mcmc-talk

May 31, 2022

[19]:
```
# Display the Readme file as a notebook header:
import IPython
with open("README.md") as f:
    display(IPython.display.Markdown(f.read()))
```

1 mcmc-talk

Greetings! This repository houses the materials for a presentation given to the Murray Lab on using Markov Chain Monte Carlo for fitting statistical models. The original presentation was on May 3, 2022.

The slides are in the format of a Jupyter notebook written to be presentation-friendly. It can be found at `./mcmc-talk.ipynb` and may be updated from time to time. For those interested, the slides were rendered using the Jupyter extension RISE. A few supporting images were pulled from the internet or various texts. Wherever used in the notebook, attribution is given adjacent to the embedded images.

You are highly encouraged to follow along and experiment by running and modifying the Python notebook for yourself. The only required libraries are `bambi==0.7.1` (link) and `seaborn`; many other supporting libraries (`numpy`, `pandas`, `matplotlib`, `pymc3`, etc.) are pulled in as dependences. An optional but helpful library is `graphviz` (link), which is used to provide nice visualizations of Bayesian graphical models.

If reading the materials as a static document, the notebook is also available in pdf form at `./mcmc-talk.pdf`.

Please feel free to reach out to me with questions/follow-ups, or to access a recording of the original presentation!

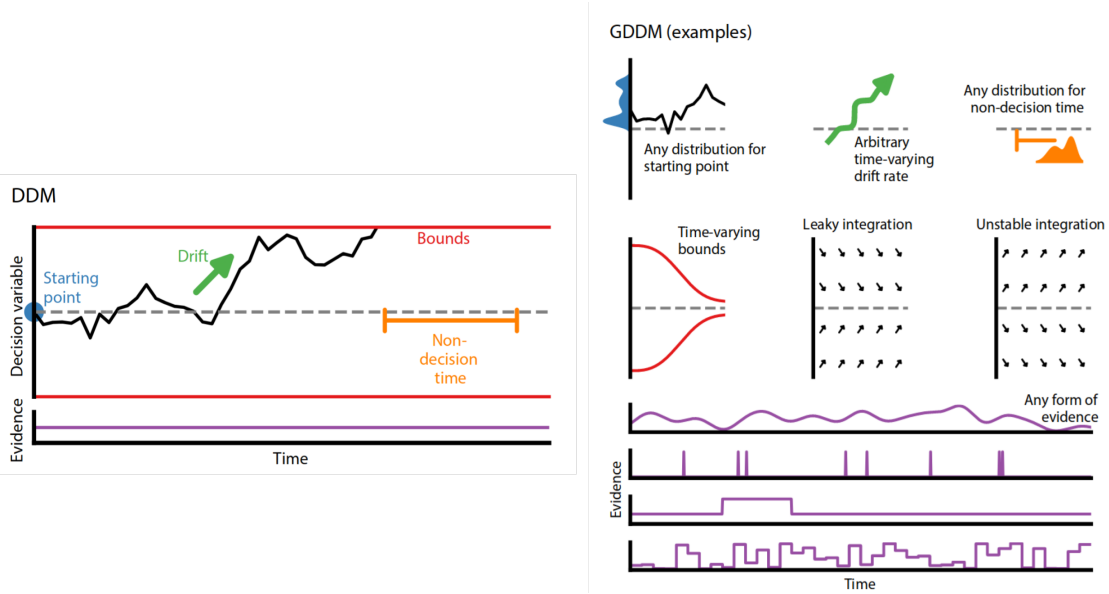
Continue to the notebook for more…

2 Intro to MCMC (*Markov Chain Monte Carlo*) for Model Fitting

2.0.1 Cove Geary

2.0.2 Last Updated May, 2022

2.1 Motivating Problem: PyDDM



Before diving headfirst into the land of Bayesian models, Markov Chains, and MCMC, here's an example of a motivating problem.

Our lab has developed a great package for fitting *Generalized Drift Diffusion Models* or GDDMs, `PyDDM` (homepage, paper). A GDDM is a flexible method for modeling response times in two-alternative forced-choice tasks. In a simple DDM, fittable parameters may include:

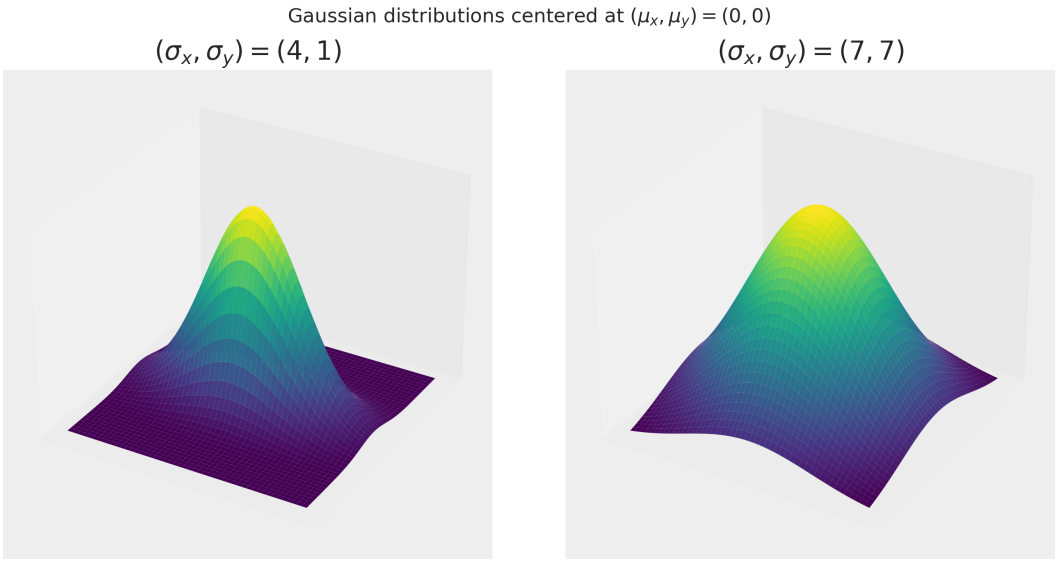
- Starting point of the decision particle
- Drift rate of the particle
- Noisiness of the drift process
- The width between the upper and lower bounds
- Non-decision time (e.g. to account for other non-decision-related processing)

In a GDDM, these parameters are much more complex, and there is usually no analytical solution. In order to fit a GDDM to data, `PyDDM` solves the Fokker-Plank equation via some fancy numerical methods and performs maximum likelihood estimation (MLE). What we get is a point estimate of the most likely model parameters given an observed dataset of reaction times.

- PyDDM returns a **maximum likelihood** estimate of the optimal model parameters, given some dataset of RTs. This is a **point estimate**.
- *What is a point estimate?*

Consider the two probability distributions below. Suppose that they are each likelihood distributions for a two-parameter model, given two different datasets. Consider: what is the MLE for each distribution?

(In this example, x and y axes represent each of the model parameters, while the z-axis represents probability density. The distributations are both multivariate Gaussians centered at $(x, y) = (0, 0)$.)



Answer: in both cases it is the parameter vector $(0, 0)$, because both distributions have the most probability density ("maximum likelihood") at $(0, 0)$.

What pros and cons can you imagine to estimating model parameters in this way? (…Should we be as confident with the MLE in the right-side example as we are in the left-side example?)

2.1.1 Drawbacks to using MLE to fit GDDM models:

1. We do not have any measure of uncertainty in parameter estimates.
    - Necessary for hypothesis testing/inference on parameters!
    - In other classical models that calculate a MLE point estimate (e.g. linear regression), it is possible to derive the *significance* of each model parameter so long as the modeling assumptions hold (e.g. standard error, t-statistic for linear regression).
    - This is not always possible with an arbitrary statistical model.
2. We cannot model differences in parameter values for different subjects or groups (in the same model).
    - Necessary for modeling differences in the data factors/levels! (Group or subject-level effects)
    - E.g.: treatment vs. control; or subject-specific parameters.

To summarize:

1. We do not have any measure of uncertainty in parameter estimates.
2. We cannot model differences in parameter values for different subjects or groups.

2.1.2 A Bayesian, sampling-based solution:

→ We can use MCMC to calculate the **full distribution** of optimal model parameters. (More formally: we can draw many samples from the posterior distribution of a complex model, given some dataset to fit.)

Bonus content: specifying our model in this way allows us to specify **arbitrary hierarchy** among model parameters. Groub/subject-level effects come free with a Bayesian model! This will allow us to make the most of our data by leveraging the structure that exists within it.

Let's move on to developing the necessary foundations that lead to MCMC for model fitting.

