N) {
   log_lik[i] = normal_lpdf(Y[i] | alpha_1  + X[i,] * beta + endog_regressor[i]*tau, sigma[1]);
  }
}
```

Now that we have specified two models, let's estimate them with the y and X we generated above.

```
# In R

compiled_model <- stan_model("")

incorrect_fit <- stan(file = "models/independent_iv.stan",
                      data = list(Y = Y,
                                  X = X,
                                  endog_regressor = endog_regressor,
                                  P = P,
                                  N = N,
                                  Z = Z),
                      iter = 600)

correct_fit <- stan(model_code = "models/joint_iv.stan",
                    data = list(Y = Y,
                                X = X,
                                endog_regressor = endog_regressor,
                                P = P,
                                N = N,
                                Z = Z),
                    iter = 600)
```

We have now fit our two competing models to the data. What has been estimated?

2.1.4.1 What do these fitted objects contain?

If you are accustomed to estimating models using ordinary least squares (OLS), maximum likelihood estimates (MLE), or the general method of moments (GMM), then you may expect point estimates for parameters: regression tables contain an estimate of the parameter along with some standard errors. Full Bayesian inference involves averaging over the uncertainty in parameter estimates, that is, the posterior distribution. For a point estimate, Bayesians typically use the mean of the posterior distribution, because it minimizes expected square error in the estimate; the posterior median minimizes expected absolute error.

For all but a few models, posterior distributions cannot be expressed analytically. Instead, numerical techniques involving simulation going under the general heading of Monte Carlo methods, are used to estimate quantities of interest by taking draws from the distribution in question.

Monte Carlo estimation is quite simple. Let's say a parameter θ is distributed according to some distribution Foo(θ) for which we have no analytical formula, but from which we can simulate random draws. We want to draw statistical inferences using this distribution; we want its mean (expected value), standard deviation, median and other quantiles for posterior intervals, etc. The Monte Carlo method allows us to make these inferences by simply generating many (not necessarily independent) draws from the distribution and then calculating the statistic of interest from those draws. Because these draws are from the distribution of interest, they will tend to come from the higher probability regions of the distribution. For example, if 50%

of the posterior probability mass is near the posterior mode, then 50% of the simulated draws (give or take sampling error) should be near the posterior mode.

For example, suppose we want to estimate the expectation of Foo(θ), or in other words, the mean of a variable θ with distribution Foo(θ). If we take M random draws from Foo,

$$\theta^{(1)}, \ldots, \theta^{(M)} \sim \text{Foo}(),$$

then we can estimate the expected value of θ (i.e., its posterior mean) as

$$\mathbb{E}[\theta] \approx \frac{1}{M} \sum_{m=1}^{M} \theta^{(m)}.$$

If the draws $\theta^{(m)}$ are independent, the result is a sequence of independent and identically distributed (i.i.d.) draws. The mean of a sequence of i.i.d. draws is governed by the central limit theorem, where the standard error on the estimates is given by the standard deviation divided by the square root of the number of draws. Thus standard error decreases as $\mathcal{O}(\frac{1}{\sqrt{M}})$ in the number of independent draws M.

What makes Bayesian inference not only possible, but practical, is that almost all of the Bayesian inference for event probabilities, predictions, and parameter estimates can be expressed as expectations and carried out using Monte Carlo methods.

There is one hitch, though. For almost any practically useful model, not only will we not be able to get an analytical formula for the posterior, we will not be able to take independent draws. Fortunately, all is not lost, as we will be able to take identically distributed draws using a technique known as Markov chain Monte Carlo (MCMC). With MCMC, the draws from a Markov chain in which each draw $\theta^{(m+1)}$ depends (only) on the previous draw $\theta^{(m)}$. Such draws are governed by the MCMC central limit theorem, wherein a quantity known as the effective sample size plays the role of the effective sample size in pure Monte Carlo estimation. The effective sample size is determined by how autocorrelated the draws are; if each draw is highly correlated with previous draws, then more draws are required to achieve the same effective sample size.

Stan is able to calculate the effective sample size for its MCMC methods and use that to estimate standard errors for all of its predictive quantities, such as parameter and event probability estimates.

