What do we need from a probabilistic programming language to support Bayesian workflow?

BOB CARPENTER, Flatiron Institute, New York City

This talk is a survey of the model building and inference steps required for a probabilistic programming language to support a pragmatic Bayesian workflow. Pragmatic Bayesians eschew both of the opposing historical traditions of subjective Bayes (encode exactly your subjective beliefs as priors and turn the crank) or objective Bayes (devise "uninformative" priors and turn the crank). Instead, we follow a more agile methodology which combines exploratory data analysis with exploratory model fitting, analysis, and visualization, to produce calibrated and sharp inferences for unknown quantities of interest.

1 INTRODUCTION

Developing, coding, and applying Bayesian models is much like developing software. We need an agile development process with lots of feedback and fail-fast mechanisms Gelman et al. [2020].[1] In this paper, I'm going to zoom in on the inference steps required to support workflow and the implications for developing probabilistic programming languages.

The most widely used probabilistic programming languages for applied statistics, Stan and PyMC3, are not flexible enough to support workflow without rewriting models [Carpenter et al. 2017; Salvatier et al. 2016]. BUGS, one of the very first, if not the first, probabilistic programming language, provides just the right level of language flexibility to support workflow, but is lacking in language expressiveness and in support for efficient inference [Gilks et al. 1994; Lunn et al. 2012].

2 THE NITTY GRITTY OF BAYESIAN WORKFLOW

Gelman et al. [2013, Section 1.1] summarizes Bayesian data analysis as being about "practical methods for making inferences from data using probability models for quantities we observe and for quantities about which we wish to learn." It outlines the process as (1) set up a joint probability model with density $p(\theta, y)$ for all observable (y) and unobservable (θ) quantities, (2) condition on observed data to generate a posterior distribution with density $p(\theta \mid y)$, and (3) evaluate the fit of the model and the implications of the resulting posterior.

Gelman et al. [2020, Section 1.2] motivate expanding our scope beyond the inference step (2 above), because (a) computation can be challenging and fail, (b) we don't know what model we need to fit or even can fit with given data and that model will change as we gather more data, (c) we need to understand the fit of a model to data, and it helps to put this in perspective of more than one model, and (d) presenting multiple models that reach different substantive conclusions helps illustrate issues of model choice.

3 INFERENCE REQUIRED FOR BAYESIAN WORKFLOW

Bayesian workflow as envisioned by Gelman et al. [2020], involves multiple steps, as shown in Figure 1. This section explains these steps and the inference required for them from a probabilistic programming language.

3.1 Prior predictive checks

Prior predictive checks are used to evaluate model behavior before data is observed (i.e., where the posterior is equivalent to the prior) [Box 1980]. Given a joint density $p(\theta, y)$, prior predictive checks

[1] This work is being extended to an open-access book, https://github.com/jgabry/bayes-workflow-book

Author's address: Bob Carpenter, Flatiron Institute, New York City, bcarpenter@flatironinstitute.org.

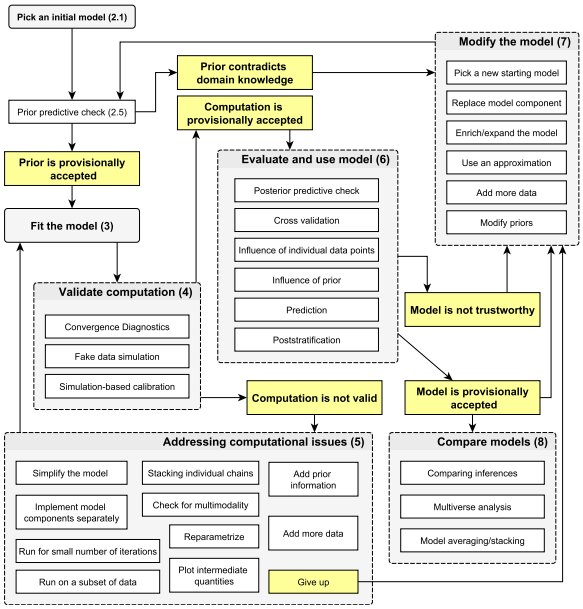

Fig. 1. *Overview of steps in Bayesian workflow. Each of the grey boxes involves forms of model fitting that benefit from probabilistic programming languages. Figure reproduced from Gelman et al. [2020, p. 5].*