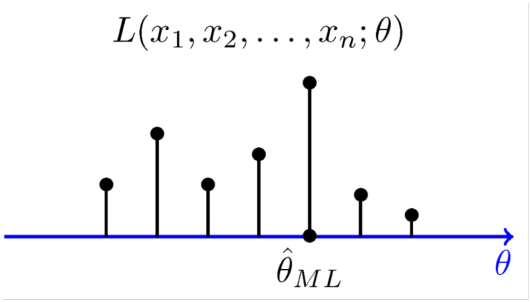
2.2 Primer: Model-Fitting

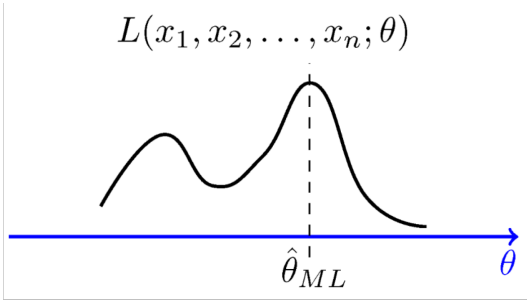
Essentially everyone in this lab is fitting statistical models of some kind; hopefully this idea should not be too foreign to you. But the processes or algorithms are sometimes abstracted so far away from what we need to do to use a model that we may lose sight of the basics.

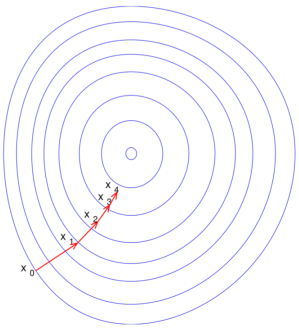
Poll for the room: what model-fitting methods can you name? (What have you used, heard of..., ?)

Your answers: * MLE (maximum likelihood) * Photoshop (ha) * MAP (maximum a posteriori) * MSE (mean squared error loss function) * Gaussian Processes

2.2.1 Simple Examples







A couple of more "classical" examples that may be familiar.

Top: MLE (this is equivalent to OLS for normal linear regression). Bottom: gradient descent.

Images from:

- https://www.probabilitycourse.com/chapter8/8_2_3_max_likelihood_estimation.php
- https://en.wikipedia.org/wiki/Gradient_descent#/media/File:Gradient_descent.svg

As discussed, classical statistical approaches generally provide **point estimates** for the parameters. Mainly what we care about in these approaches is the point value that the method "converges to."

How can we calcuate the **distribution** of model parameters that fit the data?

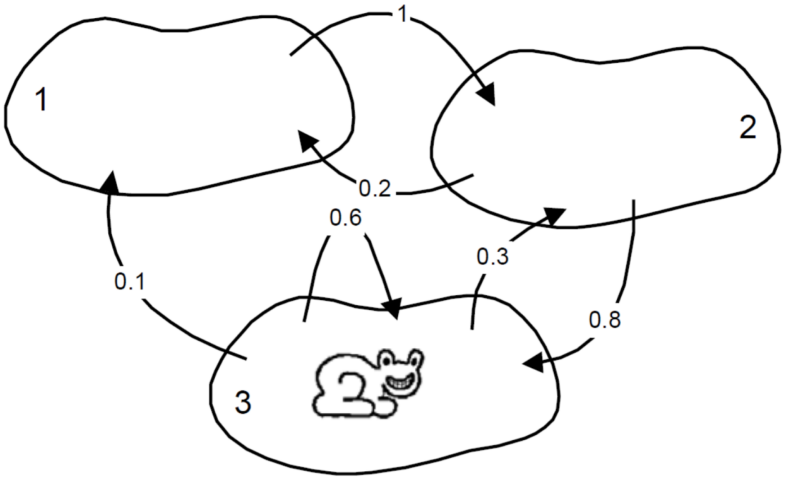
- If we can sample repeatedly *from that distribution*, then those samples would represent an *empirical posterior distribution*.
- MCMC provides a way to sample from the posterior. "Convergence" is guaranteed by the use of a special **Markov chain**.

2.3 Primer: Markov Chains

A Markov chain is a sequence of random variables $X_0, X_1, ...$, where X_t represents the state of the system at time t. The transition $X_t \to X_{t+1}$ is given by a probability transition matrix.

The formal definition of a Markov chain is more specific than this. One will need to specify, for example, the "state space" (the set of all possible states), the initial condition/distribution of the beginning of the chain, and, crucially, the properties of that transition matrix. But for our purposes, this will be plenty to understand the role of Markov chains in MCMC.

To operationalize this concept, consider the following simple example:



$$P = \begin{array}{c} \\ 1 \\ 2 \\ 3 \end{array} \begin{array}{c} \begin{array}{ccc} 1 & 2 & 3 \end{array} \\ \begin{pmatrix} 0 & 1 & 0 \\ .2 & 0 & .8 \\ .1 & .3 & .6 \end{pmatrix} \end{array}$$

This example and image are given by the manuscript "Probability, Statistics, and Data Analysis: A Bayesian Introductory Course" by Joe Chang, June 6, 2019.

The diagram on the left expresses a Markov chain for a frog that leaps from lily pad to lily pad. If we think of each lily pad as a "state", then the state space is $S = \{1, 2, 3\}$. The directed edges between lily pads represent the probability of the frog jumping from one lily pad to another at each transition time.

We can also succinctly represent this process with the transition matrix P as shown on the right: each entry P_{ij} gives the probability of jumping from lily pad i to j.

Something nice about Markov chains is that they are easy to simulate. We could code up a virtual frog in only a few lines of code, as all that we need to know are the transition probabilities and the starting conditions! It helps that these days, computers are quite good at running `for` loops very quickly.

The deep froggy question:

- *How likely is it, in the long run, that the frog is on lily pad i at any given time?*

Can be equivalently rephrased:

- *What is the limiting distribution of X_t as $t \to \infty$?*

This is often called the **stationary distribution** of the Markov chain. In this case, one may derive an algebraic solution. However, if a stationary distribution exists, we can always approximate it to some arbitrary level of precision by simulating many transitions of the chain!

So! If a transition function can be crafted which has the model's posterior as its stationary distribution, then we can estimate the posterior simply by simulating a chain with this transition.

Keep this insight in mind for later.

For a hint on how this fancy transition function can come about without knowing the posterior distribution, we must turn to **Bayes' Rule**. Note that you don't need to memorize this math in order to use MCMC, but it is helpful to keep in mind.

Furthermore, I would be absolutely remiss if I gave a talk on Bayesian model-fitting and never mentioned Bayes' Rule.

2.3.1 Obligatory Bayes' Rule

It turns out that we can craft such a transition function without needing to know the posterior distribution.

We only need:

- $P(\theta)$: the model's **prior probability**
- $P(D|\theta)$: the model's **likelihood**

(where θ is the model parameter vector and D represents the dataset).

Behind the scenes of MCMC, this is due to **Bayes' Rule**:

$$P(\theta|D) = \frac{P(D|\theta)P(\theta)}{P(D)} \propto P(D|\theta)P(\theta)$$

More simply: **Posterior** $\propto$ **likelihood** $\times$ **prior**.

2.3.2 Markov Chain Takeaways

- Simulating many steps of the chain will produce an empirical distribution that gives the probability of being in each state.
  - "Memoryless": each transition depends only on the current state, not the history of past states.
  - $\implies$ a simulated Markov chain is **not** a set of independently distributed samples. There will be some autocorrelation.
- In MCMC for model fitting:
  - "State" is any valid combination of model parameters. (The state space will likely be continuous and multidimensional.)
  - We will want to craft a transition function that has the posterior as its stationary distribution.
  - Bayes' Rule will help us out. However, we must set priors on our model parameters.

2.4 Primer: Probabilistic Graphical Models ("Bayesian Networks")

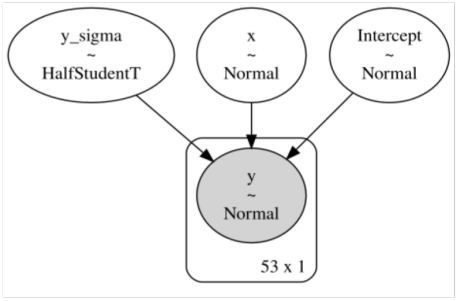
The final piece of the puzzle is how to specify a model in a way that will allow us to run MCMC. As alluded to in our discussion of Bayes' Rule, the two things that we'll need to define are the **prior** $P(\theta)$ and the **likelihood** $P(D|\theta)$.

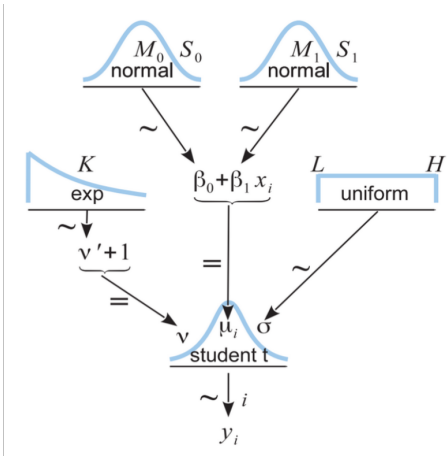
But what if we have many individual parameters within the vector θ, and we'd like to place different priors on each of them? Perhaps we also want to express hierarchical or conditional relationships between these parameters. **Bayesian networks** (a specific form of probabilistic graphical models) provide an easy way to express models in this way.