A fitted Stan object contains a sequence of M draws, where each draw contains a value for every parameter (and generated quantity) in the model. If the computation has converged, as measured by built-in convergence diagnostics, the draws are from the posterior distribution of our parameters conditioned on the observed data. These are draws from the joint posterior distribution; correlation between parameters is likely to be present in the joint posterior even if it was not present in the priors.

In the generated quantities block of the two models above, we declare variables for two additional quantities of interest.

- The first, `log_lik`, is the log-likelihood, which we use for model comparison. We obtain this value for each parameter draw, for each value of y_i. Thus if you have N observations and `iter` parameter draws, this will contain $N \times$ `iter` log-likelihood values (which may produce a lot of output for large data sets).
- The second, `y_sim`, is a *posterior predictive quantity*, in this case a replicated data set consisting of a sequence of fresh outcomes generated randomly from the parameters. Rather than each observation having a "predicted value", it has a predictive distribution that takes into account both the regression residual and uncertainty in the parameter estimates.

2.1.5 Model inspection

To address questions 1 and 2 above, we need to examine the parameter draws from the model to check for a few common problems:

- **Lack of mixing**. A poorly "mixing" Markov chain is one that moves very slowly between regions of the parameter space or barely moves at all. This can happen if the distribution of proposals is much narrower than the target (posterior) distribution or if it is much wider than the target distribution. In the former case most proposals will be accepted but the Markov chain will not explore the full parameter space whereas in the latter case most proposals will be rejected and the chain will stall. By running several Markov chains from different starting values we can see if each chain mixes well and if the chains are converging on a common distribution. If the chains don't mix well then it's unlikely we're sampling from a well specified posterior. The most common reason for this error is a poorly specified model.
- **Stationarity**. Markov chains should be covariance stationary, which means that the mean and variance of the chain should not depend on when you draw the observations. Non-stationarity is normally the consequence of a poorly specified model or an insufficient number of iterations.
- **Autocorrelation**. Especially in poorly specified or weakly identified models, a given draw of parameters can be highly dependent on the previous draw of the parameters. One consequence of autocorrelation is that the posterior draws will contain less information than the number of draws suggests. That is, the effective posterior sample size will be much less than the actual posterior sample size. For example, 2000 draws with high autocorrelation will be less informative than 2000 independent draws. Assuming the model is specified correctly, then *thinning* (keeping only every k-th draw) is one common approach to dealing with highly autocorrelated draws. However, while thinning can reduce the autocorrelation in the draws that are retained it still sacrifices information. If possible, reparameterising the model is a better approach to this problem. (See section 21 of the manual, on Optimizing Stan code).
- **Divergent transitions**. In models with very curved or irregular posterior densities, we often get "divergent transitions". This typically indicates that the sampler was unable to explore certain regions of the distribution and a respecification or changes to the sampling routine may be required. The easiest way of addressing this issue is to use `control = list(adapt_delta = 0.99)` or some other number close to 1. This will lead to smaller step sizes and therefore more steps will be required to explore the posterior. Sampling will be slower but the algorithm will often be better able to explore these problematic regions, reducing the number of divergent transitions.

All of these potential problems can be checked using the ShinyStan graphical interface, which is available in the `shinystan R` package. You can install it with `install.packages("shinystan")`, and run it with `launch_shinystan(correct_fit)`. It will bring up an interactive session in your web browser within which you can explore the estimated parameters, examine the individual Markov chains, and check various diagnostics. More information on ShinyStan is available here. We will confront most of these issues and show how to resolve them in later chapters when we work with real examples. For now just keep in mind that MCMC samples always need to be checked before they are used for making inferences.

2.1.6 Model comparison

Let's start by looking at the model outputs. The draws from each parameter can be neatly summarized with `print`:

```
# In R:

print(incorrect_fit, pars = c("beta", "tau", "sigma"))
# specify parameters to save; else we'd get `log_lik` and `y_sim`

# Some things to note:

# - mean is the mean of the draws for each observation
# - se_mean is the Monte Carlo error
#   (standard error of the Monte Carlo estimate from the true mean)
# - sd is the standard deviation of the parameter's draws
```