require sampling from the marginal data distribution, $y^{\text{sim}} \sim p(y)$, where $p(y) = \int_{\mathbb{R}^N} p(y \mid \theta) \cdot p(\theta) \, \mathrm{d}\theta$ marginalizes out the parameters θ by averaging the sampling distribution $p(y \mid \theta)$ over the prior $p(\theta)$. Prior predictive checks evaluate whether the data generated from the prior is realistic by comparing it to actual data; see Gabry et al. [2019] for examples. The comparison is based on simulating multiple data sets $y^{\text{sim}(1)}, \ldots y^{\text{sim}(M)}$, computing a statistic $t(y^{\text{sim}(m)})$ for each simulation, and comparing the quantile of the same statistic $t(y)$ of the actual data. Any statistics of the data may be used, including central tendencies like mean and median, variance, skew, quantiles (including minimum or maximum values), etc. Unlike for posterior predictive checks (see below), we do not expect calibration at this stage.

3.2 Simulation-based calibration

It is common to generate fake data, either by choosing reasonable parameter values by hand or generating them from the prior. For example, to generate a fake data set $y^{\text{sim}} \sim p(y)$ from the prior predictive distribution, it suffices to draw $\theta^{\text{sim}} \sim p(\theta)$ from the prior and $y^{\text{sim}} \sim p(y \mid \theta^{\text{sim}})$ from the sampling distribution to produce a joint draw $y^{\text{sim}}, \theta^{\text{sim}} \sim p(y, \theta)$. Then we apply inference and take a sample of draws $\theta^{(m)} \sim p(\theta \mid y^{\text{sim}})$ from the posterior and then measure if the posterior for θ is consistent with the simulated value θ^{sim}.

To make this procedure more precise and automatic, we can use simulation-based calibration (SBC), which as its name implies, evaluates whether posteriors are properly calibrated [Cook et al. 2006; Talts et al. 2018]. Suppose we've drawn simulated data and parameters $y^{\text{sim}}, \theta^{\text{sim}} \sim p(y, \theta)$. Now if we take a posterior draw $\theta^{(m)} \sim p(\theta \mid y^{\text{sim}})$, the chain rule tells us that $y^{\text{sim}}, \theta^{(m)} \sim p(y, \theta)$ is also a draw from the joint distribution. If we take multiple posterior draws, the rank of of θ^{sim} among the $\theta^{(1)}, \ldots, \theta^{(M)}$ should be uniformly distributed over multiple simulations θ^{sim}. The result is a calibration test for posteriors given well-specified data drawn from the prior. SBC requires proper priors which are restrictive enough that draws from them do not cause numerical issues like

underflow or overflow; for example, if our predictors are unit scale and we draw logistic regression coefficients from a prior $\beta_k \sim \text{normal}(0, 100)$ or even worse $\beta_k \sim \text{cauchy}(0, 1)$, then we will almost certainly get draws that cause numerical overflow or underflow in evaluating the link function.

3.3 Posterior predictive checks

Posterior predictive checks (PPC) provide a Bayesian approach to goodness-of-fit evaluations [Gelman et al. 1996; Guttman 1967; Rubin 1984]. They are like prior predictive checks, but we check discrepancies in statistics between our observed data and posterior simulations conditioned on our observed data. Technically, we start with multiple draws $\theta^{\text{sim}(1)}, \ldots, \theta^{\text{sim}(M)} \sim p(\theta \mid y^{\text{obs}})$ from the posterior. For each of these draws, we simulate data from the sampling distribution, $y^{\text{sim}(m)} \sim p(y \mid \theta^{\text{sim}(m)})$. Together, this amounts to drawing from the posterior predictive distribution $y^{\text{sim}(m)} \sim p(y \mid y^{\text{obs}})$.

From a machine learning perspective, this is a kind of "cheating" in which we evaluate the fit to the "training" data, not to held out data. But don't worry, this isn't our final test. We just use this relatively cheap step to try to reject models before going on to check their held-out behavior (see the next section), because if they can't represent the data they're fit with, they're unlikely to fit held out data.

