Beyond Bayesian Model Averaging over Paths in Probabilistic Programs with Stochastic Support

Tim Reichelt
University of Oxford

Luke Ong
Nanyang Technological University

Tom Rainforth
University of Oxford

Abstract

The posterior in probabilistic programs with stochastic support decomposes as a weighted sum of the local posterior distributions associated with each possible program path. We show that making predictions with this full posterior implicitly performs a Bayesian model averaging (BMA) over paths. This is potentially problematic, as BMA weights can be unstable due to model misspecification or inference approximations, leading to suboptimal predictions in turn. To remedy this issue, we propose alternative mechanisms for path weighting: one based on *stacking* and one based on ideas from *PAC-Bayes*. We show how both can be implemented as a cheap post-processing step on top of existing inference engines. In our experiments, we find them to be more robust and lead to better predictions compared to the default BMA weights.

1 INTRODUCTION

Universal probabilistic programming systems (PPS) (Tolpin et al., 2016; Goodman et al., 2008; Bingham et al., 2019; Ge et al., 2018; Mansinghka et al., 2014) provide flexible frameworks for expressing powerful probabilistic models, along with tools to aid performing inference in them. By permitting branching on the outcomes of sampling statements, they allow users to express programs with *stochastic support*, wherein the number of latent variables varies between program executions, leading to challenging inference problems.

Such programs can be thought of as a combination of independent sub-programs, each with static support, known as straight-line programs (SLP) (Chaganty et al., 2013; Sankaranarayanan et al., 2013; Luo et al., 2021). The overall posterior is then given by the weighted sum of individual SLP posteriors, with weights corresponding to the local normalization constants of the SLPs; a breakdown recent work has exploited to improve inference (Zhou et al., 2020; Reichelt et al., 2022a).

We show that this decomposition also reveals that the posterior of *any* program with stochastic support is a Bayesian Model Averaging (BMA) (Hoeting et al., 1999) over the constituent SLPs of the program. Thus, all PPS inference engines are implicitly estimating a BMA when the program has stochastic support, whether they explicitly account for this or not.

However, it is widely acknowledged in the Bayesian statistics literature that BMA can be a problematic mechanism for combining the posteriors of individual models (Minka, 2000; Yao et al., 2018), with alternatives often preferred in practice, especially when our aim is to make good predictions. In particular, BMA often performs poorly under *model misspecification* (Gelman and Yao, 2020; Oelrich et al., 2020), wherein it tends to produce *overconfident* posterior model weights that collapse towards a single model (Huggins and Miller, 2021; Yang and Zhu, 2018). Given that models will rarely be perfect when working with real data (Box, 1976; Key et al., 1999; Vehtari and Ojanen, 2012), this is a serious practical concern that has been observed to cause notable issues in many applied fields (Yang and Zhu, 2018; Smets and Wouters, 2007; Leff et al., 2008).

We argue that PPSs need to account for these shortfalls and provide access to more robust weighting schemes. To provide such alternatives, we suggest optimizing the SLP weights for *predictive performance*. Specifically, we introduce weighting schemes based on *stacking of predictive distributions* (Wolpert, 1992; Breiman, 1996; LeBlanc and Tibshirani, 1996; Yao et al., 2018) and *PAC-Bayes objectives* (Masegosa, 2020; Masiha et al., 2021; Morningstar et al., 2022; Alquier, 2023). We show how to run them as a cheap post-processing step on the outputs of any sample-based inference scheme, and demonstrate that they provide more robust weights with better predictive performance.

Correspondence to `reichelt@robots.ox.ac.uk`. Proceedings of the 27th International Conference on Artificial Intelligence and Statistics (AISTATS) 2024, Valencia, Spain. PMLR: Volume 238.

Our contributions are: (a) By interpreting the posterior in programs with stochastic support as a BMA, we show that the weights assigned to SLPs can be unstable, e.g. due to model misspecificiation. (b) Providing a general scheme to adapt PPS inference algorithms to utilize alternative weighting schemes, with an implementation in Pyro (Bingham et al., 2019). (c) Investigating their behaviour for a variety of different programs and showing its benefits on synthetic and real-world data.

2 BACKGROUND

2.1 Bayesian Model Averaging

An important question in Bayesian statistics is how to best combine the inferences of different possible models. In a pure Bayesian framework, this is done by weighting the model posteriors according to their *posterior model probability*, leading to a framework called Bayesian model averaging (BMA) (Hoeting et al., 1999).

To be more precise, assume we have a countable set of Bayesian models indexed by k, each with corresponding latent parameters $\theta_k \in \Theta_k$, prior $p_k(\theta_k)$, and likelihood $p_k(y|\theta_k)$, where y is data we want to condition on.[1] In BMA, we set a prior over which model generated the data, $p(M = k)$, from which we can derive the posterior model probability

$$p(M{=}k \mid y) \propto p(y \mid M{=}k)\, p(M{=}k) \tag{1}$$

where $p(y \mid M{=}k) = \int p_k(y|\theta_k)\, p_k(\theta_k) d\theta_k$ is the *model evidence*, or marginal likelihood, for the kth model.

Predictions and expectations can now be calculated by combining those from individual models using $p(M{=}k \mid y)$ as weights. In particular, the posterior predictive distribution for new hypothetical data, $\tilde{y}$, is given by

$$p(\tilde{y} \mid y) = \sum\nolimits_k p(M{=}k \mid y)\, \mathbb{E}_{p_k(\theta_k|y)}\left[p_k(\tilde{y}|\theta_k)\right], \tag{2}$$

where $p_k(\theta_k|y) \propto p_k(\theta_k)p_k(y|\theta_k)$ and $p_k(\tilde{y}|\theta_k)$ are the local posterior and local parameterized predictive distribution, respectively.

Criticisms of BMA In practice, our models will never be able to capture the full complexities of the real world as, in the words of George Box, "all models are wrong, some are useful" (Box, 1976). It is therefore important to investigate the behaviour of frameworks when our models are misspecified, that is when $\nexists\ (\theta_k, k) : p_k(y|\theta_k) = p_{\text{true}}(y)\ \forall y$, where $p_{\text{true}}(y)$ is the (unknown) true data generating distribution.

Crucially, BMA implicitly assumes that the data was sampled from exactly one of the constituent models. This is often referred to as the $\mathcal{M}$-closed assumption (Bernardo and Smith, 2009; Clyde and Iversen, 2013; Key et al., 1999). As a result, as the amount of data increases the BMA weights will always (except for a few special edge cases) collapse on a single model (Clyde and Iversen, 2013); the approach reverts to just performing model selection. Consequently, BMA predictions are often inferior compared to other model combination techniques (Minka, 2000; Yao et al., 2018).

Viewed another way, model misspecification tends to lead to posterior model probabilities that are *overconfident*: both empirical and theoretical results have shown that they too readily collapse on a single model (Huggins and Miller, 2021; Yang and Zhu, 2018), even when there are multiple plausible models with similar predictive performance. Moreover, the exact model onto which the posterior collapses can change drastically when regenerating the data from $p_{\text{true}}(y)$. In general, we expect there to be some variance in the BMA weights due to the fact that we need to *estimate* the model evidence for many real-world models. However, previous work has demonstrated that overconfidence is an issue even with *analytic* BMA weights (Yang and Zhu, 2018; Huggins and Miller, 2021; Oelrich et al., 2020).

2.2 Programs with Stochastic Support

A probabilistic program can be interpreted as defining an *unnormalized density function* $\gamma : \Theta \to \mathbb{R}^{\geq 0}$, where Θ denotes the sample space of the latent variables in the program (Borgström et al., 2016; Staton et al., 2016). These variables are typically defined as the outcomes of random sampling statements and each sample statement is associated with a unique lexical address. The goal of inference is then to find a representation of the normalized program density $\pi(\theta) = \gamma(\theta)/\int \gamma(\theta) d\theta$, where $d\theta$ is an implicitly defined reference measure (Gordon et al., 2014; Rainforth, 2017; van de Meent et al., 2018). One can informally think of $\pi(\theta)$ as a posterior distribution, $p(\theta|y)$. See App. A for a more detailed introduction.

Universal PPS allow users to branch on the outcomes of random sampling statements leading to programs with *stochastic support*. An important property of such programs is that they can be decomposed into (a countable number of) straight-line programs (SLPs), sub-programs without any control flow (Chaganty et al., 2013; Sankaranarayanan et al., 2013; Zhou et al., 2020; Luo et al., 2021; Reichelt et al., 2022a). These SLPs are effectively the different possible control-flow paths that exist in the program and they are defined by their *address path*, i.e. the sequence of the lexical addresses encountered during the program's execution.

Each SLP corresponds to a disjoint sub-region, Θ_k, of the overall sample space (such that $\Theta = \bigcup_k \Theta_k$) and has the *local unnormalized density* $\gamma_k(\theta) := \mathbb{I}\left[\theta \in \Theta_k\right] \gamma(\theta)$.

[1] Note our formulations apply equally when there are also inputs the model is conditioned on, i.e. we have $p_k(y|\theta_k, x)$, but we negate this from our notation to avoid clutter.

The unnormalized density for the whole program can thus be written as $\gamma(\theta) = \sum_k \gamma_k(\theta)$. Similarly, the normalized program density can be rewritten as

$$\pi(\theta) = \sum\nolimits_k \left(Z_k / \sum\nolimits_\ell Z_\ell \right) \pi_k(\theta), \tag{3}$$

where $Z_k = \int \gamma_k(\theta) d\theta$ and $\pi_k(\theta) = \gamma_k(\theta)/Z_k$ are the *local normalization constant* and *local posterior* respectively. We refer to $\pi(\theta)$ as the *full Bayes posterior*. Note that the disjoint supports of the SLPs means that there exists exactly one $k : \pi_k(\theta) > 0$ for any given θ.

3 INFERENCE IN STOCHASTIC SUPPORT PROGRAMS IS BMA

Examining Eq. (3), we immediately see that $\pi(\theta)$ is a weighted sum of localized posteriors. This decomposition also reveals that using the full Bayes posterior to calculate predictions or expectations is implicitly performing a BMA over the individual SLPs. To see this, consider calculating the expectation of some parameterized predictive density $p(\tilde{y}|\theta)$:

$$\mathbb{E}_{\pi(\theta)} \left[p(\tilde{y}|\theta) \right] = \sum_k \frac{Z_k}{\sum_\ell Z_\ell} \mathbb{E}_{\theta_k \sim \pi_k} \left[p_k(\tilde{y}|\theta_k) \right], \tag{4}$$

where p_k is any conditional density function such that $p_k(\tilde{y}|\theta) = p(\tilde{y}|\theta) \, \forall \tilde{y}, \theta \in \Theta_k$, and we have defined new random variables θ_k drawn from the local posterior of the kth SLP. We thus have that the downstream posterior predictive on $\tilde{y}$ is a weighted sum of the posterior predictives that would result from using the kth SLP instead of our full program.

There is now a clear analog between Eq. (4) and Eq. (2). To show that the former corresponds to a BMA, all that remains is to show that the weights can be interpreted as posterior model probabilities. At a high-level, this follows simply from the fact that the Z_k are analogous to (unnormalized) posterior model probabilities (note, though, they are *not* analogous to the model evidences).

To be more precise, consider the factorization $\gamma(\theta) = f(\theta)g(\theta)$ where $f(\theta)$ corresponds to all terms from the sampling and $g(\theta)$ all terms from conditioning statements, such that we can think of them as prior and likelihood components respectively. The prior probability of choosing the kth SLP is now given by $P_k := \int f(\theta) \mathbb{I}\left[\theta \in \Theta_k\right] d\theta$, while the "model evidence" is

$$E_k := \int \frac{f(\theta) \mathbb{I}\left[\theta \in \Theta_k\right]}{P_k} g(\theta) d\theta. \tag{5}$$

The posterior model probability is then equal to

$$\frac{P_k E_k}{\sum_\ell P_\ell E_\ell} = \frac{\int f(\theta) \mathbb{I}\left[\theta \in \Theta_k\right] g(\theta) d\theta}{\sum_\ell \int f(\theta) \mathbb{I}\left[\theta \in \Theta_\ell\right] g(\theta) d\theta} = \frac{Z_k}{\sum_\ell Z_\ell}. \tag{6}$$

Thus Eq. (4) is a BMA with model prior $p(M = k) = P_k$, local posteriors $p_k(\theta_k|y) = \pi_k(\theta_k)$, model evidences $p(y|M = k) = E_k$, and identical local parameterized predictive distributions $p_k(\tilde{y}|\theta_k) = p(\tilde{y}|\theta = \theta_k)$.

Having realized that using the full Bayes posterior leads to BMA, we can instead define a generalized model averaging scheme over the SLPs that is explicitly parameterized by a learnable set of weights, w:

$$\tilde{\pi}(\theta; w) := \sum\nolimits_k w_k \, \pi_k(\theta) \tag{7}$$

with $\sum_k w_k = 1, w_k \geq 0$. This opens the door for considering alternative approaches that avoid the shortfalls of BMA. We refer to $w_k \propto Z_k$ as the *BMA weights*; the choice made by all current inference engines.