In a Bayesian network, each parameter (or hyperparameter) is expressed as a node in a directed graph. In general, each node represents a random variable: prior, likelihood, data, etc. The idea behind this graph is that parent nodes (those with no child nodes) represent independent distributions. Child nodes take their parent node(s) as parameter value(s), hence child nodes represent distributions conditioned on the parent(s). See Horný 2015 for a brief but more theoretical introduction.

Given this background, let's consider a couple examples of regression models expressed in this graphical specification.

Each diagram represents a model designed to perform simple linear regression. In both, the variable y represents the observed/modeled data:





Notice the differing use of priors and likelihood.

The left diagram was generated by `pymc`.

The right diagram is captioned: Figure 17.2: A model of dependencies for robust linear regression. The datum, yi at the bottom of the diagram, is distributed around the central tendency i, which is a linear function of xi. Compare with Figure 16.11 on p. 437. Copyright © Kruschke, J. K. (2014). Doing Bayesian Data Analysis: A Tutorial with R, JAGS, and Stan. 2nd Edition. Academic Press / Elsevier.

Can you identify: the **intercept** term? The **slope** term? The priors that are set on each?

A: In the left model, the intercept is the top-rightmost node Intercept, and the slope is x. In the right, the the intercept is named β_0 and the slope β_1. All of these variables are given Normal

distributions as their priors. The right diagram diagram is more verbose, as it shows us the names of the distribution parameters and specifically how child nodes depend on the parents.

Can you identify: the **likelihood**? How is it distributed?

A: In general, all of our Bayesian networks will have the likelihood as the final child node that depends, whether directly or indirectly, upon all of the other nodes. Hence the likelihood node is the bottom node in each model diagram.

A further hint as to which node is the likelihood node: recall that the likelihood is defined as $P(D|\theta)$. Therefore, because the bottom node expresses how the observed data (outcome y in the regression problem) is distributed (according to a Normal or a Student-T distribution, respectively), and it is conditional upon all the model parameters by nature of being the "last" child node, it must be the likelihood.

2.4.1 Bayesian Networks Takeaways:

- Are directed acyclic graphs, such that:
- Each node is a random variable (it expresses a distribution).
  - Nodes can represent model parameters/hyperparameters (e.g. β_0, υ, σ, or Intercept), some deterministic function of the model parameters (e.g. $\beta_0+\beta_1 x_i$, or $\upsilon' + 1$), and model likelihood (e.g. $y \sim P(D|\theta)$).
  - A child node is conditioned (in the probablistic sense) on its parent node(s).
- Can be used to represent any parametric statistical model, but we **must** set priors over the parameters.
- *MCMC will sample the posterior distribution of a graphical model.*

Other niceties: * Make it very easy to express any arbitrary hierarchy or relationships between model parameters. * Express a *generative model*: if we can sample from each node of the graph, we can sample from the likelihood, which generates samples of y. * We can include nuisance parameters in our model but marginalize them out later.

Hopefully the ideas behind Bayesian networks are sitting easy with you. The graphical model framework is immensely flexible, and it can be leveraged to Bayes-ify many classical models that you may be familiar with.

- For more examples of how to translate various statistical models into Bayesian networks, consider browsing the Examples page of the `pymc` library. `pymc` is a general-purpose Python library for building Bayesian models; it allows you to express Bayesian models in an intuitive object-oriented way.
- For a simple example, this notebook walks through building simple linear models, culminating in a Bayesian "mixed model."
- For an advanced example, this notebook gives the example of building a **Bayesian neural network**. The possibilities are far and wide.

Payoff moment: Given the foundations that we've built, we can now (finally) express the essence of MCMC in concise terms!

2.5 MCMC in a nutshell

Given a **graphical model**, MCMC is a class of algorithms that provide a **Markov transition function** designed to have a **stationary distribution** equivalent (or pro-

portional) to that model's **posterior distribution**.

What does this mean for us?

> All that we must do is express our model as a Bayesian network, defining priors and the likelihood. The MCMC algorithm gives us the transition function. Then "model fitting" is as simple as simulating the Markov chain for many steps.

When we combine that sequence of samples from the Markov chain into a histogram, what we get is an **empirical joint posterior distribution** over the model's parameters. It's our estimate of the *distribution of best-fitting parameters*!

There are many different algorithms for MCMC. This remains an active area of research! The primary difference between them all: how that transition is calculated. Here is small number of important approaches, roughly sorted from most "classical" to most "modern":

- Gibbs Sampler
- Metropolis-Hastings
- Slice Sampler
- Particle/ensemble-based
- Hamiltonian Monte Carlo

(The last two are a toss-up in terms of ordering.)

Helpful note: these algorithms are often referred to as simply the "sampler" or the "step" method by practitioners and by statistical programming libraries.

The algorithmic details of these various methods are readily available in many texts and online. For some starting-point references to each, consider the following sources:

- For Gibbs sampling and Metropolis-Hastings: chapter 12 of Wasserman, L., & Lafferty, J. (2014). Statistical Machine Learning for Structured and High Dimensional Data. (link)
- Slice sampling: original paper
- Ensemble-based: emcee package and paper on ensemble samplers with affine invariance
- HMC: Betancourt, M. (2017). A conceptual introduction to Hamiltonian Monte Carlo. (link)

If you wish to choose just one, the paper on HMC by Betancourt also serves as an excellent introduction to MCMC in general.

In case anyone in presentation—or you, dear reader—asks for an example of a step method, here is a succinct description of the Metropolis-Hastings algorithm from "Probability, Statistics, and Data Analysis: A Bayesian Introductory Course" by Joe Chang, June 6, 2019.

Suppose $X_t = X$ and we wish to generate a step for X_{t+1}. Then proceed as follows:

- Stage 1: Propose a *"candidate"* state, y say, by drawing y randomly from some symmetric probability density function centered on x (we can choose this density).
- Stage 2: Now we make a probabilistic decision about whether to *"accept"* the candidate, making use of the desired density f. Calculate the ratio $r = \frac{f(y)}{f(x)}$.
  - If $r > 1$, accept the candidate; that is, take $X_{t+1} = y$.
  - If $r \leq 1$, accept the candidate with probability r. That is, we take

$$X_{t+1} = \begin{cases} y & \text{with prob } r \\ x & \text{with prob } 1 - r. \end{cases}$$

2.6 Putting it all together

Your workflow in model-fitting with MCMC should look something like the following:

1. Define your graphical model. (Set priors and likelihood.)
2. Sanity-check your priors. (Prior predictive check.)
3. Run MCMC. (Some refer to this simply as the "inference button.")
4. Investigate the posterior:
   - Assess MCMC convergence. ($\widehat{R}$, trace plots, ESS, etc.)
   - Sanity-check the posterior distribution. (Posterior predictive check.)
5. Profit! (Use the posterior to make inferences about your data.)

3 Worked Example:

3.1 Linear Regression with MCMC

We've now laid the groundwork to work through a case study! We will build a couple of Bayesian regression models, starting with a simple example, then expanding it to model hierarchy in our dataset.

Citing sources: This case study is adapted from the "Sleep Study" notebook in the Bambi documentation (link). The dataset is commonly used when teaching linear models, popularized by the R package `lme4`. This case study also draws inspiration from a notebook by Tristan Mahr, especially the points about using partial pooling to estimate parameters even when subgroup data is missing or low sample size.

For this case study, we'll be using the Python library `bambi`, which provides automatic weakly-informative priors by default.