3.4 Cross-validation

Cross-validation evaluates how well a model is able to predict a subset of the data when fit to the complementary subset of data [Stone 1974]. In the limit, leave-one-out cross-validation evaluates how well a model predicts a single data item when fit with all of the other data items [Vehtari et al. 2017]. Cross-validation relies on being able to subdivide the data into two pieces $y^{\text{obs}} = y_1^{\text{obs}}, y_2^{\text{obs}}$, and evaluate (not sample from) the posterior predictive distribution, $p(y_1^{\text{obs}} \mid y_2^{\text{obs}})$. We can evaluate the posterior predictive distribution at a point using MCMC draws $\theta^{(m)} \sim p(\theta \mid y^{\text{obs}})$ with[2]

$$p(y_1^{\text{obs}} \mid y_2^{\text{obs}}) = \mathbb{E}[p(y_1^{\text{obs}} \mid \theta) \mid y_2^{\text{obs}}] = \int p(y_1^{\text{obs}} \mid \theta) \cdot p(\theta \mid y_2^{\text{obs}}) \, \mathrm{d}\theta \approx \tfrac{1}{M} \textstyle\sum_{m=1}^{M} p(y^{\text{obs}} 1 \mid \theta^{(m)}).$$

3.5 Sensitivity analysis

The certainty through which a parameter is known can be measured through the Bayesian posterior. But this is a model-based decision and there is further uncertainty due to choice of model components (likelihood and prior) and choice of constants like hyperparameters within a given model structure. Checking how much these decisions matter is a matter of sensitivity analysis (Oakley and O'Hagan [2004] provide a broad survey).

For example, suppose we have regression coefficients with a prior $\beta_k \sim \text{normal}(0, \sigma)$ and we've chosen to set $\sigma = 2$. We can carry out inference for $\sigma = 1$ or $\sigma = 4$ and see how the results change. Or we could replace the normal distribution with a Student-t distribution of the same variance and see what happens.

The traditional approach in applied mathematics is to compute a derivative (or second-order derivative) of the quantity of interest [Saltelli et al. 2004],

$$\frac{\partial}{\partial \sigma} \mathbb{E}[f(\alpha) \mid y, \sigma, \ldots] \Big|_{\sigma=2}.$$

We can evaluate sensitivity to data elements in the same way. Or we can evaluate fits with and without data items to measure their effect in the context of other data items. All of this can be carried out in a Bayesian framework using automatic differentiation, as shown by Giordano [2019].

[2] For numerical stability, this calculation needs to be carried out on the log scale where $\log p(y_1^{\text{obs}} \mid y_2^{\text{obs}}) \approx -\log M + \text{log\_sum\_exp}_{m=1}^{M} \log p(y_1^{\text{obs}} \mid \theta^{(m)})$, for draws $\theta^{(m)} \sim p(y \mid y_2^{\text{obs}})$.

3.6 Model comparison by calibration and sharpness

Assuming we have two models that pass all of our diagnostics, how do we compare them? The standard pragmatic Bayesian approach is to compare them in terms of calibration and sharpness [Gneiting et al. 2007]. Simulation-based calibration made sure inference for *parameters* was self-consistently calibrated. Now we want to make sure that inference for held out *data* is calibrated. Calibration is the analogue of unbiased estimation for Bayesian posteriors—we want our probability statements about future data to have the appropriate *frequentist* coverage [Dawid 1982]. Sharpness is the analogue of low-bias estimation; sharp estimates have low variance, or similarly, low entropy. For example, suppose I'm trying to compare 99% intervals for Republican vote share in the next presidential election. An interval of (0.489, 0.491) is sharper than (0.46, 0.50).

We can evaluate calibration and sharpness using so-called "proper scoring metrics", such as root mean squared error of estimations or in a probabilistic way using held-out log density [Dawid 2007; Gneiting and Raftery 2007]. Such evaluations are based on evaluating (not sampling) posterior predictive inferences $p(y^{\text{new}} \mid y^{\text{obs}})$ for new data given observed data.

We do not recommend using Bayes factors, because they are ratios of prior predictive densities rather than posterior predictive densities. For example, if we have two data models $p(y \mid M_1)$ and $p(y \mid M_2)$, then Bayes factors compare the ratio of $p(y^{\text{obs}} \mid M_1)$ to $p(y^{\text{obs}} \mid M_2)$ for some observed data y^{obs}. This averages the predictions for y^{obs} over the prior $p(\theta)$, with

$$p(y^{\text{obs}} \mid M) = \mathbb{E}[p(y^{\text{obs}} \mid \theta)] = \int p(y^{\text{obs}} \mid \theta) \cdot p(\theta) \, \mathrm{d}\theta.$$