At this point, a critical reader might argue that all Bayesian inference in general, and not just the weights in a BMA, is sensitive to model misspecification. However, averaging over a finite discrete set of models has been highlighted as a special case in which Bayesian inference can give counter-intuitive results and is especially susceptible to misspecification (Yao et al., 2018; Gelman and Yao, 2020; Oelrich et al., 2020). Additionally, in most realistic models used in practice we need to estimate the posterior and the local normalization constants. As we will show in our experiments in Sec. 6 this can be an additional source of variance leading to sub-optimal predictions. This thus motivates treating the SLP weights as explicit parameters that we may wish to set in a non-Bayesian manner, while leaving the local posteriors unchanged.

4 BEYOND BMA PATH WEIGHTS

There are different ways we can choose the SLP weights, w_k, in Eq. (7), with BMA only one possible choice. A simple, albeit crude, alternative would be to just set them equally. While we find that this can sometimes empirically outperform the BMA weights (see Sec. 6), it is clearly not an appropriate general-purpose solution and there are cases where it can perform very poorly.

To provide more principled alternatives, we now show how the weights can be optimized to maximize predictive performance. For the purpose of exposition, we will introduce the main ideas through the eyes of *stacking* (Yao et al., 2018; Wolpert, 1992; Breiman, 1996; LeBlanc and Tibshirani, 1996) but we will show in Sec. 4.4 how we can also use PAC-Bayes objectives (Morningstar et al., 2022) to fit the SLP weights.

4.1 Stacking Objective For PPSs

The goal of stacking is to improve predictions by optimizing the model weights, w. To achieve this, we need to define a method for making predictions for a hypothetical new observation, which we denote as $\tilde{y} \in \mathcal{Y}$. Just as we previously defined a generalized version of the posterior in Eq. (7), we now need to establish a generalized version of the posterior predictive.

For simplicity, we will assume for now that an explicit predictive density, $p(\cdot \mid \cdot) : \mathcal{Y} \times \Theta \to \mathbb{R}^{\geq 0}$, has been provided, before showing how this can instead be derived from the program itself in Sec. 4.2. This ensures for each SLP we have a *local posterior predictive density*

$$\rho_k(\tilde{y}) := \mathbb{E}_{\pi_k(\theta)}\left[p(\tilde{y} \mid \theta)\right]. \tag{8}$$

With this, we can define the *stacked predictive density*

$$\tilde{\rho}(\tilde{y}; w) := \mathbb{E}_{\tilde{\pi}(\theta; w)}\left[p(\tilde{y} \mid \theta)\right] = \sum\nolimits_k w_k\, \rho_k(\tilde{y}) \tag{9}$$

In its most general form, stacking defines an objective with a user-defined scoring rule $S(\tilde{\rho}, \tilde{y})$ which takes as input a predictive distribution and a data point (Gneiting and Raftery (2007); c.f. App. C). It then optimizes the weights, w, to maximize the expected score

$$R(w; S) := \mathbb{E}_{p_{\text{true}}(\tilde{y})}\left[S(\tilde{\rho}(\cdot \mid w), \tilde{y})\right]. \tag{10}$$

We will focus on using the logarithmic score rule, as it is by far the most popular one used in practice, yielding

$$R(w) := \mathbb{E}_{p_{\text{true}}(\tilde{y})}\left[\log\left(\sum\nolimits_k w_k\, \rho_k(\tilde{y})\right)\right]. \tag{11}$$

As $\mathbb{E}_{p_{\text{true}}(\tilde{y})}\left[\log p_{\text{true}}(\tilde{y})\right]$ is a constant, maximizing $R(w)$ is equivalent to minimizing $\mathrm{KL}(p_{\text{true}}(\tilde{y}) \parallel \tilde{\rho}(\tilde{y}; w))$.

We now need a mechanism to estimate the expectation w.r.t. $p_{\text{true}}(\tilde{y})$ in Eq. (11). Multiple strategies for this exist (Vehtari and Ojanen, 2012). We will first show how to do so using an explicit validation set, $\{\tilde{y}_\ell\}_{\ell=1}^L$, before describing how to avoid the use of a validation set for a broad class of models in Sec. 4.3.

As an aside, the stacking weights should be interpreted differently from BMA weights. In BMA, the weight of the kth SLP represents the posterior probability that the data was generated from the kth SLP. In contrast, stacking generates a mixture of the local posterior predictive distributions and optimizes the mixture weights on held-out data. Hence, the stacking weight for the kth SLP estimates the probability that a new data point is drawn from the kth SLP.

4.2 Stacking as Post-Processing

Given (normalised) weighted samples $\{(v_s, \theta_s)\}_{s=1}^S$ generated from an arbitrary inference algorithm, the posterior of the program is approximated by the empirical measure $\hat{\pi}(\theta) = \sum_{s=1}^S v_s\, \delta_{\theta_s}(\theta)$; unweighted sampling schemes correspond to the special case $v_s = 1/S$. The local posteriors of the kth SLP are consequently approximated by all the samples which fall into the kth SLP, i.e. $\hat{\pi}_k(\theta) := \sum_{s \in I_k} (v_s / V_k)\, \delta_{\theta_s}(\theta)$ where $I_k := \{s \in \{1, \ldots, S\} \mid \theta_s \in \Theta_k\}$ are the indices of the samples from the kth SLP and $V_k := \sum_{s \in I_k} v_s$ is the sum of all the associated sample weights. Recall from Sec. 2.2 that the SLP of a sample θ_s is determined by its address path. Thus, we can generate the index sets I_k by grouping all samples with the same path.

Algorithm 1 Stacking as Post-Processing (Sec. 4.2)

Require: Program γ, Weighted samples from base inference procedure $\{(v_s, \theta_s)\}_{s=1}^S$

1: For each θ_s, record address path and return values $\{g(\tilde{y}_\ell \mid \theta_s)\}_{\ell=1}^L$ (c.f. §4.2)
2: Partition sample indices $1, \ldots, S$ into subsets $\{I_k\}_{k=1}^K$ using address paths (c.f. §4.2)
3: Compute predictive densities $\hat{\rho}_k(\tilde{y}_\ell)$ (Eq. (14))
4: Compute $w^* = \operatorname{argmax} \hat{R}(w)$ (Eq. (15))
5: Compute new sample weights ω_s (Eq. (16))
6: **return** $\{(\omega_s, \theta_s)\}_{s=1}^S$

The full Bayes posterior hence implicitly uses the approximation $\hat{\pi}(\theta) = \sum_k V_k\, \hat{\pi}_k(\theta)$, assigning the weight V_k to each SLP. We instead replace these with the learnable SLP weights w_k:

$$\tilde{\pi}(\theta; w) \approx \sum_k w_k\, \hat{\pi}_k(\theta) = \sum_k \sum_{s \in I_k} \frac{w_k\, v_s}{V_k}\, \delta_{\theta_s}(\theta). \tag{12}$$

We can then use this approximation to get an estimate of the stacked predictive density defined in Eq. (9)

$$\tilde{\rho}(\tilde{y}_\ell; w) \approx \sum\nolimits_k w_k\, \hat{\rho}_k(\tilde{y}_\ell) \tag{13}$$

$$\text{where} \quad \hat{\rho}_k(\tilde{y}_\ell) := \sum\nolimits_{s \in I_k} (v_s / V_k)\, p(\tilde{y}_\ell \mid \theta_s) \tag{14}$$

are the local posterior approximations. In our implementation, the user implicitly defines the $p(\tilde{y}_\ell \mid \cdot)$ through the program return values. We can now approximate $R(w)$ using

$$\hat{R}(w) := \frac{1}{L} \sum\nolimits_{\ell=1}^L \log\left(\sum\nolimits_k w_k\, \hat{\rho}_k(\tilde{y}_\ell)\right). \tag{15}$$

Note that, as the $\hat{\rho}_k(\tilde{y}_\ell)$ do not depend on the SLP weights, we can precompute these estimates before optimizing w in a separate, typically cheap, procedure.

After having obtained the optimized weights, $w^* = \operatorname{argmax}_w \hat{R}(w)$, we want to be able to obtain estimates w.r.t. the reweighted normalized density $\tilde{\pi}(\theta; w^*)$. This can be done easily by reweighting the individual posterior samples θ_s. Letting $k(\theta_s)$ denote the SLP index of sample θ_s we can rewrite Eq. (12) as

$$\tilde{\pi}(\theta; w) \approx \sum\nolimits_{s=1}^S \omega_s\, \delta_{\theta_s}(\theta);\ \omega_s := \frac{w_{k(\theta_s)}\, v_s}{V_{k(\theta_s)}}. \tag{16}$$

Alg. 1 summarizes the high-level steps of our post-processing stacking procedure. Given an input program and corresponding (weighted) posterior samples, we first use the program to extract the address path and return values for each sample θ_s (e.g. using the `Trace` data structure in Pyro). Then, we compute the index subsets $I_1, \ldots, I_K$ by grouping unique address paths together. We can then compute the estimate of the local posterior predictive densities $\hat{\rho}_k(\tilde{y}_\ell)$ and store the estimates in a $K \times L$ matrix. Finally, this matrix can be used to evaluate our stacking objective $\hat{R}(w)$ and

thus optimize the weights. This optimization can be done cheaply relative to the cost of inference as it is a convex optimization of a small number of parameters and does not require further inferences; we use the L-BFGS-B algorithm (Byrd et al., 1995; Zhu et al., 1997) for this. In App. E we provide further details on our full implementation in Pyro.

4.3 Stacking Without Validation Sets

So far, we have assumed the existence of an explicit predictive density p and held-out data $\{\tilde{y}_\ell\}_{\ell=1}^L$, but neither is often actually needed to utilize stacking. Specifically, if we assume that the local unnormalized SLP densities model the observed data $y_i \in \mathcal{Y}$ as conditionally independent given parameters θ, then for each SLP we can write the local unnormalized density as

$$\gamma_k(\theta, y_{1:N}) = \mathbb{I}\left[\theta \in \Theta_k\right] f(\theta) \prod_{i=1}^N g(y_i \mid \theta),$$

where $f_k : \Theta \to \mathbb{R}^{\geq 0}$ represents a prior density on θ. We then use this to derive the following leave-one out (LOO) cross-validation estimator for Eq. (11),

$$R_{\mathrm{LOO}}(w) = \frac{1}{N} \sum_{i=1}^N \log \sum_k w_k \, \rho_k(y_i \mid y_{-i}).$$

where $\rho_k(y_i \mid y_{-i})$ denotes the local posterior predictive density corresponding to $\gamma_k(\theta, y_{1:N} \setminus y_i)$, i.e. that results from removing the i-th observation term from γ_k.

Naively computing each of the predictive distributions $\rho_k(y_i \mid y_{-i})$ would require running inference N times to evaluate the stacking objective. However, as shown in Yao et al. (2018), this can be avoided using Pareto smoothed importance sampling to estimate the LOO densities (PSIS-LOO) (Vehtari et al., 2017). Computing the PSIS-LOO approximations to the densities $\rho_k(y_i \mid y_{-i})$ requires access the individual likelihood terms $g(y_i \mid \theta_s)$ for each posterior sample θ_s (c.f. App. D). Luckily, these can be extracted automatically for many common PPSs, e.g. using the `loo` function of the ArviZ library (Kumar et al., 2019) for Pyro models.

4.4 Regularized Stacking and PAC-Bayes

Stacking directly optimizes an estimate of the expected predictive density on held-out data (Eq. (11)). Such estimates are fundamentally based on a finite amount of data and optimizing them directly can, at least in principle, lead to overfitting. Our proposed remedy for this is to add an additional KL regularization term inspired by PAC-Bayes objectives. Namely we consider

$$R_\beta(w) := \frac{1}{L} \sum_{\ell=1}^L \log\left(\sum_k w_k \rho_k(\tilde{y}_\ell)\right) - (1/\beta L)\, \mathrm{KL}(\mathrm{Categorical}(w_1, \ldots, w_K) \parallel r(k)),$$

where $r(k)$ is a reference weighting we want to regularize towards. Since we want to discourage the SLP weights from collapsing towards a single SLP, we will generally take $r(k)$ to be the uniform distribution. The hyperparameter β controls the amount of regularization: $\beta \to \infty$ recovers the standard stacking objective, and $\beta \to 0$ leads to the weights following $r(k)$.

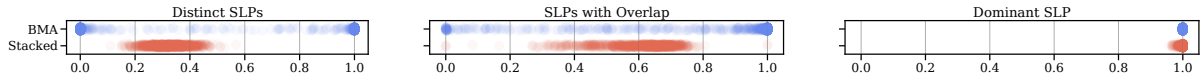
This regularized objective corresponds to a particular instantiation of a PAC-Bayes bound that was proposed by Morningstar et al. (2022). In the PAC-Bayes literature, $-R(w)$ (Eq. (11)) is sometimes referred to as the *true predictive risk*. Hence, optimizing $R_\beta(w)$ can be viewed as optimizing a stochastic bound on the true predictive risk, see App. G for details.