Feel free to skip ahead past the following blocks of setup code…

[20]:
```
%%html
<!-- For hiding code cells in a presentation but still displaying output; from
 ↪https://github.com/damianavila/RISE/issues/32 -->
<style>
```

```
 .container.slides .celltoolbar, .container.slides .hide-in-slideshow {
    display: None ! important;
}
</style>
```

```
<IPython.core.display.HTML object>
```

[21]:
```
# https://github.com/damianavila/RISE/issues/32
def hide_code_slideshow():
    from IPython import display
    import binascii
    import os
    uid = binascii.hexlify(os.urandom(8)).decode()
    html = """<div id="%s"></div>
    <script type="text/javascript">
        $(function(){
            var p = $("#%s");
            if (p.length==0) return;
            while (!p.hasClass("cell")) {
                p=p.parent();
                if (p.prop("tagName") =="body") return;
            }
            var cell = p;
            cell.find(".input").addClass("hide-in-slideshow")
        });
    </script>""" % (uid, uid)
    display.display_html(html, raw=True)
```

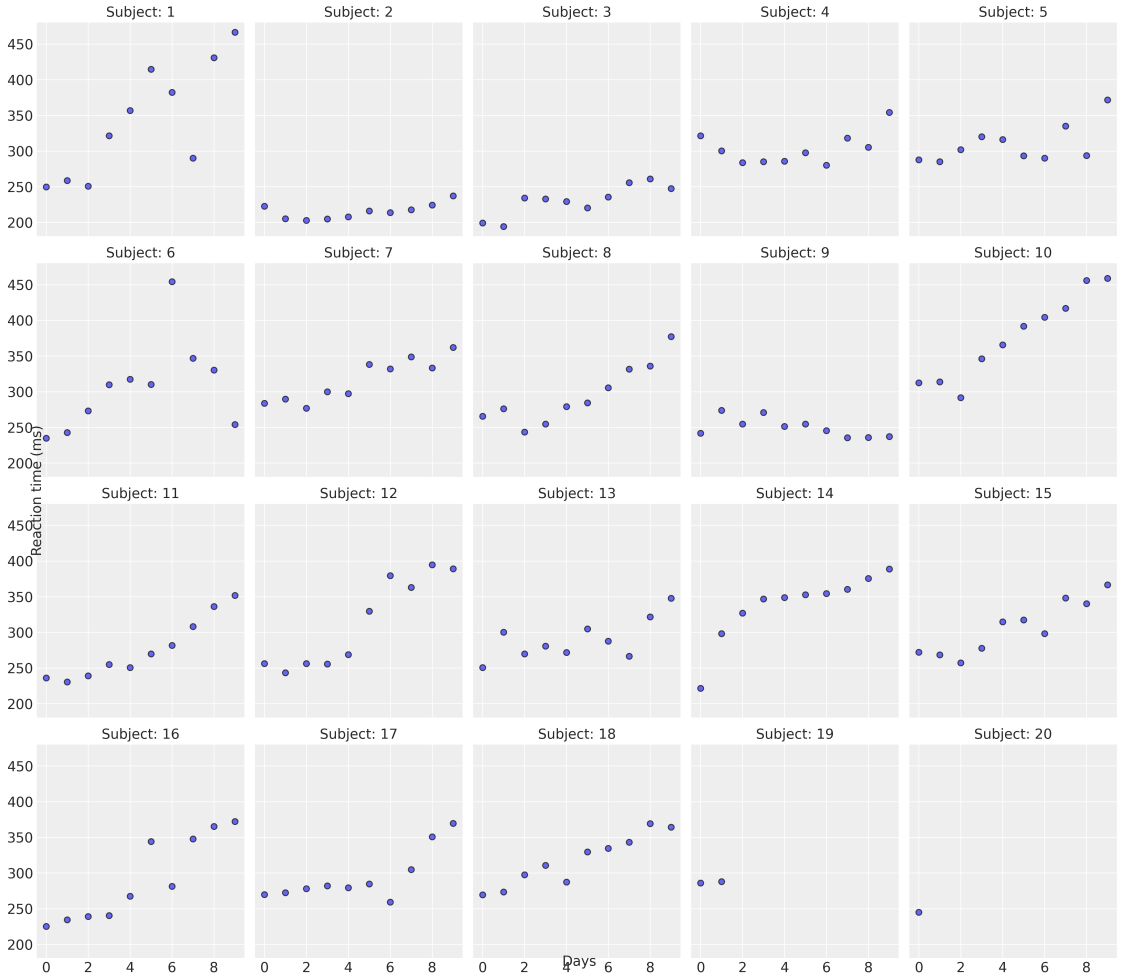
[22]:
```
import arviz as az
import bambi as bmb
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import seaborn as sns

az.style.use("arviz-darkgrid")
plt.rcParams.update({"figure.autolayout": True, "figure.dpi": 300})
SEED = 7355608

data = bmb.load_data("sleepstudy")
fake_data = pd.DataFrame([
    {"Reaction": 286, "Subject": -1, "Days": 0},
    {"Reaction": 288, "Subject": -1, "Days": 1},
    {"Reaction": 245, "Subject": -2, "Days": 0}])
data = pd.concat([data, fake_data])
data.Subject = data.Subject.factorize()[0] + 1
data.Subject = pd.Categorical(data.Subject)
```

```
def plot_data(data):
    assert len(pd.unique(data["Subject"])) == 20
    fig, axes = plt.subplots(4, 5, figsize=(16, 14), sharey=True, sharex=True)
    fig.subplots_adjust(left=0.075, right=0.975, bottom=0.075, top=0.925,␣
 ↪wspace=0.03)
    axes_flat = axes.ravel()
    for ax, subject in zip(axes_flat, data["Subject"].unique()):
        subset = data.query("Subject == @subject")
        ax.scatter(subset["Days"], subset["Reaction"], color="C0", ec="black",␣
 ↪alpha=0.7)
        ax.set_title(f"Subject: {subject}", fontsize=14)
    ax.xaxis.set_ticks([0, 2, 4, 6, 8])
    fig.text(0.5, 0.02, "Days", fontsize=14)
    fig.text(0.03, 0.5, "Reaction time (ms)", rotation=90, fontsize=14,␣
 ↪va="center")
    return axes
```

3.1.1 Our dataset

[23]:
```
plot_data(data)
plt.show()
```

```
/tmp/ipykernel_36130/3154367302.py:24: UserWarning: This figure was using
constrained_layout, but that is incompatible with subplots_adjust and/or
tight_layout; disabling constrained_layout.
  fig.subplots_adjust(left=0.075, right=0.975, bottom=0.075, top=0.925,
wspace=0.03)
```



These data are adapted from Belenky et al. (2003), a study of the cognitive impacts of sleep deprivation. In this dataset, all subjects were restricted to sleeping only three hours per night. Each day, they were assessed with a simple cognitive test that assesses reaction time (in ms) to a visual stimulus.

Reference: Belenky, G., Wesensten, N. J., Thorne, D. R., Thomas, M. L., Sing, H. C., Redmond, D. P., … & Balkin, T. J. (2003). Patterns of performance degradation and restoration during sleep restriction and subsequent recovery: A sleep dose-response study. Journal of sleep research, 12(1), 1-12.

See also the dataset in `lme4` (link).

3.2 Model 1: "Pooled" Model

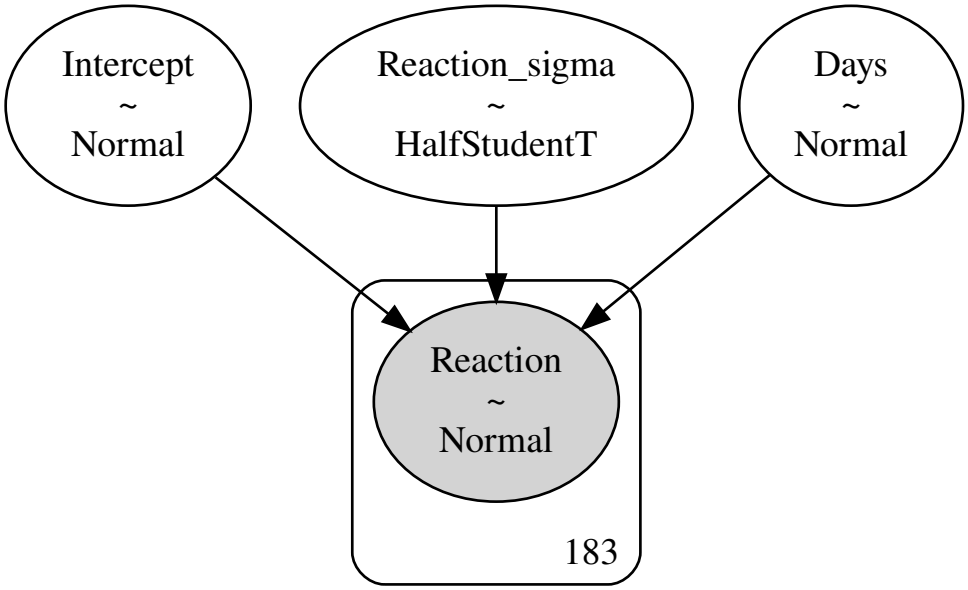
The following model is analogous to a "fixed-effects" model.