We would rather focus on posterior predictive inference and evaluation on held out data y^{new}, which instead compares the posterior predictive densities of $p(y^{\text{new}} \mid y^{\text{obs}}, M_1)$ and $p(y^{\text{new}} \mid y^{\text{obs}}, M_2)$. These average predictions over the posterior $p(\theta \mid y^{\text{obs}})$ (for a model M_1, M_2, etc.), with

$$p(y^{\text{new}} \mid y^{\text{obs}}) = \mathbb{E}[p(y^{\text{new}} \mid \theta) \mid y^{\text{obs}}] = \int p(y^{\text{new}} \mid \theta) \cdot p(\theta \mid y^{\text{obs}}) \, \mathrm{d}\theta.$$

3.7 Summary of inference and modeling required for workflow

To fully support inference proposed for workflow, we need to be able to sample from the following marginal, joint and conditional distributions for any model $p(\theta, y)$ of interest.

- $y^{\text{sim}} \sim p(y)$ [prior predictive checks]
- $y^{\text{sim}}, \theta^{\text{sim}} \sim p(y, \theta)$ [simulation-based calibration]
- $\theta^{(m)} \sim p(\theta \mid y^{\text{sim}})$ [simulation-based calibration]
- $y^{\text{sim}} \sim p(y \mid y^{\text{obs}})$ [posterior predictive checks]

We also need to be able to evaluate the following quantity.

- $p(y_1^{\text{obs}} \mid y_2^{\text{obs}}) = \mathbb{E}[p(y_1^{\text{obs}} \mid \theta) \mid y_2^{\text{obs}}]$ [cross-validation]

To evaluate Bayes factors, we'd also need to be able to evaluate

- $p(y^{\text{obs}}) = \mathbb{E}[p(y^{\text{obs}} \mid \theta)] = \int p(y^{\text{obs}} \mid \theta) \cdot p(\theta) \, \mathrm{d}\theta.$

Furthermore, all of the ways in which we might modify a model need to be accommodated somehow. I dream of a workbench that keeps track of the choices I make as a I go so that they can be easily navigated and later included in a final presentation. As is, I wind up with file names like `irt-2pl.stan`, `irt-2pl-hier.stan`, `irt-2pl-std-norm-student-zero-centered-hier-difficulty.stan`, and so on until it's impossible to keep track of which model is doing what.

4 REMEDIATING PROBLEMS DURING WORKFLOW

When computational or statistical problems arise from models, there are various steps we might take to mitigate them. These are outlined in the "modify the model" box of the workflow diagram in Figure 1. At the most radical, we might just throw out a model and start again. But typically we

work incrementally from an existing simple model. For example, we may suspect we're getting unrealistically wide posteriors for a parameter because we have a prior that's too vague and don't have enough data to pin it down. We might try to solve this problem by including a more informative prior either based on prior knowledge or by adding a meta-analysis component. Or we might relax a Poisson sampling distribution to a negative binomial distribution to account for under-dispersed inferences from a Poisson model. Or we might split a single pooled parameter into multiple parameters that vary by individual, or vice versa. Or we might take a parameter and set a value for it as data or set two parameters to be the same (i.e., "clamping," "pinning," or "tying" parameters.) As is clear from the examples, changes aren't restricted to priors—the prior and likelihood are inextricably linked in Bayesian models and cannot be understood independently [Gelman et al. 2017]; even prior predictive checks combine the two distributions.

5 WORKFLOW SUPPORT IN EXISTING PPLS

So how well do our probabilistic programming languages do at supporting the operations we need for Bayesian workflow? The workflow steps require moving flexibly between prior simulation of data, posterior simulation of parameters, and posterior predictive simulation of data or evaluation of log densities. The bad news is that all of the systems of which I am aware (by no means all of them) come up short on these measures; the good news is that'll keep us employed improving our existing PPLs and building better ones.