5 RELATED WORK

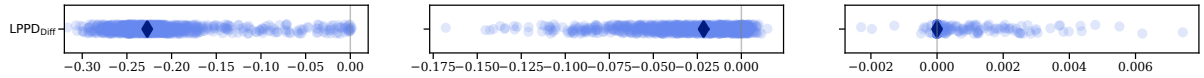
Alternatives to BMA. Bayesian model combination (BMC) (Minka, 2000; Monteith et al., 2011; Kim and Ghahramani, 2012) aims to break the BMA assumption that the data was generated from exactly one of the candidate models. This is done by specifying a new extended model which explicitly combines the predictions of all candidate models. However, fitting the new extended model is significantly more expensive as the inference task no longer breaks down into independent sub-problems. Further, how to best combine predictions from different models is highly problem dependent and a modelling decision in its own right and, hence, not suitable for automation.

Yao et al. (2022) introduced Bayesian hierarchical stacking, which infers different weights for different regions in the covariate space, similar to the frequentist mixture of experts (Gormley and Frühwirth-Schnatter, 2019). This is less suitable for the fully automated PPS setting because it requires knowledge of the covariate space and assumes the existence of covariates in the first place. So-called Pseudo-BMA weights (Geisser and Eddy, 1979) use LOO predictive densities to replace marginal likelihoods but have been shown to work less well than stacking (Yao et al., 2018). BayesBag (Huggins and Miller, 2021) weights models by generating bootstrapped datasets, and averaging the normalization constant over datasets. This is computationally very intensive as it requires running inference separately in each bootstrapped dataset.

PAC-Bayes. Warrell and Gerstein (2022) extend PAC-Bayes bounds to work with hierarchical models inspired by deep probabilistic programs (Tran et al., 2017) and use this extension to derive bounds for multi-task settings such as transfer and meta-learning. However, they do not consider programs with stochastic support, the impact of model misspecification, nor integrate their method with a PPS. PAC-Bayes style arguments have also been used to reason about the hardness of posterior inference in PPS (Freer et al., 2010).



(a) Computed weight for the first SLP; each dot is the weight computed in one of the datasets



(b) $\text{LPPD}_{\text{Diff}}$ on held-out data. Negative values imply stacking is performing better. Diamonds show medians.

Figure 1: Behaviour of the BMA and stacked weights in the models as described in Sec. 6.1.

Programs with stochastic support as BMA. Existing inference algorithms for programs with stochastic support (Wingate et al. (2011); Yang et al. (2014); Wood et al. (2014); Rainforth et al. (2016); Le et al. (2017); Mak et al. (2021, 2022), c.f. App. A) all implicitly generate a weighting of individual SLPs through the proportion of (weighted) samples generated from each SLP. Some inference algorithms are adaptions and extensions of the reversible-jump MCMC methods that were originally developed for the BMA setting (Green, 1995; Roberts et al., 2019; Cusumano-Towner et al., 2020). However, previous work does not discuss the inherent issues with targeting the BMA model weights and the consequences of this for making predictions; existing algorithms which explicitly assign weights to SLPs only target the default BMA model weights (Zhou et al., 2020; Luo et al., 2021; Reichelt et al., 2022a).

6 EXPERIMENTS

We will now compare different weighting schemes on a range of models and datasets. Our quantitative measure for comparison will be the average log posterior predictive density (LPPD) on held-out data $\tilde{y}_{1:T}$, where $\text{LPPD} := \frac{1}{T}\sum_{t=1}^{T} \log \tilde{\rho}(\tilde{y}_t; w)$. In particular, we focus on the difference in LPPD from other methods to stacking, $\text{LPPD}_{\text{Diff}} = \text{LPPD}_{\text{Other}} - \text{LPPD}_{\text{Stacking}}$.

Additionally, we will investigate the behaviour of the SLP weights w_k; a major criticism of the BMA weights is that they are too sensitive to minor changes in the data. To ensure replicable analysis, we desire the SLP weights to be *robust* and *consistent*, i.e. they should be similar across different possible generated datasets.

Except when otherwise indicated, we use a variant of the divide, conquer, and combine (DCC) inference algorithm (Zhou et al., 2020) for the base inference algorithm. Our DCC implementation uses HMC (Neal, 2011; Hoffman et al., 2014; Betancourt, 2018) for the local inference algorithm of each SLP and allocates the computational budget uniformly between SLPs. Additionally, we also consider the reversible-jump MCMC (RJMCMC, c.f. App. A) algorithm implemented in Gen (Cusumano-Towner et al., 2019). Our implementation is available at `https://github.com/treigerm/beyond_bma_in_probprog`.

6.1 When is Stacking Helpful?

First, to develop an understanding of the scenarios in which stacking is beneficial, we consider three simple examples in which we limit ourselves to input programs with two SLPs. Unless otherwise stated, BMA weights are computed analytically and the stacking weights are based on PSIS-LOO. For each problem, we generate 10^3 datasets with 200 data points each and generate another 10^3 data points to evaluate the held-out LPPD.

Distinct SLPs. For the first setting, we assume the data is generated by $y_i \sim \mathcal{N}(0,1)$. We consider a program with two, misspecified SLPs where the unnormalized density for the kth SLP is given by $\gamma_k(\theta_1, \theta_2) = \mathbb{I}[\theta_2 = k]\,\frac{1}{2}\prod_{i=1}^{N}\mathcal{N}(y_i; \theta_1, \sigma_k^2)\,\mathcal{N}(\theta_1; 0, 1)$ where we set $\sigma_1^2 = 0.62177$ and $\sigma_2^2 = 2$ (cf. App. A for the corresponding Pyro program). This example is adapted from Yang and Zhu (2018).

SLPs with overlap. Next, we generate a dataset using the relation $y_i = \sum_{d=1}^{4}\beta_d\, x_{i,d} + \epsilon_i$ where $\epsilon_i \sim \mathcal{N}(0,1)$, $x_{i,d} \sim \mathcal{N}(0,1)$ and $\beta = [1.5, 1.5, 0.3, 0.1]$. For inference, we consider a program with two SLPs with unnormalized densities $\gamma_k(\theta_1, \theta_2, \theta_3) = \mathbb{I}[\theta_3 = k]\,\frac{1}{2}\prod_{i=1}^{N}\mathcal{N}(y_i; f_k(\theta_1, \theta_2, x_i), 1)\prod_{j=1}^{2}\mathcal{N}(\theta_j; 0, 1)$, where for the first SLP $f_1(\theta_1, \theta_2, x_i) = \theta_1 x_{i,1} + \theta_2 x_{i,3}$ and for the second $f_2(\theta_1, \theta_2, x_i) = \theta_1 x_{i,1} + \theta_2 x_{i,4}$. Note, that the covariates $x_{i,d}$ are modelled as fixed by the program. Both SLPs are misspecified because they do not have access to all the covariates. However, the two sub-models share complexity/expressiveness, as they both have access to the first covariate.

Dominant SLP. Lastly, we generate data from $y_i = f(x_i) + \epsilon_i$ with $f(x) = 2\,x_i + \sin(5\,x_i)$ and $\epsilon_i, x_i \sim \mathcal{N}(0,1)$. Our program for inference has two SLPs which are defined as $\gamma_k(\theta_1, \theta_2, \theta_3) = \mathbb{I}[\theta_3 = k]\,\frac{1}{2}\prod_{i=1}^{N}\mathcal{N}(y_i; f_k(\theta_1, x_i), \theta_2^2)\,\mathcal{N}(\theta_1; 0, 1)\,\Gamma(\theta_2; 1, 1)$ where $f_1(\theta, x) = \theta\, x$, $f_2(\theta, x) = \sin(\theta\, x)$, and $\Gamma(\theta; \alpha, \beta)$ is a Gamma distribution parameterized by shape α, and rate β. The first SLP will provide a significantly better fit to the data as it is able to recover the dominant linear trend. We use importance sampling to estimate the BMA weights (see App. F).

Results. The results are presented in Fig. 1. For the first two models the stacked weights lead to better predictive performance and more robust weights. While

Table 1: $\text{LPPD}_{\text{Diff}}$ (↑ better) for models in Sec. 6.2, 6.3, and 6.4, results computed over 10 replications. Bold indicates no significant difference to Stacked under a Wilcoxon signed-rank test.

Model	Stacked	Stacked (Val)	BMA	BMA (Analytic)	RJMCMC	Equal
Subset	**0.0**	-0.01 ± 0.01	-0.11 ± 0.05	-0.11 ± 0.05	-0.11 ± 0.04	-0.02 ± 0.01
Fun. Ind. (misspecified)	**0.0**	$\mathbf{-1.73e{-3} \pm 2.98e{-3}}$	$\mathbf{-9.10e{-4} \pm 2.42e{-3}}$	N/A	-0.08 ± 0.03	-0.31 ± 0.01
Fun. Ind. (well-specified)	**0.0**	$\mathbf{-5.61e{-3} \pm 7.14e{-3}}$	$-3.76e{-1} \pm 2.51e{-1}$	N/A	-2.44 ± 0.32	-2.31 ± 0.09
California	**0.0**	$\mathbf{-1.55e{-3} \pm 2.89e{-3}}$	$-2.10e{-2} \pm 6.19e{-3}$	$-2.10e{-2} \pm 6.01e{-3}$	$-2.82e{-1} \pm 1.19e{-1}$	$-1.96e{-1} \pm 3.11e{-3}$
Diabetes	**0.0**	$\mathbf{-8.22e{-3} \pm 1.44e{-2}}$	$\mathbf{-1.01e{-2} \pm 1.64e{-2}}$	N/A	$-3.83e{-2} \pm 2.19e{-2}$	$-3.66e{-2} \pm 9.97e{-3}$
Stroke	**0.0**	$\mathbf{-7.79e{-4} \pm 1.57e{-3}}$	$-6.22e{-3} \pm 3.81e{-3}$	N/A	$-2.25e{-1} \pm 9.94e{-2}$	$-1.31e{-1} \pm 5.68e{-3}$

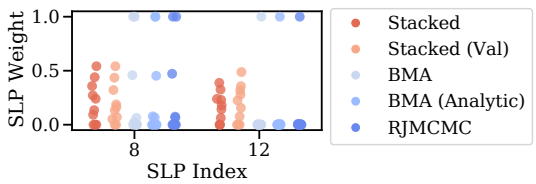


Figure 2: SLP weights for problem in Sec. 6.2. Each dot represents the weight of the corresponding SLP in the model. Results are computed over 10 generated datasets.

there is some variation in the stacked weights, their overall distribution is *unimodal* whereas the distribution of the BMA weights are *bimodal* instead. The tendency of the BMA weights to collapse on either 0 or 1 is exactly the *overconfident* behaviour we described in Sec. 3. In the setting with one dominant SLP, both the BMA and stacking weights lead to similar predictive performance (with BMA doing slightly better) and consistently collapse onto the dominant SLP; here this is desirable behaviour as the first SLP is clearly superior.

6.2 Subset Regression

To further outline the issues of the default BMA weights, we consider a regression problem with data generated from a linear model of the form $y_i = \epsilon_i + \sum_{d=1}^{15} \beta_d \, x_{i,d}$ where $\epsilon_i \sim \mathcal{N}(0, 1)$ and all the covariates $x_{n,d}$ are drawn independently from $\mathcal{N}(5, 1)$. Note, the covariates are sampled to generate the synthetic data set, but they are modelled as fixed by the program. The ground-truth values for the regression coefficients β_d are set following a scheme used in Breiman (1996) and Yao et al. (2018) which ensures all covariates are relevant for the prediction of y_n (c.f. App. F).

We compare multiple methods: 1. **Stacked**, using PSIS-LOO to compute weights, w (Sec. 4.3); 2. **Stacked (Val)**, which uses an explicit validation set instead (Sec. 4.2); 2. **BMA**, using the PI-MAIS algorithm (Martino et al., 2017) to compute local normalization constants (this is the default weighting in DCC); 3. **BMA (Analytic)**, using analytic solutions for the local normalization constants; 4. **RJMCMC**, implemented in Gen (Cusumano-Towner et al., 2019);[2] 5. **Equal**, weights each SLP equally. We use warm colors to present the results which use one of our proposed alternative objectives to set SLP weights and cooler colours for methods which target the BMA weights. Note **BMA**, **BMA (Analytic)**, and **RJMCMC** all target the posterior distribution in Eq. (3); in practice, differences between these methods will arise due to differences in the quality of the posterior approximation.