```
# - the quantiles are self-explanatory
# - n_eff is the effective number of independent draws.
#   If there is serial correlation between sequential draws,
#   the draws cannot be considered independent.
#   In Stan, high serial correlation is typically a problem in
#   poorly specified models
# - Rhat: this is the Gelman Rubin convergence diagnostic.
#   Values close to 1 indicate that the multiple chains
#   that you estimated have converged to the same
#   distribution and are "mixing" well.
```

```
# In R

print(correct_fit, pars = c("beta", "sigma", "nu"))
```

At the moment, it seems as though both our models have done about as good a job at estimating the regression coefficients β as one another. But the incorrectly specified model severely overestimates σ. This makes sense–a Student-t distribution with $\nu = 5$ will have fat tails, and so a normal distribution will try to replicate the extreme values by having a large variance.

How else might we compare the two models?

One approach is to use the `loo` package to compare the models on their estimated out-of-sample predictive performance. The idea of this package is to approximate each model's leave-one-out (LOO) cross-validation error, allowing model comparison by the LOO Information Criterion (LOOIC). LOOIC has the same purpose as the Akaike Information Criterion (AIC), which is to estimate the expected log predictive density (ELPD) for a new dataset. However, AIC ignores prior distributions and makes the assumption that the posterior is a multivariate normal distribution. The approach taken by the `loo` package does not make this distributional assumption and also integrates over (averages over) the uncertainty in the parameters.

The Bayesian LOO estimate is $\sum_{n=1}^{N} \log p(y_n \mid y_1, ..., y_{n-1}, y_{n+1}, ..., y_N)$, which requires fitting the model N times, each time leaving out one of the N data points. For large datasets or complex models the computational cost is usually prohibitive. The `loo` package does an approximation that avoids re-estimating the model and requires only the log-likelihood evaluated at the posterior draws of the parameters. The approximation will be good so long as the posterior distribution is not very sensitive to leaving out any single observation.

A big upside of this approach is that it enables us to generate probabilistic estimates of the degree to which each model is most likely to produce the best out-of-sample predictions.

We use `loo` like so:

```
# in R
#
# library(loo) # Load the library
#
# # Extract the log likelihoods of both models.
# # Note that we need to declare log_lik in the generated quantities block
llik_incorrect <- extract_log_lik(incorrect_fit, parameter_name = "log_lik")
llik_correct <- extract_log_lik(correct_fit, parameter_name = "log_lik")
#
# # Estimate the leave-one-out cross validation error
loo_incorrect <- loo(llik_incorrect)
loo_correct <- loo(llik_correct)
```

```
# # Print the LOO statistics
print("Incorrect model")
print(loo_incorrect)
```

```
print("Correct model")
print(loo_correct)
```

The quantity `elpd_loo` is the expected log pointwise predictive density (ELPD). The log pointwise predictive density is easiest to understand in terms of its computation. For each data point we compute the log of its average likelihood, where the average is computed over the posterior draws. Then we take the sum over all of the data points. We can multiply `elpd_loo` by -2 to calculate the `looic`, which you can think of like AIC or BIC, but coming from our Bayesian framework. The -2 is not important; it simply converts the value to the so-called deviance scale. The value of `p_loo` is the estimated effective number of parameters, which is a measure of model complexity. The effective number of parameters can be substantially less than the actual number of parameters when there is strong dependence between parameters (e.g. in many hierarchical models) or when parameters are given informative prior distributions. For further details on these quantities, please consult this paper.