BUGS is one of the earliest PPLs and provides excellent support for Bayesian workflow. Rather than define a programming language per se, BUGS defines a directed graphical modeling language with stochastic nodes for probabilistic models and deterministic nodes for transforms. The original BUGS [Gilks et al. 1994; Lunn et al. 2012] has inspired related packages that make the same graphical modeling assumptions, the standalone JAGS [Plummer et al. 2003] and the R-embedded NIMBLE [de Valpine et al. 2017]. Consider the following model, which I borrowed with light modifications from the JAGS documentation.

```
for (n in 1:N) { mu[n] <- alpha + beta * x[n];  y[n] ~ dnorm(mu[i], tau); }
alpha ~ dnorm(0, 1e-4);  beta ~ dnorm(0, 1e-4);  tau ~ dgamma(1e-3, 1e-3);
```

The decision as to what is observed is made at run time, even to the extent of allowing "missing data" within a vector. For example, we can perform posterior predictive sampling for y^{new} by setting $y = y^{\text{obs}}, y^{\text{new}}$ with an indication that y^{new} are unknown (BUGS uses the R notation NA). The only place that BUGS falls down is in not supporting cross-validation, because there's no easy way to evaluate a log density and set it to a value, so we can't evaluate $p(y_1^{\text{obs}} \mid y_2^{\text{obs}})$. BUGS also supports clamping of parameters to fixed values, because it's nothing more than giving a variable a value as part of data. The main drawback to using BUGS is that it's restricted to inefficient Gibbs sampling.

Stan supports all of the inferences and evaluations required for workflow, but the model has to be rewritten for most use cases because the data versus parameter status of a variable must be declared at compile time [Carpenter et al. 2017]. SlicStan points the way to more flexibility, but as defined still requires parameters to be declared as such in programs [Gorinova et al. 2019]. Clamping a variable to a value requires moving a declaration from the parameter block to the data block and recompiling. Although the need to marginalize discrete parameters provides huge benefits for inferential efficiency, robustness, and tail statistics [Stan Development Team 2021, Chapter 7], it can be difficult for users and would ideally be automated [Gorinova et al. 2020]. Stan's block-based structure also impedes coding reusable components involving parameters, data, and density contributions (Gorinova et al. [2019] sketches a partial remedy).

PyMC3 is a graphical modeling language like BUGS [Salvatier et al. 2016] rather than an imperative language like Stan. But it has much better support for inference through the

no-U-turn sampler [Hoffman and Gelman 2014]. Unfortunately, it has the same drawback as Stan in that the declaration of which variables are observed is built into their declarations. The very first model on their home page (https://docs.pymc.io) provides an example.

```
with pm.Model() as linear_model:
    y_obs = pm.Normal("y_obs", mu=X @ weights, sigma=noise, observed=y)
```

While we could have made `y` a free variable bound on the outside, we can't get around the fact that the distribution statement for `y_obs` bakes in the fact that it is observed with `observed=y`. PyMC3 allows discrete parameters, but this can be problematic because of the challenge of discrete sampling.

ADMB was the first autodiff-based probabilistic programming language [Fournier et al. 2012]. Like Stan and PyMC3, variables are declared in the program as to whether they are data or parameters (see, e.g., https://github.com/admb-project/admb/tree/master/examples/admb).

Pyro is more like BUGS in that users define Python functions for models, then use additional Python statements like `poutine.condition(model, data={"obs": y})(sigma)` in order to condition on observed data [Bingham et al. 2019]. In this example, `y` and `sigma` are a data scalar and vector with actual values.

Edward2 is like Pyro in that it sets up a Python object representing a model, which may then be inspected and manipulated in different ways [Moore and Gorinova 2018; Tran et al. 2018]. For example, their GitHub home page (https://github.com/google/edward2) has an example where the model `logistic_regression` is instantiated in two stages using

```
log_joint = ed.make_log_joint_fn(logistic_regression) def
  target_log_prob_fn(c, i): log_joint(features, coeffs=c, intercept=i, outcomes=o)
```

where the coefficients and outcomes are defined as function arguments to be sampled and the features and outcomes are set from the outside to actual values.

Turing.jl is a PPL embedded in Julia. It shares the nice property of BUGS that lets you declare which variables are observed flexibly (albeit at compile time), but doing so is tangled with which variables are sampled and must be included in the function defining the model (see the example on https://turing.ml/dev/docs/using-turing/guide). It also adopts a more flexible version of autodiff with respect to its containing language than the Python-based systems, all of which are forced to "pun" (aka overload) NumPy and restrict language features.

Oryx is like many of the embedded PPLs in that it serves as more of a lightweight library for random variables on top of JAX than a standalone PPL [Bradbury et al. 2018]. It provides useful tools such as `intervene()` for clamping parameter values and `nest()` for embedding subprograms modularly and the ability to neatly invert unconstraining variable transforms with Jacobian adjustments. Functions like `condition()` make it clear that things like data vs. parameter distinctions can be left until run time.