Our input program has 15 SLPs and each SLP only gets access to one of the covariates so our overall model is misspecified. However, since every covariate influences the targets y_i, each SLP is relevant for making good predictions. We generate 50 different datasets from the true data generating distribution, with 200 data points used to run our inferences and 10^3 data points to evaluate the held-out log posterior predictive density. For the **Stacked (Val)** we use half of the 200 data points for inference and use the rest to estimate the stacking objective (c.f. Eq. (15)).

Tab. 1 shows that stacking outperforms all other methods in terms of predictive performance. The BMA weights here actually provide even worse predictions than weighting each SLP equally. Fig. 2 shows the behaviour of the weights for the SLPs with $k = 8$ and $k = 12$ over different randomly generated datasets (see Fig. 9 in the Appendix for others). Both **BMA** and **BMA (Analytic)** exhibit clear signs of overconfidence as described in Sec. 2.1: the weights often collapse onto a single SLP, but the exact SLP changes between datasets, leading to a bimodal sampling distribution for the SLP weights. As expected, **RJMCMC** produces qualitatively and quantitivaly similar results to the other Bayesian weighting mechanisms. This is in contrast with the **Stacked** weights which are more evenly spread. The **Stacked (Val)** weights behave qualitatively similarly to using PSIS-LOO, but with slightly worse predictive performance. This is likely due to the corresponding reduction in training set size.

6.3 Function Induction

Our next example investigates how well stacking scales to a larger number of SLPs. We generate observations from the relation $y_i = -x_i + 2 \sin(2 \, x_i^2) + \epsilon_i$ with $\epsilon_i \sim \mathcal{N}(0, 0.1^2)$ and the inputs x_i uniformly sampled between -5 and 5. We generate 400 data points used for

[2] We also have conducted experiments that run stacking on top RJMCMC and found that this also led to improvements in predictive performance (c.f. App. F). Here, we only present the stacking results with samples generated from DCC as this gave the best base inference procedure.

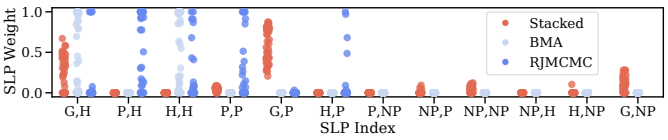


Figure 3: SLP weights for Sec. 6.5. X-tick labels indicate the different modelling choices for α and β; the pattern is "α model choice, β model choice" with P = pooling, NP = no pooling, H = hierarchical, and G = group-level predictor.

Table 2: $\text{LPPD}_{\text{Diff}}$ for Radon model.

Method	$\text{LPPD}_{\text{Diff}}$ (↑)
Stacked	**0.0**
BMA	$-8.24\text{e}{-3} \pm 1.20\text{e}{-2}$
RJMCMC	$-5.99\text{e}{-2} \pm 1.86\text{e}{-2}$
Equal	$-1.88\text{e}{-2} \pm 1.18\text{e}{-2}$

inference and 10^3 data points for evaluation. Following Zhou et al. (2020), we use a probabilistic context-free grammar (PCFG) to posit a model over functions. We consider two PCFGs, the *misspecified* PCFG has production rules $e \to \{x \mid \sin(a * e) \mid a * e + b * e\}$, where x is a terminal symbol denoting an input, and a, b are coefficients to be inferred. The *well-specified* PCFG additionally includes the terminal symbol x^2. Therefore the well-specified PCFG can express the data generating function whereas the misspecified one cannot. The program recursively samples from the PCFG using samples from categorical distributions to select production rules and defines latent variables for all the coefficients in a given expression (c.f. App. F). Note analytic BMA weights cannot be calculated here.

Tab. 1 shows that **Stacked** provides better predictions compared to all other methods, even when the model is well-specified! Notably, inference in this model is particularly challenging due to the fact that distinct SLPs can have similar or even identical posterior predictive distributions due to symmetries in the PCFG, e.g. $x + \sin(x)$ and $\sin(x) + x$ correspond to two separate SLPs. The fact that stacking outperforms the methods targeting the BMA weights in the well-specified case is an indicator that the inference algorithms are struggling in this model. Indeed, we found **RJMCMC** tends to get stuck in a single SLP which is a well-known issue with MCMC methods for programs with stochastic support (all weights are shown in App. F). Even though **Stacked** weights give better predictions, we found that the weights themselves exhibit relatively high-variance. This is due to the finite sample size of the dataset and the usage of approximate inference algorithms which introduce variance in estimating the stacking objective. However, the fact that stacking is able to produce superior predictions shows that it can be a useful mechanism for improving predictive performance even when inference algorithms struggle to produce accurate posteriors approximations.

6.4 Variable Selection

Next, we apply stacking to real-world classification and regression tasks. Here, we have a matrix of covariates $X \in \mathbb{R}^{N \times D}$ and targets $y_{1:N}$, and we want to do variable selection, i.e. select a subset of the features $\mathcal{D} \subseteq \{1, \ldots, D\}$ to make predictions. This problem of variable selection can be encoded as a probabilistic program with stochastic support in which each SLP corresponds to one of the potential subsets of the features $\mathcal{D}$. We consider three different datasets: *California* (regression) (Pace and Barry, 1997), *Diabetes* (classification) (Smith et al., 1988), and *Stroke* (classification) (Kaggle, 2020). For the regression, our model is a linear regression with conjugate priors, permitting an analytic solution to the BMA weights. For the classification tasks, we use a logistic regression model which does not permit an analytic solution. As the true data generating process in this setting is unknown, we run each method on different train-test splits to estimate the variation in the weights and predictions.

Tab. 1 shows the LPPD values of the weighting schemes on different datasets. The **Stacked** and **Stacked (Val)** weighting schemes generally give better predictive performance compared to the alternatives. Overall, these results show that stacking can be beneficial for predictive performance even on real-world data.

6.5 Radon Contamination

Our final example considers the analysis of data about Radon contamination for houses in different US counties (Gelman and Hill, 2006). We here only give a high-level description of the dataset and model, full details in App. F. For each house recorded in the dataset, we have radon measurements, y_i, as well as, a covariate, $x_i \in \{0, 1\}$, which indicates whether the measurement was made in the basement ($x_i = 0$) or first floor of the house ($x_i = 1$). Our program for this dataset has at its core the regression relation $y_i = \alpha + \beta\, x_i + \epsilon_i$ and the different SLPs in the program make different assumptions for how to model the coefficients α and β. For both, we can either: 1) Fit the same coefficient across all counties; 2) Fit a separate coefficient α_c for each county c; or 3) Have separate coefficients for each county but assume they come from the same underlying population distribution. For the intercept term we also consider a fourth option: using county-wide level uranium measurements as a group-level predictor. The program considers all combinations of modelling choices for α and β, i.e. it has $4 \cdot 3 = 12$ SLPs.

Each SLP in this program can have potentially hundreds of latent variables and exhibit complex posterior geometries, making this a good testing ground to compare the different weighting schemes. In Tab. 2 we

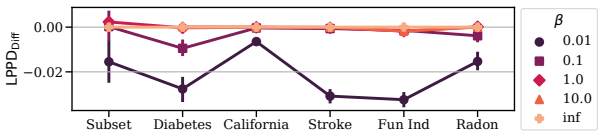


Figure 4: Impact of regularization parameter β on predictive performance in the different models (higher is better). Plotted are mean and standard deviation.

Table 3: Timings for running inference and stacking, averaged over 5 runs. Inference is conducted using DCC.

	Subset	Var. Select.	Fun. Ind.	Radon
Inference	29 s	297 s	285 m	700 s
Stacking	0.09 s	43 s	11 s	0.2 s

find that the **Stacked** weights give better performance compared to the **BMA** weights (we do not consider using a validation set here because some counties contain only a handful of observations). Fig. 3 shows that the **BMA** weights tend to concentrate on SLPs with modelling choices "H,H" or "G,H" and have a bimodal sampling distribution. The **Stacked** weights are more robust, giving more consistent results between different train-test splits and more conservative weights. Notably, **RJMCMC** here collapses onto different SLPs than the BMA weights. This is due to the fact that the RJMCMC struggles with SLPs which have a large number of latent variables. In the limit of infinite samples, the behaviour of RJMCMC and BMA will be identical but Fig. 3 illustrates nicely that approximate inference algorithms might collapse onto different SLPs based on the quality of their posterior approximation.

6.6 Impact of Regularization: PAC-Bayes

As we have shown in Sec. 4.4, the PAC-Bayes bound $R_\beta(w)$ offers an alternative objective to fit the weights and can be interpreted as the stacking loss with an added regularization term where the hyperparameter β controls the amount of regularization. Smaller values of β push the stacking weights closer to the uniform distribution over SLPs. In Fig. 4 we plot the effect of varying β on the predictive performance on the different models. In our experiments, values of β below 1 tend to lead to worse predictive performance and the stacking objective with no regularization ($\beta = \infty$ in Fig. 4) is not outperformed by any form of regularization. However, depending on the application setting some level of regularization might still be desirable.

7 DISCUSSION

We have demonstrated that in programs with stochastic support, the conventional posterior path probabilities can be unstable (e.g. due to model misspecification or inference approximations) and that this in turn can lead to sub-optimal predictions.

In practice, one of the key sources of instability is model misspecification. When dealing with misspecification, the general advice is to revise or expand the model (Gelman et al., 2020). However, when using real-world data it is often not obvious how to further expand a model and mitigate against misspecification. The radon experiment (Sec. 6.5) is a good example here as it is already the result of multiple model iterations and expansions, with no clear strategy for how to extend it further. Additionally, revising and fitting a new expanded model is often prohibitively expensive and therefore not a viable alternative. As our timings in Tab. 3 demonstrate, stacking is a very cheap procedure compared to the cost of inference. With the automated post-processing techniques presented in this paper, stacking can therefore be conveniently applied at the end of an analysis after the user has gone through multiple iterations of model building. Thus, rather than viewing stacking as a replacement for model expansion, we view it as a useful tool to safeguard against the instabilities of the default BMA weighting scheme.

While we have demonstrated that the instability in the BMA weights can appear in realistic models and datasets, for any given problem there is no guarantee that the BMA weights will indeed be unstable. For example, as we saw in the initial experiments in Sec. 6.1, stacking and BMA will produce similar weights if there is one SLP which clearly dominates the others. However, finding clear criteria that determine when BMA will lead to unstable weights is still an area of open research (Yang and Zhu, 2018; Oelrich et al., 2020; Huggins and Miller, 2021), so for practitioners it is difficult to know a priori whether a given model will produce unstable SLP weights or not.

Overall, this means there are few reasons not to use stacking: it is cheap, easy-to-use, provides generally more robust weights, and leads to improved predictions.

Acknowledgements

We would like to thank Mrinank Sharma and Andrew Campbell for feedback on earlier drafts of this manuscript. We would also like to thank the anonymous reviewers for providing their reviews, especially for encouraging us to explore the connections to PAC-Bayes. Tim Reichelt is supported by the UK EPSRC CDT in Autonomous Intelligent Machines and Systems with the grant EP/S024050/1. Luke Ong acknowledges support from the National Research Foundation, Singapore, under its RSS Scheme (NRF-RSS2022-009). Tom Rainforth is supported by the UK EPSRC grant EP/Y037200/1. The authors would like to acknowledge the use of the University of Oxford Advanced Research Computing (ARC) facility in carrying out this work (http://dx.doi.org/10.5281/zenodo.22558).

References

David Tolpin, Jan-Willem van de Meent, Hongseok Yang, and Frank Wood. Design and Implementation of Probabilistic Programming Language Anglican. In *Proceedings of the 28th Symposium on the Implementation and Application of Functional Programming Languages - IFL 2016*, pages 1–12, Leuven, Belgium, 2016. ACM Press. ISBN 978-1-4503-4767-9. doi: 10.1145/3064899.3064910. URL http://dl.acm.org/citation.cfm?doid=3064899.3064910.

Noah D. Goodman, Vikash K. Mansinghka, Daniel Roy, Keith Bonawitz, and Joshua B. Tenenbaum. Church: A language for generative models. UAI'08, July 2008.

Eli Bingham, Jonathan P. Chen, Martin Jankowiak, Fritz Obermeyer, Neeraj Pradhan, Theofanis Karaletsos, Rohit Singh, Paul Szerlip, Paul Horsfall, and Noah D. Goodman. Pyro: Deep Universal Probabilistic Programming. *Journal of Machine Learning Research*, 2019.

Hong Ge, Kai Xu, and Zoubin Ghahramani. Turing: A Language for Flexible Probabilistic Inference. In *International Conference on Artificial Intelligence and Statistics*. PMLR, March 2018.

Vikash Mansinghka, Daniel Selsam, and Yura Perov. Venture: A higher-order probabilistic programming platform with programmable inference. *arXiv:1404.0099 [cs, stat]*, March 2014.

Arun Chaganty, Aditya Nori, and Sriram Rajamani. Efficiently Sampling Probabilistic Programs via Program Analysis. In *Proceedings of the Sixteenth International Conference on Artificial Intelligence and Statistics*, pages 153–160. PMLR, April 2013. URL https://proceedings.mlr.press/v31/chaganty13a.html.