[24]:
```
model_pooled = bmb.Model("Reaction ~ 1 + Days", data)
display(model_pooled)
```

```
model_pooled.build()
model_pooled.graph()
```

```
Formula: Reaction ~ 1 + Days
Family name: Gaussian
Link: identity
Observations: 183
Priors:
  Common-level effects
    Intercept ~ Normal(mu: 298.0897, sigma: 255.1709)
    Days ~ Normal(mu: 0.0, sigma: 48.1899)

  Auxiliary parameters
    sigma ~ HalfStudentT(nu: 4, sigma: 55.8615)
```

[24]:



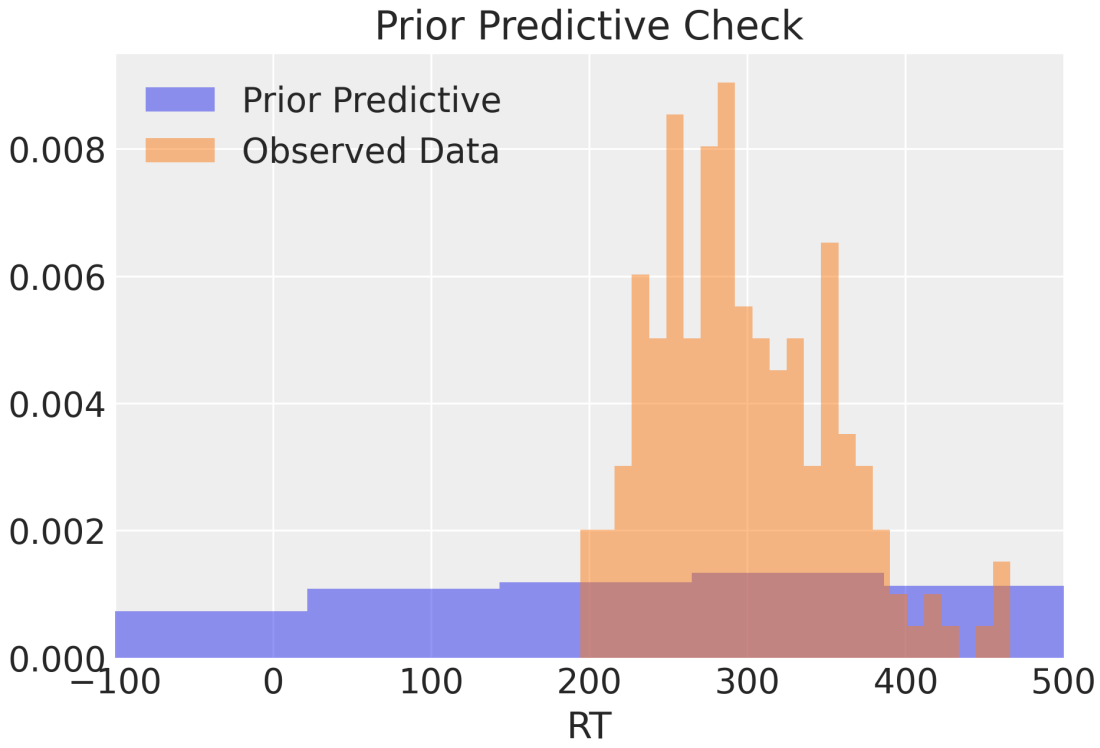
3.2.1 Before sampling: the prior predictive check

- Compare the generative model's predictions to your observed data

[25]:
```
prior_predictive_RTs = model_pooled.prior_predictive().prior_predictive.
↪Reaction.values.flatten()
plt.hist(prior_predictive_RTs, alpha=0.5, bins=25, density=True, label="Prior␣
↪Predictive")
plt.hist(data.Reaction.values, alpha=0.5, bins=25, density=True,␣
↪label="Observed Data")
plt.xlim([-100, 500])
plt.title("Prior Predictive Check")
```

```
plt.xlabel("RT")
plt.legend()
plt.show()
hide_code_slideshow()
```

```
/home/cove/Lab/mcmc-talk/venv-mcmc/lib/python3.10/site-
packages/IPython/core/pylabtools.py:151: UserWarning: This figure was using
constrained_layout, but that is incompatible with subplots_adjust and/or
tight_layout; disabling constrained_layout.
  fig.canvas.print_figure(bytes_io, **kw)
```



We see the impact of the *weakly* informative prior: the prior predictive distribution is much wider than the true data. Primarily we want to see that the prior distribution allows for the range of observed data, which it certainly does in this case.

Some (though not all) Bayesians may argue that a more informative prior could be more effective, perhaps yielding tighter estimates or more efficient sampling. For our intents and purposes, the default Bambi prior will work just fine.

3.3 Sampling

[26]:
```
# The following line is all we need to run MCMC.
trace_pooled = model_pooled.fit(draws=1000, chains=4, random_seed=SEED)
```

```
Auto-assigning NUTS sampler…
Initializing NUTS using jitter+adapt_diag…
Multiprocess sampling (4 chains in 4 jobs)
NUTS: [Reaction_sigma, Days, Intercept]
```

```
<IPython.core.display.HTML object>
```

```
<IPython.core.display.HTML object>
```

```
Sampling 4 chains for 1_000 tune and 1_000 draw iterations (4_000 + 4_000 draws
total) took 1 seconds.
```

3.3.1 Sampler Diagnostics

[27]:
```
az.summary(trace_pooled, kind="diagnostics")
```

[27]:

	mcse_mean	mcse_sd	ess_bulk	ess_tail	r_hat
Intercept	0.083	0.059	5780.0	2982.0	1.0
Days	0.017	0.012	5087.0	3134.0	1.0
Reaction_sigma	0.036	0.026	5184.0	3219.0	1.0

> One can interpret R-hat with diverse seeding as an operationalization of the qualitative statement that, after warmup, convergence of the Markov chain should be relatively insensitive to the starting point, at least within a reasonable part of the parameter space. This is the closest we can come to verifying empirically that the Markov chain is geometrically ergodic, which is a critical property if we want a central limit theorem to hold for approximate posterior expectations.

Vehtari, A., Gelman, A., Simpson, D., Carpenter, B., & Bürkner, P. C. (2021). Rank-normalization, folding, and localization: An improved^R for assessing convergence of MCMC (with Discussion). Bayesian analysis, 16(2), 667-718.

3.3.2 Trace Plots

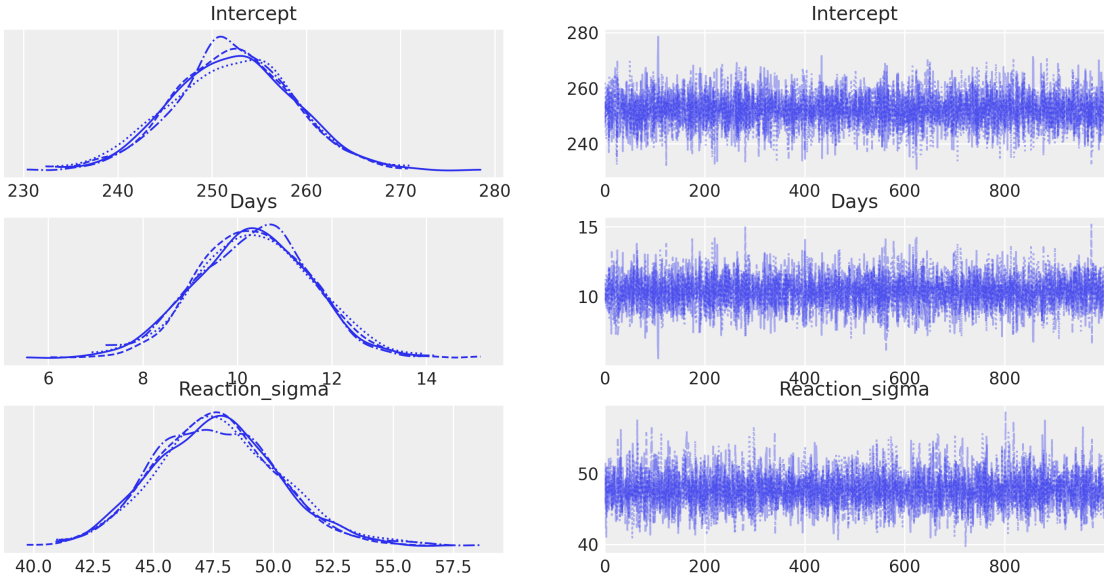
Each line type represents a different chain of our sampling process. (Recall that four independent chains were run.)

Left: smoothed histogram. Right: trace plot (the state of the chain at each iteration t).