6 CONCLUSION

Given the recent boom in probabilistic programming, this survey could only scratch the surface of some of the most popular PPLs. Hopefully there will be time during the workshop to discuss other languages and their support for workflow, including approaches such as Infer.NET [Minka et al. 2018] that are based on approximations and the Church family (including Anglican, Venture, and WebPPL), which reason over program states [Goodman et al. 2012; Goodman 2013].

ACKNOWLEDGMENTS

I'd like to thank my co-authors on Bayesian workflow, Andrew Gelman, Aki Vehtari, Dan Simpson, Charles Margossian, Yuling Yao, Lauren Kennedy, Jonah Gabry, Paul Bürkner, and Martin Modrák.

REFERENCES

Eli Bingham, Jonathan P Chen, Martin Jankowiak, Fritz Obermeyer, Neeraj Pradhan, Theofanis Karaletsos, Rohit Singh, Paul Szerlip, Paul Horsfall, and Noah D Goodman. 2019. Pyro: Deep universal probabilistic programming. *The Journal of Machine Learning Research* 20, 1 (2019), 973–978.

George EP Box. 1980. Sampling and Bayes' inference in scientific modelling and robustness. *Journal of the Royal Statistical Society: Series A (General)* 143, 4 (1980), 383–404.

James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. 2018. JAX: composable transformations of Python+NumPy programs. http://github.com/google/jax

Bob Carpenter, Andrew Gelman, Matthew D Hoffman, Daniel Lee, Ben Goodrich, Michael Betancourt, Marcus A Brubaker, Jiqiang Guo, Peter Li, and Allen Riddell. 2017. Stan: a probabilistic programming language. *Journal of Statistical Software* 76, 1 (2017), 1–32.

Samantha R Cook, Andrew Gelman, and Donald B Rubin. 2006. Validation of software for Bayesian models using posterior quantiles. *Journal of Computational and Graphical Statistics* 15, 3 (2006), 675–692.

A Philip Dawid. 1982. The well-calibrated Bayesian. *J. Amer. Statist. Assoc.* 77, 379 (1982), 605–610.

A Philip Dawid. 2007. The geometry of proper scoring rules. *Annals of the Institute of Statistical Mathematics* 59, 1 (2007), 77–93.

Perry de Valpine, Daniel Turek, Christopher J Paciorek, Clifford Anderson-Bergman, Duncan Temple Lang, and Rastislav Bodik. 2017. Programming with models: writing statistical algorithms for general model structures with NIMBLE. *Journal of Computational and Graphical Statistics* 26, 2 (2017), 403–413.

David A Fournier, Hans J Skaug, Johnoel Ancheta, James Ianelli, Arni Magnusson, Mark N Maunder, Anders Nielsen, and John Sibert. 2012. AD Model Builder: using automatic differentiation for statistical inference of highly parameterized complex nonlinear models. *Optimization Methods and Software* 27, 2 (2012), 233–249.

Jonah Gabry, Daniel Simpson, Aki Vehtari, Michael Betancourt, and Andrew Gelman. 2019. Visualization in Bayesian workflow. *Journal of the Royal Statistical Society: Series A (Statistics in Society)* 182, 2 (2019), 389–402.

Andrew Gelman, John B Carlin, Hal S Stern, David B Dunson, Aki Vehtari, and Donald B Rubin. 2013. *Bayesian Data Analysis* (third ed.). Chapman & Hall/CRC Press.

Andrew Gelman, Xiao-Li Meng, and Hal Stern. 1996. Posterior predictive assessment of model fitness via realized discrepancies. *Statistica sinica* (1996), 733–760.

Andrew Gelman, Daniel Simpson, and Michael Betancourt. 2017. The prior can often only be understood in the context of the likelihood. *Entropy* 19, 10 (2017), 555.

Andrew Gelman, Aki Vehtari, Daniel Simpson, Charles C Margossian, Bob Carpenter, Yuling Yao, Lauren Kennedy, Jonah Gabry, Paul-Christian Bürkner, and Martin Modrák. 2020. Bayesian workflow. *arXiv preprint arXiv:2011.01808* (2020).

Wally R Gilks, Andrew Thomas, and David J Spiegelhalter. 1994. A language and program for complex Bayesian modelling. *Journal of the Royal Statistical Society: Series D (The Statistician)* 43, 1 (1994), 169–177.