Sriram Sankaranarayanan, Aleksandar Chakarov, and Sumit Gulwani. Static analysis for probabilistic programs: Inferring whole program properties from finitely many paths. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 447–458, New York, NY, USA, June 2013. Association for Computing Machinery. ISBN 978-1-4503-2014-6. doi: 10.1145/2491956.2462179. URL https://doi.org/10.1145/2491956.2462179.

Yicheng Luo, Antonio Filieri, and Yuan Zhou. Symbolic parallel adaptive importance sampling for probabilistic program analysis. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1166–1177, 2021.

Yuan Zhou, Hongseok Yang, Yee Whye Teh, and Tom Rainforth. Divide, Conquer, and Combine: A New Inference Strategy for Probabilistic Programs with Stochastic Support. In *Proceedings of the 37th International Conference on Machine Learning*, pages 11534–11545. PMLR, November 2020. URL https://proceedings.mlr.press/v119/zhou20e.html.

Tim Reichelt, Luke Ong, and Tom Rainforth. Rethinking variational inference for probabilistic programs with stochastic support. In *Advances in Neural Information Processing Systems*, 2022a.

Jennifer A Hoeting, David Madigan, Adrian E Raftery, and Chris T Volinsky. Bayesian model averaging: a tutorial (with comments by m. clyde, david draper and ei george, and a rejoinder by the authors. *Statistical science*, 14(4):382–417, 1999.

Thomas P Minka. Bayesian model averaging is not model combination. 2000.

Yuling Yao, Aki Vehtari, Daniel Simpson, and Andrew Gelman. Using Stacking to Average Bayesian Predictive Distributions (with Discussion). *Bayesian Analysis*, 13(3):917–1007, September 2018. ISSN 1936-0975, 1931-6690. doi: 10.1214/17-BA1091.

Andrew Gelman and Yuling Yao. Holes in Bayesian statistics. *Journal of Physics G: Nuclear and Particle Physics*, 48(1), December 2020.

Oscar Oelrich, Shutong Ding, Måns Magnusson, Aki Vehtari, and Mattias Villani. When are bayesian model probabilities overconfident? *arXiv preprint arXiv:2003.04026*, 2020.

Jonathan H. Huggins and Jeffrey W. Miller. Reproducible Model Selection Using Bagged Posteriors, December 2021. URL http://arxiv.org/abs/2007.14845.

Ziheng Yang and Tianqi Zhu. Bayesian selection of misspecified models is overconfident and may cause spurious posterior probabilities for phylogenetic trees. *Proceedings of the National Academy of Sciences*, 115(8):1854–1859, February 2018. doi: 10.1073/pnas.1712673115. URL https://www.pnas.org/doi/full/10.1073/pnas.1712673115.

George EP Box. Science and statistics. *Journal of the American Statistical Association*, 71(356):791–799, 1976.

Jane T Key, Luis R Pericchi, and Adrian FM Smith. Bayesian model choice: what and why. *Bayesian statistics*, 6:343–370, 1999.

Aki Vehtari and Janne Ojanen. A survey of Bayesian predictive methods for model assessment, selection and comparison. *Statistics Surveys*, 6(none):142–228, January 2012. ISSN 1935-7516. doi: 10.1214/12-SS102.

Frank Smets and Rafael Wouters. Shocks and frictions in us business cycles: A bayesian dsge approach. *American economic review*, 97(3):586–606, 2007.

Alexander P Leff, Thomas M Schofield, Klass E Stephan, Jennifer T Crinion, Karl J Friston, and Cathy J Price. The cortical dynamics of intelligible speech. *Journal of Neuroscience*, 28(49):13209–13215, 2008.

David H. Wolpert. Stacked generalization. *Neural Networks*, 5(2):241–259, 1992. ISSN 0893-6080. doi: https://doi.org/10.1016/S0893-6080(05)80023-1. URL https://www.sciencedirect.com/science/article/pii/S0893608005800231.

Leo Breiman. Stacked regressions. *Machine learning*, 24(1):49–64, 1996.

Michael LeBlanc and Robert Tibshirani. Combining estimates in regression and classification. *Journal of the American Statistical Association*, 1996.

Andres Masegosa. Learning under model misspecification: Applications to variational and ensemble methods. *Advances in Neural Information Processing Systems*, 33:5479–5491, 2020.

Mohammad Saeed Masiha, Amin Gohari, Mohammad Hossein Yassaee, and Mohammad Reza Aref. Learning under distribution mismatch and model misspecification. In *2021 IEEE International Symposium on Information Theory (ISIT)*, pages 2912–2917. IEEE, 2021.

Warren R. Morningstar, Alex Alemi, and Joshua V. Dillon. Pacm-bayes: Narrowing the empirical risk gap in the misspecified bayesian regime. In Gustau Camps-Valls, Francisco J. R. Ruiz, and Isabel Valera, editors, *Proceedings of The 25th International Conference on Artificial Intelligence and Statistics*, volume 151 of *Proceedings of Machine Learning Research*, pages 8270–8298. PMLR, 28–30 Mar 2022. URL https://proceedings.mlr.press/v151/morningstar22a.html.

Pierre Alquier. User-friendly introduction to pac-bayes bounds, 2023.

José M Bernardo and Adrian FM Smith. *Bayesian theory*, volume 405. John Wiley & Sons, 2009.

Merlise Clyde and Edwin S Iversen. Bayesian model averaging in the M-open framework. In *Bayesian Theory and Applications*. Oxford University Press, 01 2013.

Johannes Borgström, Ugo Dal Lago, Andrew D. Gordon, and Marcin Szymczak. A lambda-calculus foundation for universal probabilistic programming. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*, ICFP 2016. Association for Computing Machinery, September 2016.

Sam Staton, Hongseok Yang, Frank Wood, Chris Heunen, and Ohad Kammar. Semantics for probabilistic programming: Higher-order functions, continuous distributions, and soft constraints. In *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science*, LICS '16, New York, NY, USA, July 2016. Association for Computing Machinery.

Andrew D. Gordon, Thomas A. Henzinger, Aditya V. Nori, and Sriram K. Rajamani. Probabilistic programming. In *Future of Software Engineering Proceedings*, FOSE 2014, pages 167–181, New York, NY, USA, May 2014. Association for Computing Machinery.

Thomas William Gamlen Rainforth. *Automating Inference, Learning, and Design Using Probabilistic Programming*. http://purl.org/dc/dcmitype/Text, University of Oxford, 2017. URL https://ora.ox.ac.uk/objects/uuid:e276f3b4-ff1d-44bf-9d67-013f68ce81f0.

Jan-Willem van de Meent, Brooks Paige, Hongseok Yang, and Frank Wood. An Introduction to Probabilistic Programming. *arXiv:1809.10756 [cs, stat]*, September 2018. URL http://arxiv.org/abs/1809.10756.

Tilmann Gneiting and Adrian E Raftery. Strictly proper scoring rules, prediction, and estimation. *Journal of the American statistical Association*, 102 (477):359–378, 2007.

Richard H Byrd, Peihuang Lu, Jorge Nocedal, and Ciyou Zhu. A limited memory algorithm for bound constrained optimization. *SIAM Journal on scientific computing*, 16(5):1190–1208, 1995.

Ciyou Zhu, Richard H Byrd, Peihuang Lu, and Jorge Nocedal. Algorithm 778: L-bfgs-b: Fortran subroutines for large-scale bound-constrained optimization. *ACM Transactions on mathematical software (TOMS)*, 23(4):550–560, 1997.

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

Ravin Kumar, Colin Carroll, Ari Hartikainen, and Osvaldo Martin. Arviz a unified library for exploratory analysis of bayesian models in python. *Journal of Open Source Software*, 2019. doi: 10.21105/joss.01143.

Kristine Monteith, James L. Carroll, Kevin Seppi, and Tony Martinez. Turning bayesian model averaging into bayesian model combination. In *The 2011 International Joint Conference on Neural Networks*, 2011.

Hyun-Chul Kim and Zoubin Ghahramani. Bayesian classifier combination. In Neil D. Lawrence and Mark Girolami, editors, *Proceedings of the Fifteenth International Conference on Artificial Intelligence*

and Statistics, volume 22 of *Proceedings of Machine Learning Research*, pages 619–627, La Palma, Canary Islands, 21–23 Apr 2012. PMLR. URL https://proceedings.mlr.press/v22/kim12.html.

Yuling Yao, Gregor Pirš, Aki Vehtari, and Andrew Gelman. Bayesian hierarchical stacking: Some models are (somewhere) useful. *Bayesian Analysis*, 17(4): 1043–1071, 2022.

Isobel Claire Gormley and Sylvia Frühwirth-Schnatter. Mixture of experts models. In *Handbook of mixture analysis*, pages 271–307. Chapman and Hall/CRC, 2019.

Seymour Geisser and William F Eddy. A predictive approach to model selection. *Journal of the American Statistical Association*, 74(365):153–160, 1979.

Jonathan Warrell and Mark Gerstein. Higher-order generalization bounds: Learning deep probabilistic programs via pac-bayes objectives. *arXiv preprint arXiv:2203.15972*, 2022.

Dustin Tran, Matthew D Hoffman, Rif A Saurous, Eugene Brevdo, Kevin Murphy, and David M Blei. Deep probabilistic programming. *arXiv preprint arXiv:1701.03757*, 2017.

Cameron E Freer, Vikash K Mansinghka, and Daniel M Roy. When are probabilistic programs probably computationally tractable. In *NIPS Workshop on Monte Carlo Methods for Modern Applications*, 2010.

David Wingate, Andreas Stuhlmueller, and Noah Goodman. Lightweight Implementations of Probabilistic Programming Languages Via Transformational Compilation. In *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, pages 770–778. JMLR Workshop and Conference Proceedings, June 2011. URL https://proceedings.mlr.press/v15/wingate11a.html.

Lingfeng Yang, Patrick Hanrahan, and Noah Goodman. Generating efficient mcmc kernels from probabilistic programs. In *Artificial Intelligence and Statistics*, pages 1068–1076. PMLR, 2014.

Frank Wood, Jan Willem Meent, and Vikash Mansinghka. A New Approach to Probabilistic Programming Inference. In *Artificial Intelligence and Statistics*, pages 1024–1032. PMLR, April 2014. URL http://proceedings.mlr.press/v33/wood14.html.

Tom Rainforth, Christian Naesseth, Fredrik Lindsten, Brooks Paige, Jan-Willem Vandemeent, Arnaud Doucet, and Frank Wood. Interacting particle markov chain monte carlo. In Maria Florina Balcan and Kilian Q. Weinberger, editors, *Proceedings of The 33rd International Conference on Machine Learning*, volume 48 of *Proceedings of Machine Learning Research*, pages 2616–2625, New York, New York, USA, 20–22 Jun 2016. PMLR. URL https://proceedings.mlr.press/v48/rainforth16.html.

Tuan Anh Le, Atilim Gunes Baydin, and Frank Wood. Inference Compilation and Universal Probabilistic Programming. In *Artificial Intelligence and Statistics*, pages 1338–1348. PMLR, April 2017.

Carol Mak, Fabian Zaiser, and Luke Ong. Nonparametric Hamiltonian Monte Carlo. In *Proceedings of the 38th International Conference on Machine Learning*, pages 7336–7347. PMLR, July 2021. URL https://proceedings.mlr.press/v139/mak21a.html.

Carol Mak, Fabian Zaiser, and Luke Ong. Nonparametric involutive markov chain monte carlo. In Kamalika Chaudhuri, Stefanie Jegelka, Le Song, Csaba Szepesvári, Gang Niu, and Sivan Sabato, editors, *International Conference on Machine Learning, ICML 2022, 17-23 July 2022, Baltimore, Maryland, USA*, volume 162 of *Proceedings of Machine Learning Research*, pages 14802–14859. PMLR, 2022. URL https://proceedings.mlr.press/v162/mak22a.html.

Peter J. Green. Reversible jump Markov chain Monte Carlo computation and Bayesian model determination. *Biometrika*, 82(4):711–732, December 1995. ISSN 0006-3444. doi: 10.1093/biomet/82.4.711. URL https://doi.org/10.1093/biomet/82.4.711.

David A. Roberts, Marcus Gallagher, and Thomas Taimre. Reversible Jump Probabilistic Programming. In *Proceedings of the Twenty-Second International Conference on Artificial Intelligence and Statistics*, pages 634–643. PMLR, April 2019. URL https://proceedings.mlr.press/v89/roberts19a.html.

Marco Cusumano-Towner, Alexander K. Lew, and Vikash K. Mansinghka. Automating Involutive MCMC using Probabilistic and Differentiable Programming. *arXiv:2007.09871 [stat]*, July 2020. URL http://arxiv.org/abs/2007.09871.

Radford M. Neal. MCMC using Hamiltonian dynamics. In *Handbook of Markov Chain Monte Carlo*, pages 113–162. Chapman & Hall / CRC Press, 2011.