[28]:
```
az.plot_trace(trace_pooled)
plt.show()
```

```
/home/cove/Lab/mcmc-talk/venv-mcmc/lib/python3.10/site-
packages/IPython/core/pylabtools.py:151: UserWarning: This figure was using
constrained_layout, but that is incompatible with subplots_adjust and/or
```

```
tight_layout; disabling constrained_layout.
  fig.canvas.print_figure(bytes_io, **kw)
```

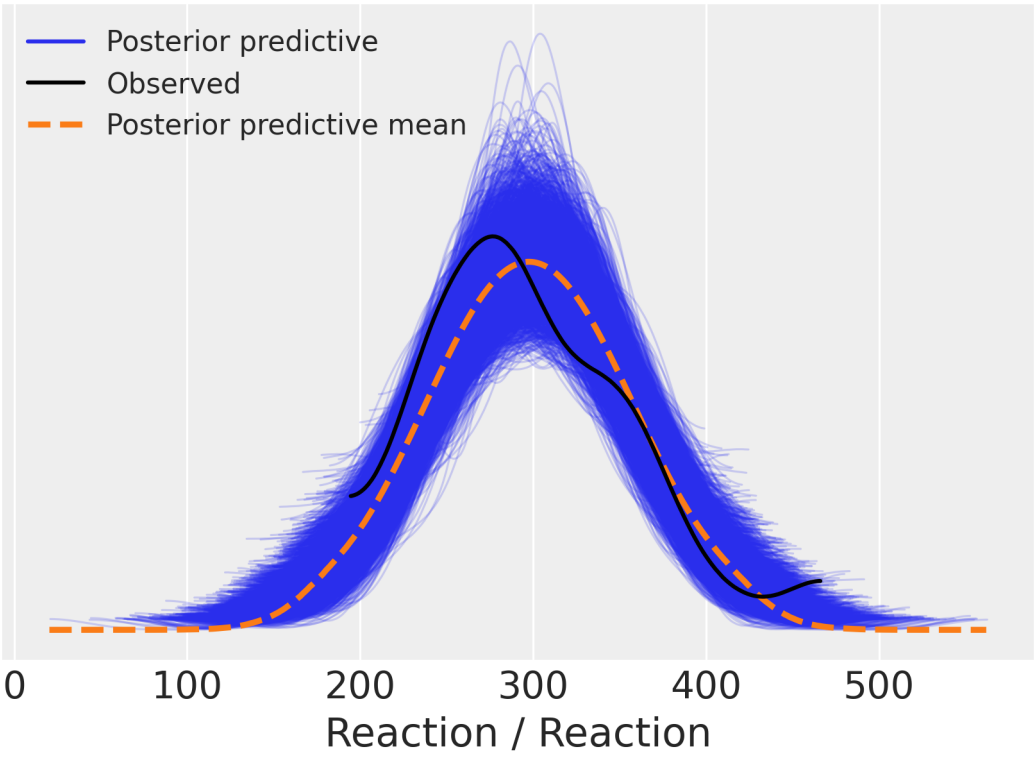


3.3.3 Posterior Predictive Check

- In the same spirit of the prior predictive check, but now with our posterior estimates!

[29]:
```
model_pooled.predict(trace_pooled, kind="pps")
az.plot_ppc(trace_pooled)
```

[29]:
```
<AxesSubplot:xlabel='Reaction / Reaction'>
```



3.4 Model 2: "Partial Pooling" Model

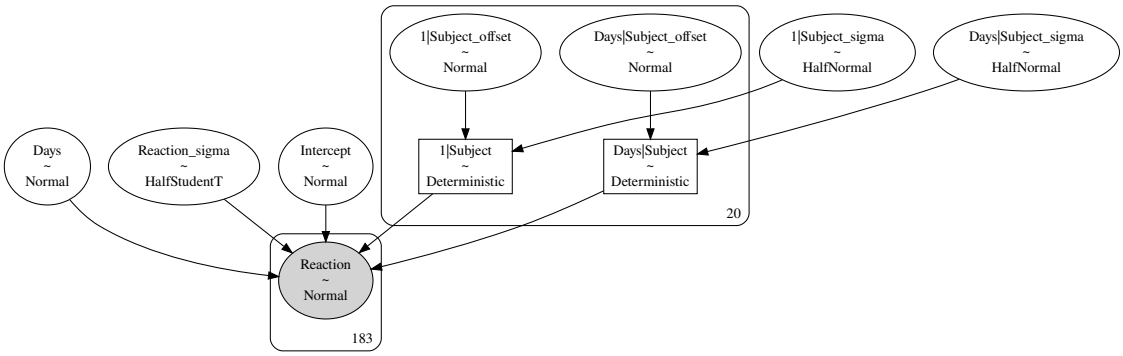
The following model is analogous to a "mixed-effects" model.

[30]:
```
model_hier = bmb.Model("Reaction ~ 1 + Days + (Days | Subject)", data)
display(model_hier)
model_hier.build()
model_hier.graph()
```

```
Formula: Reaction ~ 1 + Days + (Days | Subject)
Family name: Gaussian
Link: identity
Observations: 183
Priors:
  Common-level effects
    Intercept ~ Normal(mu: 298.0897, sigma: 255.1709)
    Days ~ Normal(mu: 0.0, sigma: 48.1899)

  Group-level effects
    1|Subject ~ Normal(mu: 0, sigma: HalfNormal(sigma: 255.1709))
    Days|Subject ~ Normal(mu: 0, sigma: HalfNormal(sigma: 48.1899))

  Auxiliary parameters
```

```
sigma ~ HalfStudentT(nu: 4, sigma: 55.8615)
```

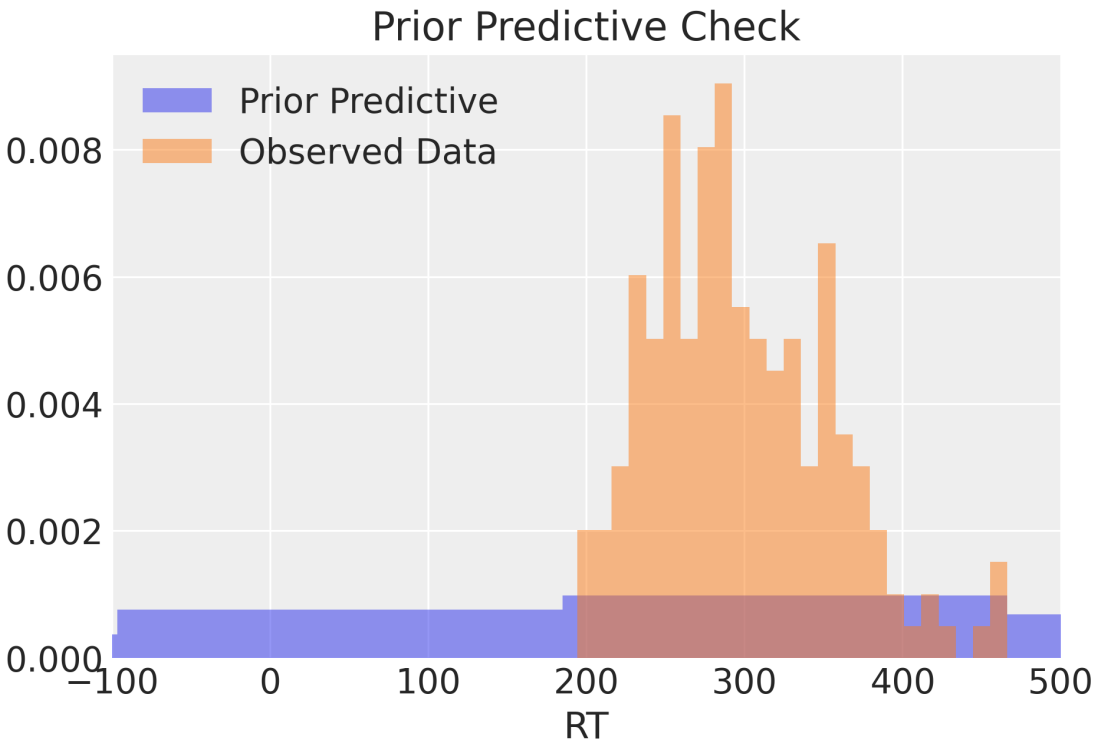
[30]:



3.4.1 The prior predictive...

[31]:
```
prior_predictive_RTs = model_hier.prior_predictive().prior_predictive.Reaction.
  ↪values.flatten()
plt.hist(prior_predictive_RTs, alpha=0.5, bins=25, density=True, label="Prior␣
  ↪Predictive")
plt.hist(data.Reaction.values, alpha=0.5, bins=25, density=True,␣
  ↪label="Observed Data")
plt.xlim([-100, 500])
plt.title("Prior Predictive Check")
plt.xlabel("RT")
plt.legend()
hide_code_slideshow()
```

```
/home/cove/Lab/mcmc-talk/venv-mcmc/lib/python3.10/site-
packages/IPython/core/pylabtools.py:151: UserWarning: This figure was using
constrained_layout, but that is incompatible with subplots_adjust and/or
tight_layout; disabling constrained_layout.
  fig.canvas.print_figure(bytes_io, **kw)
```



3.4.2 Running MCMC

[32]:
```
trace_hier = model_hier.fit(draws=1000, chains=4, random_seed=SEED)
model_hier.predict(trace_hier, kind="pps")
```

```
Auto-assigning NUTS sampler…
Initializing NUTS using jitter+adapt_diag…
Multiprocess sampling (4 chains in 4 jobs)
NUTS: [Reaction_sigma, Days|Subject_offset, Days|Subject_sigma,
1|Subject_offset, 1|Subject_sigma, Days, Intercept]
```

```
<IPython.core.display.HTML object>
```

```
<IPython.core.display.HTML object>
```

```
Sampling 4 chains for 1_000 tune and 1_000 draw iterations (4_000 + 4_000 draws
total) took 8 seconds.
```