Ryan James Giordano. 2019. *On the Local Sensitivity of M-Estimation: Bayesian and Frequentist Applications.* Ph.D. Dissertation. UC Berkeley.

Tilmann Gneiting, Fadoua Balabdaoui, and Adrian E Raftery. 2007. Probabilistic forecasts, calibration and sharpness. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)* 69, 2 (2007), 243–268.

Tilmann Gneiting and Adrian E Raftery. 2007. Strictly proper scoring rules, prediction, and estimation. *J. Amer. Statist. Assoc.* 102, 477 (2007), 359–378.

Noah Goodman, Vikash Mansinghka, Daniel M Roy, Keith Bonawitz, and Joshua B Tenenbaum. 2012. Church: a language for generative models. *arXiv preprint arXiv:1206.3255* (2012).

Noah D Goodman. 2013. The principles and practice of probabilistic programming. *ACM SIGPLAN Notices* 48, 1 (2013), 399–402.

Maria Gorinova, Dave Moore, and Matthew Hoffman. 2020. Automatic reparameterisation of probabilistic programs. In *International Conference on Machine Learning*. PMLR, 3648–3657.

Maria I Gorinova, Andrew D Gordon, and Charles Sutton. 2019. Probabilistic programming with densities in SlicStan: efficient, flexible, and deterministic. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 1–30.

Irwin Guttman. 1967. The use of the concept of a future observation in goodness-of-fit problems. *Journal of the Royal Statistical Society: Series B (Methodological)* 29, 1 (1967), 83–100.

Matthew D Hoffman and Andrew Gelman. 2014. The No-U-Turn sampler: adaptively setting path lengths in Hamiltonian Monte Carlo. *J. Mach. Learn. Res.* 15, 1 (2014), 1593–1623.

David Lunn, Chris Jackson, Nicky Best, Andrew Thomas, and David Spiegelhalter. 2012. *The BUGS book: A practical introduction to Bayesian analysis.* CRC press.

T. Minka, J.M. Winn, J.P. Guiver, Y. Zaykov, D. Fabian, and J. Bronskill. 2018. /Infer.NET 0.3. Microsoft Research Cambridge. http://dotnet.github.io/infer.

Dave Moore and Maria I Gorinova. 2018. Effect handling for composable program transformations in Edward2. *arXiv preprint arXiv:1811.06150* (2018).

Jeremy E Oakley and Anthony O'Hagan. 2004. Probabilistic sensitivity analysis of complex models: a Bayesian approach. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)* 66, 3 (2004), 751–769.

Martyn Plummer et al. 2003. JAGS: A program for analysis of Bayesian graphical models using Gibbs sampling. In *Proceedings of the 3rd international workshop on distributed statistical computing*, Vol. 124. Vienna, Austria., 1–10.

Donald B Rubin. 1984. Bayesianly justifiable and relevant frequency calculations for the applied statistician. *The Annals of Statistics* (1984), 1151–1172.

Andrea Saltelli, Stefano Tarantola, Francesca Campolongo, and Marco Ratto. 2004. *Sensitivity analysis in practice: a guide to assessing scientific models*. Vol. 1. Wiley Online Library.

John Salvatier, Thomas V Wiecki, and Christopher Fonnesbeck. 2016. Probabilistic programming in Python using PyMC3. *PeerJ Computer Science* 2 (2016), e55.

Stan Development Team. 2021. Stan User's Guide. https://mc-stan.org/documentation

Mervyn Stone. 1974. Cross-validatory choice and assessment of statistical predictions. *Journal of the Royal Statistical Society: Series B (Methodological)* 36, 2 (1974), 111–133.

Sean Talts, Michael Betancourt, Daniel Simpson, Aki Vehtari, and Andrew Gelman. 2018. Validating Bayesian inference algorithms with simulation-based calibration. *arXiv preprint arXiv:1804.06788* (2018).

Dustin Tran, Matthew Hoffman, Dave Moore, Christopher Suter, Srinivas Vasudevan, Alexey Radul, Matthew Johnson, and Rif A Saurous. 2018. Simple, distributed, and accelerated probabilistic programming. *arXiv preprint arXiv:1811.02091* (2018).

Aki Vehtari, Andrew Gelman, and Jonah Gabry. 2017. Practical Bayesian model evaluation using leave-one-out cross-validation and WAIC. *Statistics and Computing* 27, 5 (2017), 1413–1432.