Matthew D Hoffman, Andrew Gelman, et al. The no-u-turn sampler: adaptively setting path lengths in hamiltonian monte carlo. *J. Mach. Learn. Res.*, 15 (1):1593–1623, 2014.

Michael Betancourt. A Conceptual Introduction to Hamiltonian Monte Carlo. *arXiv:1701.02434 [stat]*, July 2018.

Marco F. Cusumano-Towner, Feras A. Saad, Alexander K. Lew, and Vikash K. Mansinghka. Gen: A general-purpose probabilistic programming system with programmable inference. PLDI 2019, June 2019.

L. Martino, V. Elvira, D. Luengo, and J. Corander. Layered adaptive importance sampling. *Statistics and Computing*, 27(3), May 2017.

R Kelley Pace and Ronald Barry. Sparse spatial autoregressions. *Statistics & Probability Letters*, 33(3): 291–297, 1997.

Jack W Smith, James E Everhart, WC Dickson, William C Knowler, and Robert Scott Johannes. Using the adap learning algorithm to forecast the onset of diabetes mellitus. In *Proceedings of the annual symposium on computer application in medical care*, page 261. American Medical Informatics Association, 1988.

Kaggle. Stroke prediction dataset. https://www.kaggle.com/datasets/fedesoriano/stroke-prediction-dataset, 2020. Accessed: 2023-10-05.

Andrew Gelman and Jennifer Hill. *Data analysis using regression and multilevel/hierarchical models*. Cambridge university press, 2006.

Andrew Gelman, Aki Vehtari, Daniel Simpson, Charles C. Margossian, Bob Carpenter, Yuling Yao, Lauren Kennedy, Jonah Gabry, Paul-Christian Bürkner, and Martin Modrák. Bayesian Workflow. *arXiv:2011.01808 [stat]*, November 2020. URL http://arxiv.org/abs/2011.01808.

Alexander K. Lew, Marco F. Cusumano-Towner, Benjamin Sherman, Michael Carbin, and Vikash K. Mansinghka. Trace types and denotational semantics for sound programmable inference in probabilistic languages. *Proceedings of the ACM on Programming Languages*, 4(POPL):19:1–19:32, December 2019. doi: 10.1145/3371087. URL https://doi.org/10.1145/3371087.

Lawrence M. Murray and Thomas B. Schön. Automated learning with a probabilistic programming language: Birch. *arXiv:1810.01539 [cs, stat]*, April 2020. URL http://arxiv.org/abs/1810.01539.

Atilim Güneş Baydin, Lei Shao, Wahid Bhimji, Lukas Heinrich, Lawrence Meadows, Jialin Liu, Andreas Munk, Saeid Naderiparizi, Bradley Gram-Hansen, Gilles Louppe, Mingfei Ma, Xiaohui Zhao, Philip Torr, Victor Lee, Kyle Cranmer, Prabhat, and Frank Wood. Etalumis: Bringing probabilistic programming to scientific simulators at scale. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. Association for Computing Machinery, November 2019.

William Harvey, Andreas Munk, Atılım Güneş Baydin, Alexander Bergholm, and Frank Wood. Attention for inference compilation. *arXiv preprint arXiv:1910.11961*, 2019.

David Tolpin, Jan-Willem van de Meent, Brooks Paige, and Frank Wood. Output-sensitive adaptive metropolis-hastings for probabilistic programs. In *Machine Learning and Knowledge Discovery in Databases: European Conference, ECML PKDD 2015, Porto, Portugal, September 7-11, 2015, Proceedings, Part II 15*, pages 311–326. Springer, 2015.

David Wingate and Theophane Weber. Automated Variational Inference in Probabilistic Programming. *arXiv:1301.1299 [cs, stat]*, January 2013. URL http://arxiv.org/abs/1301.1299.

Rajesh Ranganath, Sean Gerrish, and David Blei. Black Box Variational Inference. In *Proceedings of the Seventeenth International Conference on Artificial Intelligence and Statistics*, pages 814–822. PMLR, April 2014. URL https://proceedings.mlr.press/v33/ranganath14.html.

Timothy Brooks Paige. *Automatic Inference for Higher-Order Probabilistic Programs*. http://purl.org/dc/dcmitype/Text, University of Oxford, 2016. URL https://ora.ox.ac.uk/objects/uuid:d912c4de-4b08-4729-aa19-766413735e2a.

José M Bernardo. Expected information as expected utility. *the Annals of Statistics*, pages 686–690, 1979.

Aki Vehtari, Daniel Simpson, Andrew Gelman, Yuling Yao, and Jonah Gabry. Pareto smoothed importance sampling. *arXiv preprint arXiv:1507.02646*, 2015.

Robert Zinkov and Chung-chieh Shan. Composing inference algorithms as program transformations. *Proceedings of Uncertainty in Artificial Intelligence (UAI)*, page 10, 2017.

Tim Reichelt, Adam Goliński, Luke Ong, and Tom Rainforth. Expectation programming: Adapting probabilistic programming systems to estimate expectations efficiently. In *Uncertainty in Artificial Intelligence*, pages 1676–1685. PMLR, 2022b.

Alexander K Lew, Mathieu Huot, Sam Staton, and Vikash K Mansinghka. Adev: Sound automatic differentiation of expected values of probabilistic programs. *Proceedings of the ACM on Programming Languages*, 7(POPL):121–153, 2023.

Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, C J Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R. Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1.0 Contributors.

SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 17:261–272, 2020. doi: 10.1038/s41592-019-0686-2.

Dong C Liu and Jorge Nocedal. On the limited memory bfgs method for large scale optimization. *Mathematical programming*, 45(1-3):503–528, 1989.

Christopher M Bishop and Nasser M Nasrabadi. *Pattern recognition and machine learning*, volume 4. Springer, 2006.

Yuri Burda, Roger Grosse, and Ruslan Salakhutdinov. Importance weighted autoencoders. *arXiv preprint arXiv:1509.00519*, 2015.

Warren Morningstar, Sharad Vikram, Cusuh Ham, Andrew Gallagher, and Joshua Dillon. Automatic differentiation variational inference with mixtures. In *International Conference on Artificial Intelligence and Statistics*, pages 3250–3258. PMLR, 2021.

Checklist

1. For all models and algorithms presented, check if you include:
   (a) A clear description of the mathematical setting, assumptions, algorithm, and/or model. [Yes]
   (b) An analysis of the properties and complexity (time, space, sample size) of any algorithm. [Yes]
   (c) (Optional) Anonymized source code, with specification of all dependencies, including external libraries. [Yes]

2. For any theoretical claim, check if you include:
   (a) Statements of the full set of assumptions of all theoretical results. [Not Applicable]
   (b) Complete proofs of all theoretical results. [Not Applicable]
   (c) Clear explanations of any assumptions. [Not Applicable]

3. For all figures and tables that present empirical results, check if you include:
   (a) The code, data, and instructions needed to reproduce the main experimental results (either in the supplemental material or as a URL). [Yes]
   (b) All the training details (e.g., data splits, hyperparameters, how they were chosen). [Yes]
   (c) A clear definition of the specific measure or statistics and error bars (e.g., with respect to the random seed after running experiments multiple times). [Yes]
   (d) A description of the computing infrastructure used. (e.g., type of GPUs, internal cluster, or cloud provider). [Yes]

4. If you are using existing assets (e.g., code, data, models) or curating/releasing new assets, check if you include:
   (a) Citations of the creator If your work uses existing assets. [Yes]
   (b) The license information of the assets, if applicable. [Yes]
   (c) New assets either in the supplemental material or as a URL, if applicable. [Yes]
   (d) Information about consent from data providers/curators. [Not Applicable]
   (e) Discussion of sensible content if applicable, e.g., personally identifiable information or offensive content. [Not Applicable]

5. If you used crowdsourcing or conducted research with human subjects, check if you include:
   (a) The full text of instructions given to participants and screenshots. [Not Applicable]
   (b) Descriptions of potential participant risks, with links to Institutional Review Board (IRB) approvals if applicable. [Not Applicable]
   (c) The estimated hourly wage paid to participants and the total amount spent on participant compensation. [Not Applicable]

Beyond Bayesian Model Averaging over Paths in Probabilistic Programs with Stochastic Support

A A Short Introduction to Probabilistic Programming

A.1 Programs as Unnormalized Densities

```
def model(data):
  x = sample("x", Normal(0, 1))
  m1 = sample("m1", Bernoulli(0.5))
  if m1:
    std = 0.62177
    with plate("data1", data.shape[0]):
      sample("y1", Normal(x, std1), obs=data)
  else:
    std = 2.0
    with plate("data2", data.shape[0]):
      sample("y2", Normal(x, std2), obs=data)
```

Figure 5: Example Pyro program with stochastic support.

We here give an informal description for how the probabilistic programming language Pyro (Bingham et al., 2019) defines unnormalized densities. For more formal description of the semantic foundations of probabilistic programming we refer the interested reader to Borgström et al. (2016); Staton et al. (2016) and Lew et al. (2019). Pyro provides the `sample` primitive function to both define latent random variables and condition on observed data. More precisely, calling the function `sample(addr, dist)` draws a sample from the distribution object `dist` with the corresponding lexical address `addr`. Conditioning on observed data is made possible by passing in the observed data to the `sample` function as a keyword argument like so `sample(addr, dist, obs=data)`, where again `addr` and `dist` are assumed to be a lexical address and distribution object. The users is responsible for ensuring that each `sample` statement gets assigned a unique address, i.e. an address that has not been encountered in the current execution, and that every lexical `sample` statement within the program has a distinct address. The second condition avoids the edge case that a program has multiple branches, each of which samples the same sequence of addresses. This condition is automatically satisfied in some PPS such as Anglican (Wood et al., 2014), however, in PPS in which users manually define addresses they are responsible for adhering to this constraint. An interesting avenue for future work is automatically verifying that this constraint is satisfied.

Pyro is a universal PPS as it is embedded within the Python programming language and users are free to use language features such as branching on the outcomes of `sample` statements. This means Pyro is able to express programs with stochastic support i.e. the number of `sample` statements and their corresponding distribution type can vary from one execution to the next. Fig. 5 gives an example of a Pyro program with stochastic support.

A Pyro program defines an unnormalized density over the *raw random draws*, $\theta_{1:n_\theta} \in \Theta$, which are defined as the sequence of outcomes of the `sample` statements (without any observed data) encountered in the program. Crucially, in models with stochastic support the number of `sample` statements n_x can be a random variable and is not fixed for a given program. Furthermore, the raw random draws are assumed to be the only source of randomness in the program, such that for given raw random draws $\theta'_{1:n_\theta}$ all the intermediate variables and return values can be determined deterministically.

Each `sample` statement encountered during the program's execution contributes one term to the program's unnormalized density $\gamma(\theta_{1:n_\theta})$. If a `sample` does not have associated observed data, then it contributes the term $f_{a_i}(\theta_i \mid \eta_i)$ where a_i is the lexical address, θ_i the outcome of the `sample` statement, f_{a_i} is the parameterized density

function of the associated distribution object, and η_i are the corresponding parameters. Similarly, a `sample` with observed data y_j contributes the term $g_{b_j}(y_j \mid \phi_j)$ with b_j denoting the lexical address, g_{b_j} the parameterized density function, and ϕ_j the parameters. Overall, as noted by (Rainforth, 2017, §4.3.2) who formalized this for the PPS Anglican (Wood et al., 2014), this implies the program's unnormalized density function is given by

$$\gamma(\theta_{1:n_\theta}) := \prod_{i=1}^{n_\theta} f_{a_i}(\theta_i \mid \eta_i) \prod_{j=1}^{n_y} g_{b_j}(y_j \mid \phi_j). \tag{17}$$

Inference in a probabilistic program corresponds to the task of finding a representation of the normalized density $\pi(\theta_{1:n}) = \gamma(\theta_{1:n})/Z$ with normalization constant $Z = \int_\Theta \gamma(\theta_{1:n_\theta}) d\theta_{1:n_\theta}$.

A.2 Approaches to Inference

For most problems encountered in practice we cannot exactly compute the normalized program density $\pi(\theta_{1:n_\theta})$ and instead have to resort to approximate inference algorithms. A variety of approaches for approximate inference in probabilistic programs with stochastic support exist. Particle-based approaches (Wood et al., 2014; Rainforth et al., 2016; Murray and Schön, 2020) and algorithms based on importance sampling (Le et al., 2017; Baydin et al., 2019; Harvey et al., 2019) generate a set of weighted samples as an approximate posterior. Markov chain Monte Carlo (MCMC) methods with either automatic or manual proposals try to generate a set of samples directly from the posterior (Wingate et al., 2011; Yang et al., 2014; Tolpin et al., 2015; Roberts et al., 2019; Cusumano-Towner et al., 2020; Mak et al., 2021, 2022). Variational inference algorithms create a parameterized distribution $q(\theta; \phi)$ and optimize the parameters ϕ such that $q(\theta; \phi)$ is "close" to the full Bayes posterior where closeness is measured with some divergence (most commonly the KL divergence) (Wingate and Weber, 2013; Ranganath et al., 2014; Paige, 2016).