```
# Print the comparison between the two models
print(compare(loo_incorrect, loo_correct), digits = 2)
```

When using the `compare` function to compare two models the `elpd_diff` gives us the difference in the ELPD estimates for the models. A positive `elpd_diff` indicates that the second model is estimated to have better out-of-sample predictive accuracy than the first, which is precisely what we expect in this case. When comparing more than two models the `compare` function will order the models by their ELPD estimates.

2.2 Tools of the trade: borrowing from software engineering

Building economic and statistical models increasingly requires sophisticated computation. This has the potential to improve our modeling, but carries with it risks; as the complexity of our models grows, so too does the prospect of making potentially influential mistakes. The well-known spreadsheet error in Rogoff and Reinhart's (Cite) paper—a fairly simple error in very public paper—was discovered. Who knows how many other errors exist in more complex, less scruitinized work?

Given the ease of making errors that substantively affect our models' outputs, it makes sense to adopt a workflow that minimizes the risk of such error happening. The set of tools discussed in this section, all borrowed from software engineering, are designed for this purpose. We suggest incorporating the following into your workflow:

- Document your code formally. At the very least, this will involve commenting your code to the extend where a colleague could read it and not have too many questions. Ideally it will include formal documentation of every function that you write.
- When you write functions, obey what we might call "Tinbergen's rule of writing software": *one function, one objective*. Try not to write omnibus functions that conduct a large part of your analysis. Writing small, modular functions will allow you to use **unit testing**, a framework that lets you run a set of tests automatically, ensuring that changing one part of your code base does not break other parts.
- Use Git to manage your workflow. Git is a very powerful tool that serves several purposes. It can help you back up your work, which is handy. It also allows you to view your codebase at periods when you *committed* some code to the code base. It lets you experiment on *branches*, without risking the main ("production") code base. Finally helps you work in teams; formalizing a **code-review** procedure that should help catch errors.

Chapter 3

A more difficult model

3.1 This session

In this session, we'll cover two of the things that Stan lets you do quite simply: implement state space models, and finite mixtures.

3.1.1 Finite mixtures

In a post here, I describe a simple model in which each observation of our data could have one of two densities. We estimated the parameters of both densities, and the probability of the data coming from either. While finite mixture models as in the last post are a useful learning aid, we might want richer models for applied work. In particular, we might want the probability of our data having each density to vary across observations. This is the first of two posts dedicated to this topic. I gave a talk covering some of this also (best viewed in Safari).

For sake of an example, consider this: the daily returns series of a stock has two states. In the first, the stock is 'priced to perfection', and so the price is an I(1) random walk (daily returns are mean stationary). In the second, there is momentum—here, daily returns have AR(1) structure. Explicitly, for daily log returns r_t:

State 1: $r_t \sim \text{normal}(\alpha_1, \sigma_1)$

State 2: $r_t \sim \text{normal}(\alpha_2 + \rho_1 r_{t-1}, \sigma_2)$

When we observe a value of r_t, we don't know for sure whether it came from the first or second model–that is precisely what we want to infer. For this, we need a model for the probability that an observation came from each state $s_t \in 1, 2$. One such model could be:

$$\text{prob}(s_t = 1 | \mathcal{I}_t) = \text{Logit}^{-1}(\mu_t)$$

with

$$\mu_t \sim \text{normal}(\alpha_3 + \rho_2 \mu_{t-1} + f(\mathcal{I}_t), \sigma_3)$$

Here, $f(\mathcal{I}_t)$ is a function of the information available at the beginning of day t. If we had interesting information about sentiment, or news etc., it could go in here. For simplicity, let's say $f(\mathcal{I}_t) = \beta r_{t-1}$.

Under this specification (and for a vector containing all parameters, θ), we can specify the likelihood contribution of an observation. It is simply the weighted average of likelihoods under each candidate data generating process, where the weights are the probabilities that the data comes from each density.

$$p(r_t|\theta) = \text{Logit}^{-1}(\mu_t)\,\text{normal}(r_t|\,\alpha_1, \sigma_1) + (1 - \text{Logit}^{-1}(\mu_t))\,\text{normal}(r_t|\,\alpha_2 + \rho r_{t-1}, \sigma_2)$$

As discussed in the last post, we work in log likelihoods, not likelihoods. This means we should use the `log_sum_exp()` function in Stan. This means that we express the log likelihood contribution of a single point as:

```
log_sum_exp(log(inv_logit(mu[t])) + normal_lpdf(r[t] | alpha[1], sigma[1]),
            log((1 - inv_logit(mu[t]))) + normal_lpdf(r[t] | alpha[2] + rho[1], sigma[2]))
```

Stan has recently added another function which performs the same calculation, but makes writing it out a bit easier. For two log densities `lp1`, `lp2` and a mixing probability `theta`, we have

```
log_mix(theta, lp1, lp2) = log_sum_exp(log(theta) + lp1,
                                       log(1-theta) + lp2)
```

3.1.2 Writing out the model

The Stan code for the model is:

```
// saved as time_varying_finite_mixtures.stan
data {
  int T;
  vector[T] r;
}
parameters {
  vector[T] mu;
  vector[2] rho;
  real beta;
  vector<lower = 0>[3] sigma;
  vector[3] alpha;
}
model {
  // priors
  mu[1] ~ normal(0, .1);
  sigma ~ cauchy(0, 0.5);
  rho ~ normal(1, .1);
  beta~ normal(.5, .25);
  alpha[1:2] ~ normal(0, 0.1);
  alpha[3] ~ normal(0, 1);

  // likelihood
  for(t in 2:T) {
    mu[t] ~ normal(alpha[3] + rho[1]*mu[t-1] + beta* r[t-1], sigma[3]);

    target += log_mix(inv_logit(mu[t]),
                      normal_lpdf(r[t] | alpha[1], sigma[1]),
                      normal_lpdf(r[t] | alpha[2] + rho[2] * r[t-1], sigma[2]));
  }
}
```

3.1.3 Recapturing 'known unknowns'

As should be clear by now, I believe strongly that we should simulate from the model and make sure that we can recapture "known unknowns" before taking the model to real data. Below we simulate some fake data.

```
# Set some fake parameters
alpha1 <- -0.01
alpha2 <- 0.015
rho1 <- 0.95
rho2 <- 0.8
beta <- 0.5

sigma1 <- 0.05
sigma2 <- 0.03
sigma3 <- 0.3
T <- 500
r <- rep(NA, T)
r[1] <- 0

mu <- rep(NA, T)
z <- rep(NA, T)
mu[1] <- 0
z[1] <- 1


# Simulate the data series
for(t in 2:T) {
  mu[t]  <- rho1 * mu[t-1] + beta*(r[t-1]) + rnorm(1, 0, sigma3)
  prob <- arm::invlogit(mu[t])
  z[t] <- sample(1:2, 1, prob = c(prob, 1-prob))

  if(z[t]==1) {
    # random walk state
    r[t] <- rnorm(1, alpha1, sigma1)
  } else {
    # momentum state
    r[t] <- rnorm(1, alpha2 + rho2*r[t-1], sigma2)
  }
}
```

You should plot your data before doing anything. Let's take a look.

```
# Plot the returns
plot.ts(r)
# Plot the probability of the random walk state
plot.ts(arm::invlogit(mu))
```

Looks good! Now we compile and run the model.

```
compiled_model <- stan_model("time_varying_finite_mixtures.stan")

estimated_model <- sampling(compiled_model, data = list(r = r, T = T), cores = 4, chains = 4)
```

Now we inspect the parameter estimates, which should align with those in our data generating process.

```
print(estimated_model, pars = c("alpha", "rho", "sigma"))
```

It seems that most of the parameters appear to have estimated quite cleanly–most of the Rhats are fairly close, to 1, with the exception of the standard deviation of the updates in the latent series (which will be very weakly identified, given we don't observe `mu`). We would fix this by adding better prior information to the model.

3.1.4 Taking the model to real data

Now we know that our program can recapture a known model, we can take it to some real data. In this case, we'll use the log differences in sequential adjusted closing prices for Apple's common stock. With Apple being such a large, well-researched (and highly liquid) stock, we should expect that it spends almost all time in the random walk state. Let's see what the data say!

```
# Now with real data!
aapl <- Quandl::Quandl("YAHOO/AAPL")

aapl <- aapl %>%
  mutate(Date = as.Date(Date)) %>%
  arrange(Date) %>%
  mutate(l_ac = log(`Adjusted Close`),
         dl_ac = c(NA, diff(l_ac))) %>%
  filter(Date > "2015-01-01")

aapl_mod <- sampling(compiled_model, data= list(T = nrow(aapl), r = aapl$dl_ac*100))
```

Now check that the model has fit properly

```
shinystan::launch_shinystan(aapl_mod)
```

And finally plot the probability of being in each state.