3.4.3 Base Diagnostics

[33]:
```
az.summary(trace_hier)
```

[33]:

	mean	sd	hdi_3%	hdi_97%	mcse_mean	mcse_sd	\
Intercept	252.332	7.012	239.644	265.688	0.154	0.109	

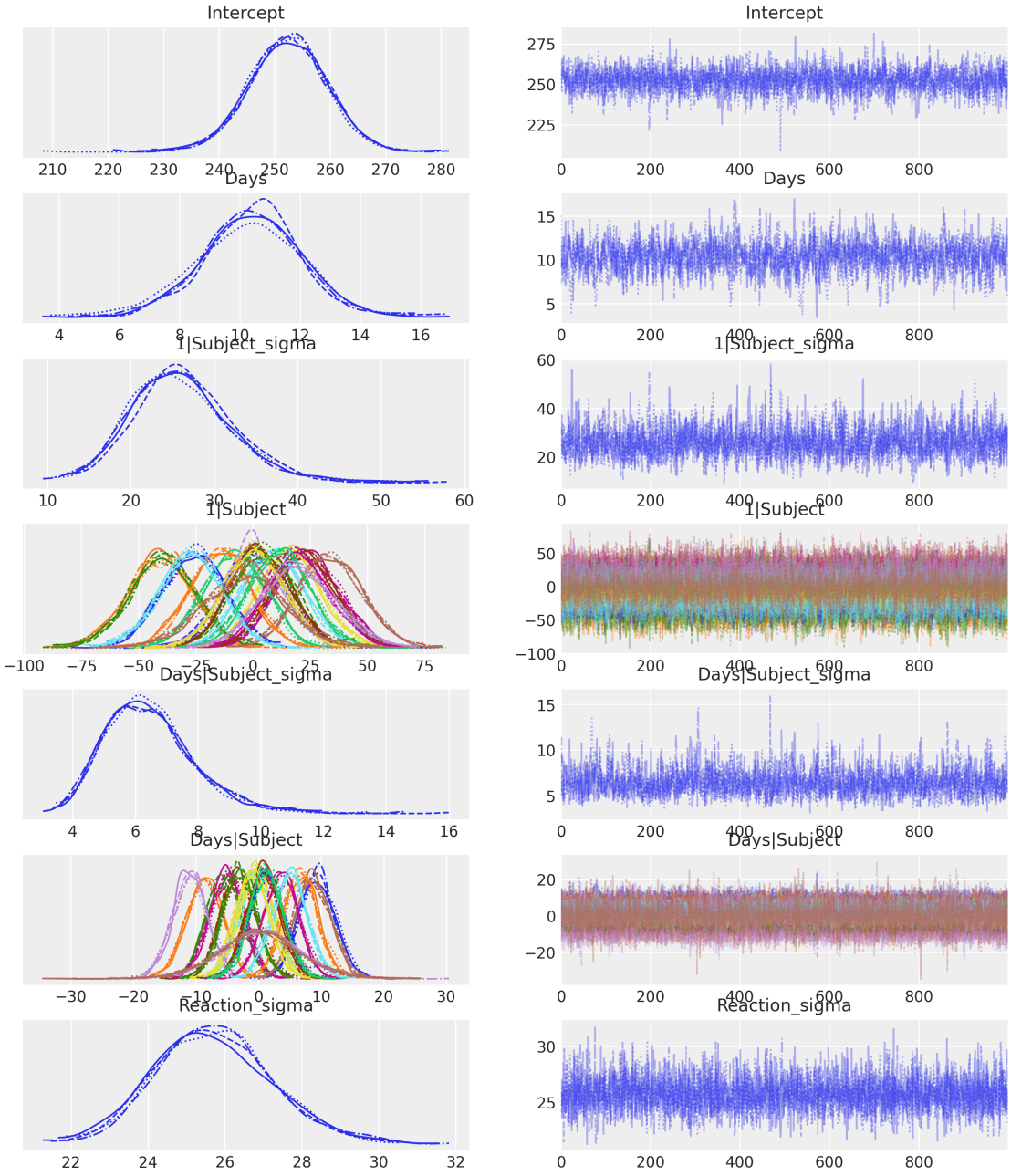
Days	10.369	1.702	7.017	13.392	0.050	0.036
1|Subject_sigma	26.101	6.139	15.380	37.697	0.146	0.104
1|Subject[1]	0.589	13.841	-27.014	25.030	0.239	0.215
1|Subject[2]	-40.608	14.366	-66.874	-13.550	0.231	0.165
1|Subject[3]	-39.234	14.508	-67.895	-12.847	0.247	0.176
1|Subject[4]	24.099	13.869	-1.164	50.039	0.252	0.191
1|Subject[5]	22.256	13.805	-3.900	46.948	0.250	0.194
1|Subject[6]	8.653	13.638	-15.583	35.334	0.242	0.191
1|Subject[7]	16.388	13.483	-8.224	42.090	0.237	0.188
1|Subject[8]	-8.047	13.679	-33.236	16.526	0.224	0.183
1|Subject[9]	0.205	13.297	-24.661	25.440	0.230	0.196
1|Subject[10]	33.950	14.172	6.962	60.247	0.237	0.172
1|Subject[11]	-26.095	13.713	-52.401	-0.745	0.235	0.166
1|Subject[12]	-14.518	13.577	-40.236	10.396	0.210	0.169
1|Subject[13]	4.291	13.347	-19.551	30.101	0.226	0.192
1|Subject[14]	19.909	13.736	-4.962	46.134	0.232	0.186
1|Subject[15]	3.125	13.113	-21.011	28.048	0.232	0.201
1|Subject[16]	-27.187	13.826	-54.041	-1.958	0.225	0.171
1|Subject[17]	0.575	13.330	-24.024	25.822	0.239	0.191
1|Subject[18]	11.785	13.464	-13.288	37.353	0.230	0.181
1|Subject[19]	19.710	16.025	-10.047	50.591	0.242	0.190
1|Subject[20]	-3.492	18.334	-39.437	30.857	0.265	0.277
Days|Subject_sigma	6.464	1.445	3.976	9.156	0.035	0.026
Days|Subject[1]	9.415	2.855	4.231	14.834	0.059	0.043
Days|Subject[2]	-8.615	2.943	-13.985	-3.141	0.061	0.043
Days|Subject[3]	-5.461	2.901	-10.629	0.219	0.059	0.042
Days|Subject[4]	-4.980	2.809	-10.194	0.339	0.059	0.043
Days|Subject[5]	-3.127	2.842	-8.017	2.529	0.059	0.042
Days|Subject[6]	-0.298	2.802	-5.595	4.926	0.060	0.042
Days|Subject[7]	-0.186	2.793	-5.870	4.671	0.061	0.043
Days|Subject[8]	1.163	2.835	-3.988	6.701	0.063	0.046
Days|Subject[9]	-10.907	2.813	-15.911	-5.353	0.059	0.042
Days|Subject[10]	8.740	2.887	3.680	14.226	0.060	0.044
Days|Subject[11]	1.274	2.811	-4.176	6.353	0.058	0.042
Days|Subject[12]	6.775	2.799	1.530	12.024	0.059	0.044
Days|Subject[13]	-3.058	2.757	-8.295	2.302	0.056	0.040
Days|Subject[14]	3.640	2.826	-2.024	8.585	0.058	0.042
Days|Subject[15]	0.823	2.766	-4.158	6.177	0.059	0.042
Days|Subject[16]	5.037	2.855	-0.564	10.081	0.059	0.044
Days|Subject[17]	-0.962	2.790	-6.190	4.256	0.063	0.045
Days|Subject[18]	1.314	2.728	-3.901	6.395	0.058	0.043
Days|Subject[19]	0.338	6.418	-10.817	13.273	0.105	0.114
Days|Subject[20]	-0.013	6.747	-13.644	11.834	0.097	0.115
Reaction_sigma	25.710	1.519	22.951	28.663	0.025	0.018