We now give a more detailed description of involutive MCMC which is the foundation behind Gen's implementation of reversible-jump MCMC (RJMCMC) (Cusumano-Towner et al., 2020). In involutive MCMC the user specifies an auxiliary kernel κ which acts on an auxiliary variable $v \in Y$, s.t. $\kappa_\theta(\cdot) : Y \to [0, \infty)$ for all $\theta \in \Theta$ with $\gamma(\theta) > 0$, and an involution $\Phi : \Theta \times Y \to \Theta \times Y$, i.e. $\Phi^{-1} = \Phi$. Given an initial state θ, involutive MCMC proceeds by first sampling a new auxiliary variable $v \sim \kappa_\theta(\cdot)$, then applying the involution to get the newly proposed state $(\theta, v) \leftarrow \Phi(\theta, v)$, and accepting the new state with the acceptance probability

$$\alpha := \min\Big\{1, \frac{\gamma(\theta')\,\kappa_\theta(v)}{\gamma(\theta)\,\kappa_\theta(v)} |\det(\nabla \Phi(\theta, v))|\Big\}. \tag{18}$$

The usage of involutive MCMC is partly automated in Gen. The user only has to implement the auxiliary kernel and the involution using a domain-specific language. Based on these quantities Gen is then able to automatically construct an involutive MCMC kernel (Cusumano-Towner et al., 2020).

B Probabilistic Programs without Predictive Distribution

```
def model(y):
  # Input data is a list of length 2
  if y[0] > 10:
    x = sample("x1", Normal(10, 1))
    sample("y1", Normal(x, 10), obs=y[0])
    sample("y2", Normal(y[0], sqrt(y[0])), obs=y[1])
  else:
    x = sample("x2", Normal(0, 1))
    sample("y1", Normal(x, 1), obs=y[0])
    sample("y2", Normal(y[0], 1), obs=y[1])
```

Figure 6: Example of a probabilistic program without a predictive distribution.

Conventionally, in Bayesian statistics the modeller defines a prior $p(\theta)$ and predictive distribution $p(y_i \mid \theta)$. Then for a dataset $\mathbf{y} = (y_1, \ldots, y_N)$, these two ingredients together define the joint density $p(\theta, \mathbf{y}) = \prod_{i=1}^N p(y_i \mid \theta)\, p(\theta)$ from which can compute the posterior $p(\theta \mid \mathbf{y}) \propto p(\theta, \mathbf{y})$. In order to predict new data $\tilde{y}$, we can then use the

posterior predictive distribution defined as

$$p(\tilde{y} \mid \mathbf{y}) = \int p(\tilde{y} \mid \theta)\, p(\theta \mid \mathbf{y}) dx. \tag{19}$$

However, more generally, the joint density does not necessarily factorize in this manner which makes it more difficult to automatically deduce a prediction task from the model alone. Additionally, in the setting of universal probabilistic programming languages the input data can directly influence the model definition as well. Take for example the Pyro program in Fig. 6. Here, the input data y directly influences what `sample` statements we encounter during execution. Furthermore, in both `sample` statenents with address `"y2"` the distribution depends on the first data point `y[0]`. Therefore, just from this program definition alone, it is not clear what exactly a reasonable predictive distribution for a new data point would be.

C Introduction to Scoring Rules

This introduction mainly follows from Gneiting and Raftery (2007) and Yao et al. (2018). *Scoring rules* are functions which take as input a *probabilistic forecast* and a realized event. The goal of scoring rules is then to evaluate the quality of the probabilistic forecast. The terminology is specifically used in meteorological forecasts. In a Bayesian context, these scores are often instead referred to as utilities and Bayesian decision theory aims to maximize the predicted utility of a given action (Bernardo and Smith, 2009).

More formally, assume we have a random variable on the sample space $(\Omega, \mathcal{A})$ and $\mathcal{P}$ is a convex class of probability measures on $(\Omega, \mathcal{A})$. Then, any member $P \in \mathcal{P}$ is referred to as a probabilistic forecast and a scoring rule is a function $S : \mathcal{P} \times \Omega \to [-\infty, \infty]$ s.t. $S(P, \cdot)$ is $\mathcal{P}$-quasi-integrable for all forecasts $P \in \mathcal{P}$. So for a probabilistic forecast P and observed event $y \in \Omega$, $S(P, y)$ is the score which indicates the quality of our forecast. For notational convenience, if P and Q are both probabilistic forecasts, we define $S(P, Q) = \int S(P, y) dQ(y)$. Then a scoring rule is *proper* if $S(Q, Q) \geq S(P, Q)$ holds for all $P \in \mathcal{P}$, and *strictly proper* if the equality holds only when $P = Q$ almost surely.

Some common examples of scoring rules include: the *quadratic score* $S(\rho, y) = 2\rho(y) - \|\rho\|_2^2$, where ρ is a predictive density; the *logarithmic score* $S(\rho, y) = \log \rho(y)$; and the *continuous-ranked probability score* $S(F, y) = -\int (F(y') - \mathbb{I}[y' \geq y]) dy'$, where F is the cumulative distribution function of the forecast. Under regularity conditions, Bernardo (1979) showed that the logarithmic scoring rule is the only proper *local* scoring rule where a local scoring rule is a rule that depends on the predictive density ρ only through the actual observed event y.

We refer the reader to Gneiting and Raftery (2007) for more extensive details on scoring rules.

D PSIS-LOO Approximation

We here only give a brief description of the PSIS-LOO approximation as it is a common procedure. We refer the reader to Vehtari et al. (2017) for full details on the approximation and to Vehtari et al. (2015) for more details on PSIS. We want to approximate the local posterior predictive

$$\rho_k(y_i \mid y_{-i}) = \int_{\Theta_k} g_k(y_i \mid \theta)\, \pi_k(\theta \mid y_{-i}) d\theta \tag{20}$$

which can be rewritten in terms of the posterior of the full dataset as

$$= \int_{\Theta_k} g_k(y_i \mid \theta)\, \frac{\pi_k(\theta \mid y_{-i})}{\pi_k(\theta \mid y_{1:N})}\, \pi_k(\theta \mid y_{1:N}) d\theta. \tag{21}$$

Importantly, the ratio in that integral is proportional to a term that only depends on the individual predictive density

$$r_i := \frac{1}{g_k(y_i \mid \theta)} \propto \frac{\pi_k(\theta \mid y_{-i})}{\pi_k(\theta \mid y_{1:N})}. \tag{22}$$

Hence, we can use a self-normalized importance sampler to get an estimate of Eq. (21) as follows

$$\rho_k(y_i \mid y_{-i}) \approx \frac{\sum_{s \in I_k} r_i^s\, g_k(y_i \mid \theta_s)}{\sum_{s \in I_k} r_i^s} \tag{23}$$

where $r_i^s = 1/g_k(y_i \mid \theta_s)$ and θ_s are (approximate) samples from the posterior. However, these ratios r_i^s can often have high variance since $\pi_k(\theta \mid y_{1:N})$ will usually have lower variance and thinner tails than $\pi_k(\theta \mid y_{-i})$, in turn, leading to unstable estimates. To avoid these instabilities, the PSIS-LOO approximation replaces the ratios r_i^s with smoothed importance weights ν_i^s. The importance weights are computed by fitting a generalized Pareto distribution to the raw ratios r_i^s and replacing them with the expected order statistics of the fitted Pareto distribution. This then leads to the estimate

$$\hat{\rho}_k(y_i \mid y_{-i}) = \frac{\sum_{s \in I_k} \nu_i^s \, g_k(y_i \mid \theta_s)}{\sum_{s \in I_k} \nu_i^s}. \tag{24}$$

E Implementation details

```
def pyro_subset_regression(X, y, X_val, y_val):
    k = pyro.sample(
        "k", dist.Categorical(torch.ones(X.shape[1]) / X.shape[1]),
        infer={"branching": True},
    )
    X = X[:, k]
    beta = pyro.sample(f"beta_{k.item()}", dist.Normal(0, np.sqrt(10)))
    sigma = pyro.sample("sigma", dist.Gamma(0.1, 0.1))
    mean = X * beta
    with pyro.plate("data", X.shape[0]):
        pyro.sample("y", dist.Normal(mean, sigma), obs=y)

    X_val = X_val[:, k]
    mean_val = X_val * beta
    return dist.Normal(mean_val, sigma).log_prob(y_val)
```

Figure 7: Pyro program for the experiments in Sec. 6.2.

To be able to evaluate the stacking objective defined in Eq. (15) we need access to $\hat{\rho}_k(\tilde{y}_\ell)$, the estimates of the predictive densities. These, in turn, depend on the evaluations of the predictive densities. Hence, given validation data $\tilde{y}_1, \ldots, \tilde{y}_L$ and posterior samples $\theta_1, \ldots, \theta_S$, for stacking we require the evaluations $g(\tilde{y}_\ell \mid \theta_s)$. Note that the local posterior predictive distributions $\rho_k(\tilde{y}_\ell)$ are expectations under the posterior.

This is important because previous work noted that in the context of probabilistic programming these types of expectations can be formalized as the *expected return values* of a program (Gordon et al., 2014; Zinkov and Shan, 2017; Reichelt et al., 2022b; Lew et al., 2023). Then to apply stacking, we require the user to define a program in which the return values for a given sample θ_s are the predictive density evaluations $g(\tilde{y}_1 \mid \theta_s), \ldots, g(\tilde{y}_L \mid \theta_s)$. For our Pyro implementation this means that we require the user to define a program which returns an L-dimensional vector which correspond to the (log) predictive density on the validation data points. Fig. 7 shows how this can be done for the subset regression model from Sec. 6.2.

To actually compute the stacking weights we rely on Pyro's `Trace`[3] data structure which saves important metadata of each program execution such as the distribution type of each `sample` statement, the corresponding sampled value, its address, and, crucially, also the return value of the program. Algorithm 1 can then be implemented as a simple function which takes as input a list of posterior samples in the form of Pyro `Trace`s. Because each `Trace` stores the address path of a given run, we can easily determine the set of SLPs and associate each sample with its SLP. From the `Trace` data structure we can then also extract the return values i.e. the evaluations $\log g(\tilde{y}_\ell \mid \theta_s)$. From these evaluations we can calculate the estimates $\hat{\rho}_k(\tilde{y}_\ell)$ which are needed in our stacking objective in Eq. (15). The optimization of the objective is done using SciPy's (Virtanen et al., 2020) implementation of the L-BFGS-B optimizer (Liu and Nocedal, 1989).

Similarly, to Zhou et al. (2020) and Reichelt et al. (2022a) we also allow the user to annotate specific discrete sampling statements to indicate that they influence the SLP of the program. In our implementation, this is done by passing `{"branching": True}` to the `infer` keyword argument of Pyro's `sample` statement. Using this annotation the inference backend can then control which SLP of the program is sampled by conditioning these `sample`

[3] https://docs.pyro.ai/en/stable/poutine.html#trace

statements on specific values. Furthermore, these annotations allow the inference backend to deterministically enumerate all SLPs. In our implementation, we give the user to enumerate all SLPs using breadth-first search, i.e. in the enumeration procedure we run the program as normal but if we encounter an annotated sampling statement we enumerate the whole support of the distribution and put each possible sampled value onto a queue. Once enumeration of a sample site has finished, execution resumes by popping a value from the front of the queue and continuing executing the program from the `sample` statement from which the value was sampled. This enumeration procedure is similar to how marginalization of discrete sample sites is implemented in Pyro. Note that we are assuming that each annotated discrete `sample` site has finite support. Future implementations could deal with finite support by truncating the sampling distribution.

E.1 Alternative Interface for Stacking

```
def model_average(models, model_args, model_kwargs):
    k = pyro.sample(
        "k", dist.Categorical(torch.ones(len(models)) / len(models)),
        infer={"branching": True},
    )
    return models[k](*model_args, **model_kwargs)
```

Figure 8: Alternative interface to enable stacking of a list of user-specified models.

The main aim of our paper is to highlight the shortcomings of the default BMA weights in posteriors of probabilistic programs with stochastic support. However, as a side-effect our Pyro implementation also provides a convenient mechanism for users to utilize stacking. Probabilistic programs with stochastic support allow users to encode a large class of problem settings, including the traditional BMA setting. However, in some cases, for convenience, users might not wish to write a single program with stochastic support which encodes all the different models in the BMA. Instead, the user might want to write K separate programs, each with static support, and then want to apply either BMA or stacking to that list of models. However, this interface is easily encoded within our framework because the user only needs to write another program which combines all the models together. This program is shown in Fig. 8, it takes as input a list of Pyro models, and the arguments that should be passed to these models. The program then samples an index k and chooses one of the candidate models. Running standard inference in this program would correspond to BMA. If the user instead wants to run stacking instead of BMA, they can simply choose our stacking implementation and apply it to this program.