```
plot1 <- aapl_mod %>%
  as.data.frame() %>%
  select(contains("mu")) %>%
  melt() %>%
  group_by(variable) %>%
  summarise(lower = quantile(value, 0.95),
            median = median(value),
            upper = quantile(value, 0.05)) %>%
  mutate(date = aapl$Date,
         ac = aapl$l_ac) %>%
  ggplot(aes(x = date)) +
  geom_ribbon(aes(ymin = arm::invlogit(lower), ymax = arm::invlogit(upper)), fill= "orange", alpha = 0.
  geom_line(aes(y = arm::invlogit(median))) +
  ggthemes::theme_economist() +
  xlab("Date") +
  ylab("Probability of random walk model")


plot2 <- aapl_mod %>%
  as.data.frame() %>%
  select(contains("mu")) %>%
  melt() %>%
  group_by(variable) %>%
  summarise(lower = quantile(value, 0.95),
            median = median(value),
```

```
            upper = quantile(value, 0.05)) %>%
  mutate(date = aapl$Date,
         ac = aapl$`Adjusted Close`) %>%
  ggplot(aes(x = date, y = ac)) +
  geom_line() +
  ggthemes::theme_economist() +
  xlab("Date") +
  ylab("Adjusted Close")

gridExtra::grid.arrange(plot1, plot2)
```

And there we go! As expected, Apple spends almost all their time in the random walk state, but, surprisingly, appears to have had a few periods with some genuine (mainly negative) momentum.

3.1.5 Building up the model

The main problem with this model is that our latent state μ can only really vary so much from period to period. That can delay the response to the appearance of a new state, and slow the process of "flipping back" into the regular state. One way of getting around this is to have a discrete state with more flexibility in flipping between states. We'll explore this in the next post, on Regime-Switching models.

3.2 A state space model involving polls

This tutorial covers how to build a low-to-high frequency interpolation model in which we have possibly many sources of information that occur at various frequencies. The example I'll use is drawing inference about the preference shares of Clinton and Trump in the current presidential campaign. This is a good example for this sort of imputation:

- Data (polls) are sporadically released. Sometimes we have many released simultaneously; at other times there may be many days with no releases.
- The various polls don't necessarily agree. They might have different methodologies or sampling issues, resulting in quite different outcomes. We want to build a model that can incorporate this.

There are two ingredients to the polling model. A multi-measurement model, typified by Rubin's 8 schools example. And a state-space model. Let's briefly describe these.

3.2.1 Multi-measurement model and the 8 schools example

Let's say we run a randomized control trial in 8 schools. Each school i reports its own treatment effect te_i, which has a standard error σ_i. There are two questions the 8-schools model tries to answer:

- If you administer the experiment at one of these schools, say, school 1, and have your estimate of the treatment effect te_1, what do you expect would be the treatment effect if you were to run the experiment again? In particular, would your expectations of the treatment effect in the next experiment change once you learn the treatment effects estimated from the experiments in the other schools?
- If you roll out the experiment at a new school (school 9), what do we expect the treatment effect to be?

The statistical model that Rubin proposed is that each school has its own *true* latent treatment effect y_i, around which our treatment effects are distributed.

$$te_i \sim \mathcal{N}(y_i, \sigma_i)$$

These "true" but unobserved treatment effects are in turn distributed according to a common hyper-distribution with mean μ and standard deviation τ

$$y_i \sim \mathcal{N}(\mu, \tau)$$

Once we have priors for μ and τ, we can estimate the above model with Bayesian methods.

3.2.2 A state-space model

State-space models are a useful way of dealing with noisy or incomplete data, like our polling data. The idea is that we can divide our model into two parts:

- **The state**. We don't observe the state; it is a latent variable. But we know how it changes through time (or at least how large its potential changes are).
- **The measurement**. Our state is measured with imprecision. The measurement model is the distribution of the data that we observe around the state.

A simple example might be consumer confidence, an unobservable latent construct about which our survey responses should be distributed. So our state-space model would be:

The state

$$conf_t \sim \mathcal{N}(conf_{t-1}, \sigma)$$

which simply says that consumer confidence is a random walk with normal innovations with a standard deviation σ, and

$$\text{survey measure}_t \sim \text{normal}(\text{conf}_t, \tau)$$

which says that our survey measures are normally distributed around the true latent state, with standard deviation τ.

Again, once we provide priors for the initial value of the state $conf_0$ and τ, we can estimate this model quite easily.

The important thing to note is that we have a model for the state even if there is no observed measurement. That is, we know (the distribution for) how consumer confidence should progress even for the periods in which there are no consumer confidence surveys. This makes state-space models ideal for data with irregular frequencies or missing data.

3.2.3 Putting it together

As you can see, these two models are very similar: they involve making inference about a latent quantity from noisy measurements. The first shows us how we can aggregate many noisy measurements together *within a single time period*, while the second shows us how to combine irregular noisy measures *over time*. We can now combine these two models to aggregate multiple polls over time.

The data generating process I had in mind is a very simple model where each candidate's preference share is an unobserved state, which polls try to measure. Unlike some volatile poll aggregators, I assume that the unobserved state can move according to a random walk with normal disturbances of standard deviation .25%. This greatly smoothes out the sorts of fluctuations we see around the conventions etc. We could estimate this parameter using fairly tight priors, but I just hard-code it in for simplicity.

That is, we have the state for candidate c in time t evolving according to

$$\text{Vote share}_{c,t} \sim \mathcal{N}(\text{Vote share}_{c,t-1}.0.25)$$

with measurements being made of this in the polls. Each poll p at time t is distributed according to

$$\text{poll}_{c,p,t} \sim \mathcal{N}(\text{Vote share}_{c,t}.\tau)$$

I give an initial state prior of 50% to Clinton and a 30% prior to Trump May of last year. As we get further from that initial period, the impact of the prior is dissipated.

The code to download the data, run the model is below. You will need to have the most recent version of ggplot2 installed.