	ess_bulk	ess_tail	r_hat
Intercept	2114.0	2235.0	1.0

```
Days                  1157.0    1459.0    1.0
1|Subject_sigma       1724.0    2309.0    1.0
1|Subject[1]          3371.0    2633.0    1.0
1|Subject[2]          3886.0    3113.0    1.0
1|Subject[3]          3421.0    2887.0    1.0
1|Subject[4]          3070.0    2736.0    1.0
1|Subject[5]          3084.0    2881.0    1.0
1|Subject[6]          3194.0    2999.0    1.0
1|Subject[7]          3284.0    3066.0    1.0
1|Subject[8]          3761.0    3024.0    1.0
1|Subject[9]          3351.0    3005.0    1.0
1|Subject[10]         3600.0    2902.0    1.0
1|Subject[11]         3393.0    3051.0    1.0
1|Subject[12]         4209.0    3394.0    1.0
1|Subject[13]         3500.0    2918.0    1.0
1|Subject[14]         3559.0    2677.0    1.0
1|Subject[15]         3213.0    2689.0    1.0
1|Subject[16]         3787.0    2784.0    1.0
1|Subject[17]         3113.0    3357.0    1.0
1|Subject[18]         3407.0    2905.0    1.0
1|Subject[19]         4387.0    2846.0    1.0
1|Subject[20]         4822.0    2847.0    1.0
Days|Subject_sigma    1795.0    1833.0    1.0
Days|Subject[1]       2308.0    2869.0    1.0
Days|Subject[2]       2353.0    2916.0    1.0
Days|Subject[3]       2406.0    2581.0    1.0
Days|Subject[4]       2236.0    2547.0    1.0
Days|Subject[5]       2329.0    2432.0    1.0
Days|Subject[6]       2199.0    2523.0    1.0
Days|Subject[7]       2100.0    2487.0    1.0
Days|Subject[8]       2037.0    2511.0    1.0
Days|Subject[9]       2259.0    2863.0    1.0
Days|Subject[10]      2327.0    3050.0    1.0
Days|Subject[11]      2321.0    2433.0    1.0
Days|Subject[12]      2278.0    2746.0    1.0
Days|Subject[13]      2390.0    2580.0    1.0
Days|Subject[14]      2412.0    2859.0    1.0
Days|Subject[15]      2211.0    2356.0    1.0
Days|Subject[16]      2350.0    2411.0    1.0
Days|Subject[17]      1943.0    2483.0    1.0
Days|Subject[18]      2252.0    2489.0    1.0
Days|Subject[19]      3835.0    2637.0    1.0
Days|Subject[20]      4751.0    2873.0    1.0
Reaction_sigma        3556.0    2619.0    1.0
```

[34]:
```
az.plot_trace(trace_hier)
plt.show()
```

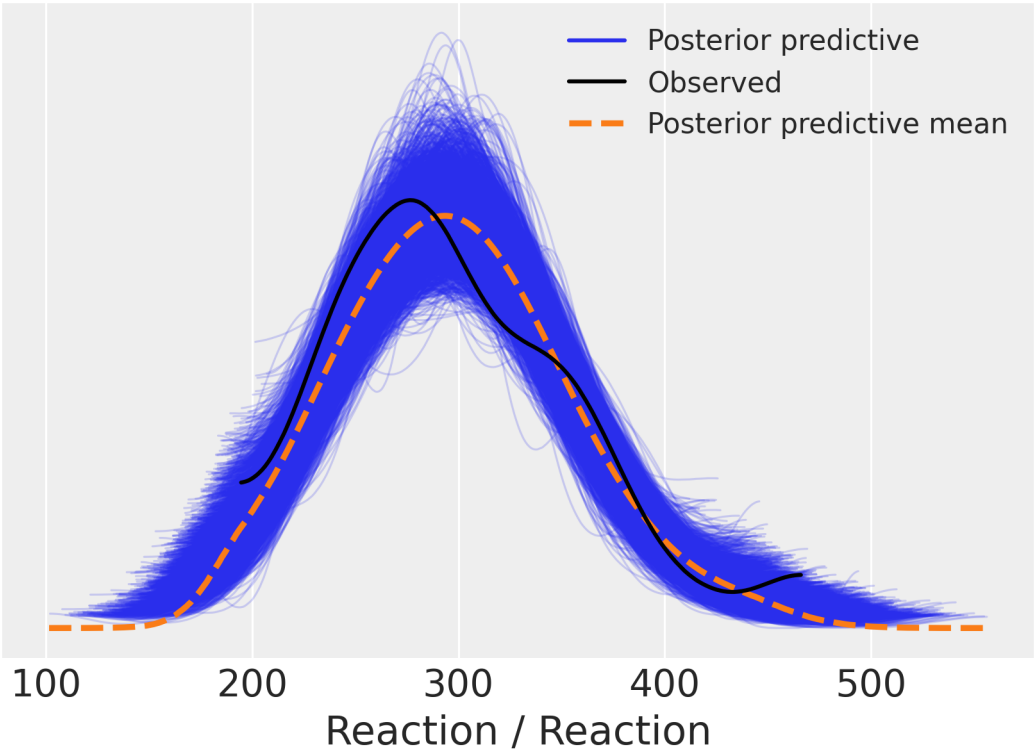
```
/home/cove/Lab/mcmc-talk/venv-mcmc/lib/python3.10/site-
packages/IPython/core/pylabtools.py:151: UserWarning: This figure was using
constrained_layout, but that is incompatible with subplots_adjust and/or
tight_layout; disabling constrained_layout.
  fig.canvas.print_figure(bytes_io, **kw)
```



3.4.4 Posterior Predictive ...

[35]:
```
az.plot_ppc(trace_hier)
```

[35]:
```
<AxesSubplot:xlabel='Reaction / Reaction'>
```



- Any suggestions for modifying our model?

4 Appendix: Model Comparison

There are many approaches to Bayesian model comparison, and full treatment of the options is outside the scope of this introduction. However, as a starting point, the library `arviz` implements a number of options for comparing models. For example, let's quickly compare the nonhierarchical to the hierarchical models:

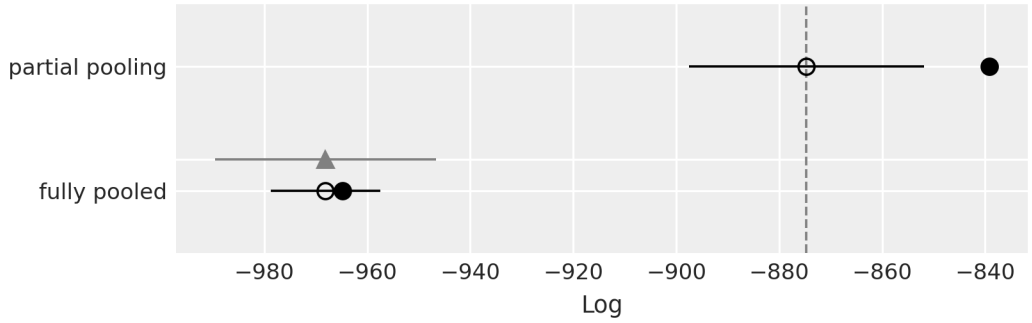
[36]:
```
df = az.compare({"fully pooled": trace_pooled, "partial pooling": trace_hier})
display(df)
az.plot_compare(df)
```

```
/home/cove/Lab/mcmc-talk/venv-mcmc/lib/python3.10/site-
packages/arviz/stats/stats.py:812: UserWarning: Estimated shape parameter of
Pareto distribution is greater than 0.7 for one or more samples. You should
consider using a more robust model, this is because importance sampling is less
```

```
likely to work well if the marginal posterior and LOO posterior are very
different. This is more likely to happen with a non-robust model and highly
influential observations.
  warnings.warn(

                rank         loo      p_loo      d_loo    weight         se  \
partial pooling    0 -874.742959  35.542741   0.000000  0.953069  22.832715
fully pooled       1 -968.121335   3.313175  93.378376  0.046931  10.623783

                     dse  warning loo_scale
partial pooling  0.000000     True       log
fully pooled    21.469243    False       log
```

[36]: <AxesSubplot:xlabel='Log'>

```
/home/cove/Lab/mcmc-talk/venv-mcmc/lib/python3.10/site-
packages/IPython/core/pylabtools.py:151: UserWarning: This figure was using
constrained_layout, but that is incompatible with subplots_adjust and/or
tight_layout; disabling constrained_layout.
  fig.canvas.print_figure(bytes_io, **kw)
```



In this case, a lower number is better, so it seems that our partial pooling model definitively outshines the fully pooled model.

To go deeper in Bayesian model comparison, refer to the following sources as a starting point:

- Leave-one-out cross validation (LOO-PSIS) and widely applicable information criterion (WAIC): Vehtari et al. (2017); link
- Projection predictive variable selection: Catalina et al. (2017); link, R library

References:

Vehtari, A., Gelman, A., & Gabry, J. (2017). Practical Bayesian model evaluation using leave-one-out cross-validation and WAIC. Statistics and computing, 27(5), 1413-1432.

Catalina, A., Bürkner, P. C., & Vehtari, A. (2022, May). Projection predictive inference for generalized linear and additive multilevel models. In International Conference on Artificial Intelligence and Statistics (pp. 4446-4461). PMLR.

[38]:
```
%load_ext watermark
%watermark -n -u -v -iv -w
```

```
Last updated: Tue May 31 2022

Python implementation: CPython
Python version       : 3.10.4
IPython version      : 8.4.0

arviz     : 0.12.1
numpy     : 1.21.6
IPython   : 8.4.0
pandas    : 1.4.2
matplotlib: 3.5.2
seaborn   : 0.11.2
bambi     : 0.8.0

Watermark: 2.3.1
```