```
// saved as models/state_space_polls.stan

data {
  int polls; // number of polls
  int T; // number of days
  matrix[T, polls] Y; // polls
  matrix[T, polls] sigma; // polls standard deviations
  real initial_prior;
}
parameters {
  vector[T] mu; // the mean of the polls
  real<lower = 0> tau; // the standard deviation of the random effects
  matrix[T, polls] shrunken_polls;
}
model {
  // prior on initial difference
  mu[1] ~ normal(initial_prior, 1);
  tau ~ student_t(4, 0, 5);
  // state model
  for(t in 2:T) {
    mu[t] ~ normal(mu[t-1], 0.25);
  }

  // measurement model
  for(t in 1:T) {
    for(p in 1:polls) {
      if(Y[t, p] != -9) {
        Y[t,p]~ normal(shrunken_polls[t, p], sigma[t,p]);
        shrunken_polls[t, p] ~ normal(mu[t], tau);
      } else {
        shrunken_polls[t, p] ~ normal(0, 1);
      }
    }
  }
}
```

```
library(rvest); library(dplyr); library(ggplot2); library(rstan); library(reshape2); library(stringr); l
options(mc.cores = parallel::detectCores())
source("models/theme.R")

# The polling data
realclearpolitics_all <- read_html("http://www.realclearpolitics.com/epolls/2016/president/us/general_el
```

```
# Scrape the data
polls <- realclearpolitics_all %>%
  html_node(xpath = '//*[@id="polling-data-full"]/table') %>%
  html_table() %>%
  filter(Poll != "RCP Average")

# Function to convert string dates to actual dates
get_first_date <- function(x){
  last_year <- cumsum(x=="12/22 - 12/23")>0
  dates <- str_split(x, " - ")
  dates <- lapply(1:length(dates), function(x) as.Date(paste0(dates[[x]],
                                                              ifelse(last_year[x], "/2015", "/2016")),
                                                       format = "%m/%d/%Y"))
  first_date <- lapply(dates, function(x) x[1]) %>% unlist
  second_date <- lapply(dates, function(x) x[2])%>% unlist
  data_frame(first_date = as.Date(first_date, origin = "1970-01-01"),
             second_date = as.Date(second_date, origin = "1970-01-01"))
}

# Convert dates to dates, impute MoE for missing polls with average of non-missing,
# and convert MoE to standard deviation (assuming MoE is the full 95% one sided interval length??)
polls <- polls %>%
  mutate(start_date = get_first_date(Date)[[1]],
         end_date = get_first_date(Date)[[2]],
         N = as.numeric(gsub("[A-Z]*", "", Sample)),
         MoE = as.numeric(MoE))%>%
  select(end_date, `Clinton (D)`, `Trump (R)`, MoE) %>%
  mutate(MoE = ifelse(is.na(MoE), mean(MoE, na.rm = T), MoE),
         sigma = MoE/2) %>%
  arrange(end_date) %>%
  filter(!is.na(end_date))


# Stretch out to get missing values for days with no polls
polls3 <- left_join(data_frame(end_date = seq(from = min(polls$end_date),
                                              to= as.Date("2016-08-04"),
                                              by = "day")), polls) %>%
  group_by(end_date) %>%
  mutate(N = 1:n()) %>%
  rename(Clinton = `Clinton (D)`,
         Trump = `Trump (R)`)


# One row for each day, one column for each poll on that day, -9 for missing values
Y_clinton <- polls3 %>% dcast(end_date ~ N, value.var = "Clinton") %>%
  dplyr::select(-end_date) %>%
  as.data.frame %>% as.matrix
Y_clinton[is.na(Y_clinton)] <- -9

Y_trump <- polls3 %>% dcast(end_date ~ N, value.var = "Trump") %>%
  dplyr::select(-end_date) %>%
  as.data.frame %>% as.matrix
Y_trump[is.na(Y_trump)] <- -9
```

```
# Do the same for margin of errors for those polls
sigma <- polls3 %>% dcast(end_date ~ N, value.var = "sigma")%>%
  dplyr::select(-end_date)%>%
  as.data.frame %>% as.matrix
sigma[is.na(sigma)] <- -9

# Run the two models

clinton_model <- stan("models/state_space_polls.stan",
                      data = list(T = nrow(Y_clinton),
                                  polls = ncol(Y_clinton),
                                  Y = Y_clinton,
                                  sigma = sigma,
                                  initial_prior = 50), iter = 600)


trump_model <- stan("models/state_space_polls.stan",
                    data = list(T = nrow(Y_trump),
                                polls = ncol(Y_trump),
                                Y = Y_trump,
                                sigma = sigma,
                                initial_prior = 30), iter = 600)



# Pull the state vectors

mu_clinton <- extract(clinton_model, pars = "mu", permuted = T)[[1]] %>%
  as.data.frame

mu_trump <- extract(trump_model, pars = "mu", permuted = T)[[1]] %>%
  as.data.frame

# Rename to get dates
names(mu_clinton) <- unique(paste0(polls3$end_date))
names(mu_trump) <- unique(paste0(polls3$end_date))


# summarise uncertainty for each date

mu_ts_clinton <- mu_clinton %>% melt %>%
  mutate(date = as.Date(variable)) %>%
  group_by(date) %>%
  summarise(median = median(value),
            lower = quantile(value, 0.025),
            upper = quantile(value, 0.975),
            candidate = "Clinton")

mu_ts_trump <- mu_trump %>% melt %>%
  mutate(date = as.Date(variable)) %>%
  group_by(date) %>%
  summarise(median = median(value),
            lower = quantile(value, 0.025),
```

```
            upper = quantile(value, 0.975),
            candidate = "Trump")

# Plot results


bind_rows(mu_ts_clinton, mu_ts_trump) %>%
  ggplot(aes(x = date)) +
  geom_ribbon(aes(ymin = lower, ymax = upper, fill = candidate),alpha = 0.1) +
  geom_line(aes(y = median, colour = candidate)) +
  ylim(30, 60) +
  scale_colour_manual(values = c("blue", "red"), "Candidate") +
  scale_fill_manual(values = c("blue", "red"), guide = F) +
  geom_point(data = polls3, aes(x = end_date, y = `Clinton`), size = 0.2, colour = "blue") +
  geom_point(data = polls3, aes(x = end_date, y = Trump), size = 0.2, colour = "red") +
  theme_lendable() + # Thanks to my employer for their awesome theme!
  xlab("Date") +
  ylab("Implied vote share") +
  ggtitle("Poll aggregation with state-space smoothing",
          subtitle= paste("Prior of 50% initial for Clinton, 30% for Trump on", min(polls3$end_date)))
```

Bibliography

Meager, R. (2016). Aggregating distributional treatment effects: A bayesian hierarchical analysis of the microcredit literature.

Rubin, D. (1981). Estimation in parallel randomized experiments. *Journal of education statistics*, 6:377–401.