F Experimental Details and Additional Results

To make the stacking approach widely applicable we implemented a version of the DCC framework in Pyro (Bingham et al., 2019). For our implementation of DCC, we assume that all the stochastic branching happens on variables with discrete support and that the remaining latent variables in the program are continuous. Models of this form then permit the usage of Pyro's built-in Hamiltonian Monte Carlo (Neal, 2011; Hoffman et al., 2014; Betancourt, 2018) as the local inference algorithm. To improve efficiency, our implementation also leverages Pyro's just-in-time (JIT) compilation ability to compile each SLP. Note that leveraging Pyro's JIT compilation is possible due to breaking down the original program into its SLPs because by default the JIT compilation cannot handle programs with stochastic support.

Our experiments were conducted on an internal compute cluster which consists of a mix of Intel Broadwell, Haswell, and Cascade Lake CPUs. In general, we use 16 cores to parallelize our computation and running a single replication of an experiment finishes in a matter of minutes if not seconds. An exception is the function induction experiment in Sec. 6.3 which takes around 3 hours for a single replication, using 32 cores.

F.1 When is Stacking Helpful?

For each setting we collect 10^3 HMC samples from each SLP with 400 burn-in samples. To estimate the normalization constant for the dominant SLP setting we use a proposal of $q(\theta_1, \theta_2) = \mathcal{N}(\theta_1; 0, 1)\,\Gamma(\theta_2; 1, 1)$ with 10^6 samples.

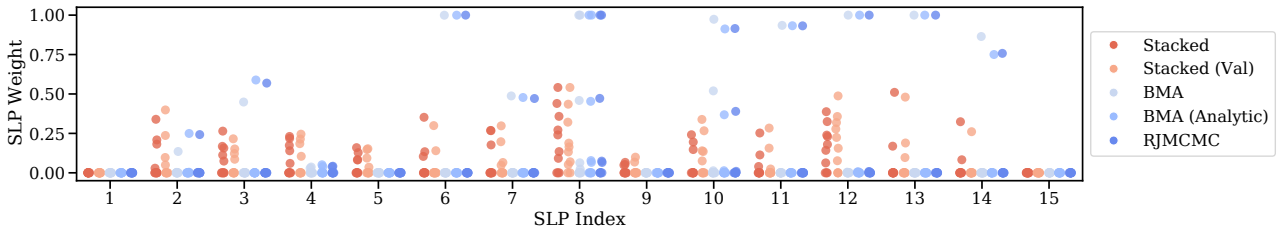


Figure 9: SLP weights for model in Sec. 6.2. Each dot represents the weight of the corresponding SLP in the model. Results are computed over 50 randomly generated datasets.

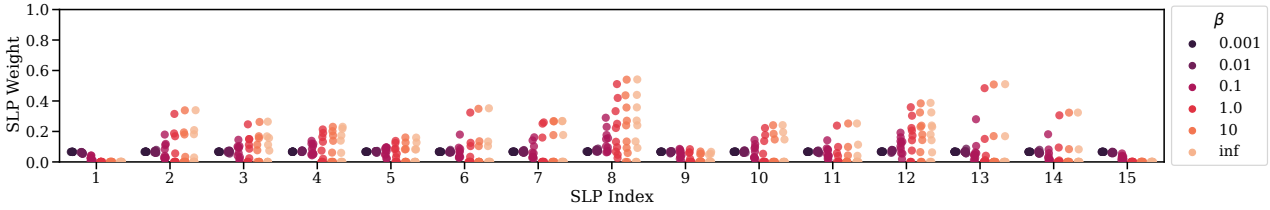


Figure 10: Impact of varying regularization parameter β on the computed SLP weights. Smaller values of β lead to more regularization, pushing the SLP weights more towards the uniform distribution over SLPs.

Overall, the experiments show that stacking is particularly useful if there are multiple SLPs which fit the data roughly equally well (with respect to the local normalization constant). In these cases, the SLP weights are at risk of being unstable and stacking can provide a more robust weight estimation procedure. On the other hand, if there is one dominant SLP which clearly performs better than all the other SLPs then we would expect there to be less to gain from stacking, as the weights will already be fairly stable.

F.2 Subset Regression

Following Breiman (1996) and Yao et al. (2018), we generate the regression coefficients according to $\beta_d = \eta\left(\zeta_d(4) + \zeta_d(8) + \zeta_d(12)\right)$ with $\zeta_d(a) := \mathbb{I}\left[|d-a<h|\right]\,(h-|d-a|)^2$. The parameter h determines the number of "strong" coefficients. Following Yao et al. (2018) we set $h=5$ which leads to 15 weak coefficients and set η such that the signal-to-noise ratio $\mathbb{V}\left[\sum_{d=1}^{15}\beta_d X_d\right]/(1+\mathbb{V}\left[\sum_{d=1}^{15}\beta_d X_d\right]) = 4/5$ where X_d is the random variable for the dth covariate. Our input program has 15 SLPs and each SLP has the unnormalized density $\gamma_k(\theta_1,\theta_2,\theta_3) = \mathbb{I}\left[\theta_3=k\right]\prod_{i=1}^{N}\mathcal{N}(y_i;\theta_1 x_{i,k},\theta_2^2)\,\mathcal{N}(\theta_1;0,10)\,\Gamma(\theta_2;0.1,0.1)\,\text{DiscreteUniform}(\theta_3;1,15)$. For DCC inference, for each SLP we collect 10^3 HMC samples with 400 burn-in samples. For RJMCMC our transition kernel samples a new θ_3, the variable controlling the covariate that is selected, from a uniform categorical distribution and new θ_1, the local regression coefficient, from a standard normal distribution. The noise variable is independently updated using a Metropolis-Hastings kernel. The individual weights for each SLP are shown in Fig. 9.

F.3 Function Induction

We are using the probabilistic context-free grammar $e \to \{x \mid \sin(a*e) \mid a*e+b*e\}$. In our model, we use prior production probabilities $[0.4, 0.4, 0.2]$ and for each of the sampled coefficients (denoted a and b in the PCFG) we use a $\mathcal{N}(0,10)$ prior. Using the PCFG we can sample an expression for a function $f:\mathbb{R}\to\mathbb{R}$, then informally our model can be written as

$$f \sim \text{PCFG}(); \quad \sigma \sim \Gamma(\sigma;1,1); \quad y_i \sim \mathcal{N}(f(x_i),\sigma^2) \;\text{ for }\; i=1,\ldots,N. \tag{25}$$

For DCC inference, we run 4 chains with 500 HMC samples and 200 burn-in samples for each SLP. The sampling distribution of SLP weights are shown in Fig. 11. For RJMCMC, our transition kernel represents each expression as a PCFG parse tree. To propose a new expression, the transition kernel picks a random node in the parse tree and replaces that node with a sampled expression from the prior PCFG. The standard deviation of the likelihood, σ, is updated independently with a Metropolis-Hastings transition kernel. For RJMCMC, we collect the same number of total samples as for DCC to ensure a fair comparison.

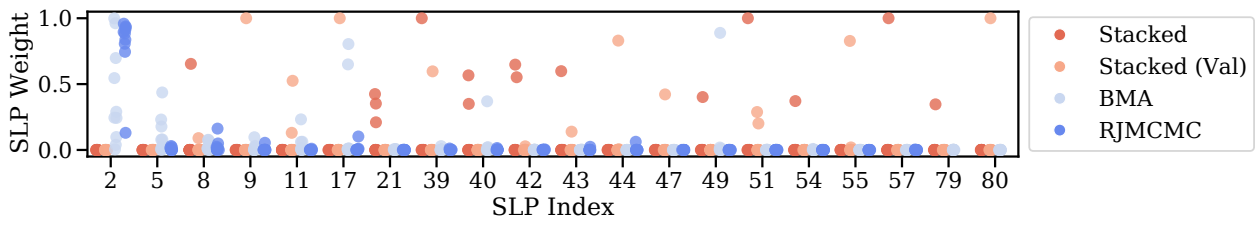


Figure 11: SLP weights for the function induction model with the misspecified PCFG. We only plot SLPs which have achieved a weight > 0.3 in any run for any method.

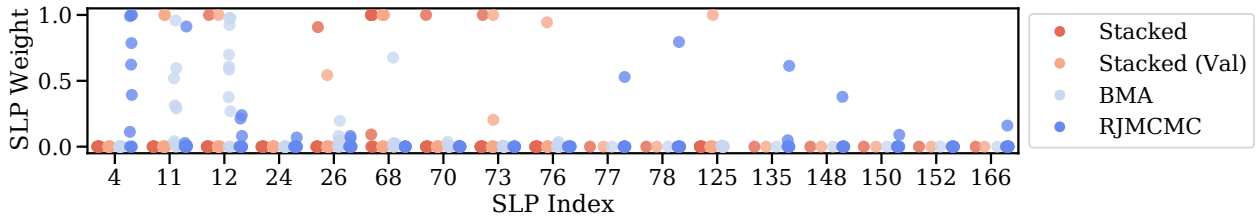


Figure 12: SLP weights for the function induction model with the well-specified PCFG. We only plot SLPs which have achieved a weight > 0.3 in any run for any method. The SLP indices 11 and 12 both correspond to the true function (due to the symmetry in the $+$ operator there are two SLPs which represent the true function).

Due to recursion in the PCFG rules, the program actually defines an infinite number of SLPs. Similar to Zhou et al. (2020), to avoid infinite recursion we restrict the underlying inference engine to only consider a finite number of SLPs. Our inference engine enumerates the possible PCFG expressions using breadth-first search and only considers the first 128 expressions.

F.4 Variable Selection

```
def variable_selection_model(X, y,):
    num_features = X.shape[1]
    features_included = torch.zeros((num_features,), dtype=torch.bool)
    for ix in range(num_features):
        features_included[ix] = pyro.sample(
            f"feature_{ix}", dist.Bernoulli(0.5), infer={"branching": True}
        )
    X_selected = X[:, features_included]
    num_selected = X_selected.shape[1]
    noise_var = pyro.sample("noise_var", dist.InverseGamma(2.0, 1.0))
    with pyro.plate("features", num_selected):
        w = pyro.sample("weights", dist.Normal(0, noise_var.sqrt()))
    means = w @ X_selected.T
    with pyro.plate("data", y.shape[0]):
        pyro.sample("obs", dist.Normal(means, noise_var.sqrt()), obs=y)
```

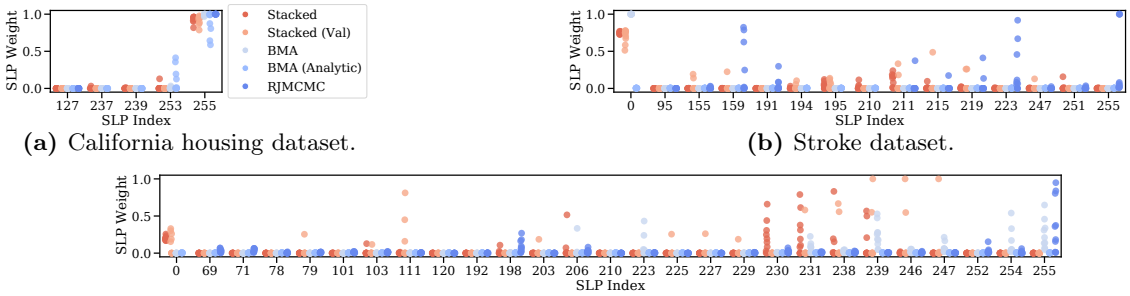
Figure 13: Pyro program for the variable selection experiments.

We assume we are given data $y_{1:N}$ and an associated matrix of covariates $X \in \mathbb{R}^{N \times D}$. We consider both regression and classification problems. The problem of variable selection is to find a subset of the features $\mathcal{D} \subseteq \{1, \dots, D\}$ to make predictions given our data. For regression task, we have $y_i \in \mathbb{R}$ and our model for a specific subset $\mathcal{D}$ of the features is given by

$$\sigma^2 \sim \Gamma^{-1}(2, 1), \tag{26}$$

$$\beta_d \sim \mathcal{N}(0, \sigma^2) \quad \text{for } d \in \mathcal{D}, \tag{27}$$

$$y_i \sim \mathcal{N}(\sum\nolimits_{d \in \mathcal{D}} \beta_d x_{i,d}, \sigma^2). \tag{28}$$



(a) California housing dataset.

(b) Stroke dataset.

(c) Diabetes dataset.

Figure 14: SLP weights for the models in Sec. 6.4 (conventions as in Fig. 11). The Stroke and Diabetes problems do not permit an analytic solution to the BMA weights.

This form of the regression model allows us to analytically calculate the marginal likelihood (see e.g. Ch. 3.5 in Bishop and Nasrabadi (2006)). For classification tasks, we instead have $y_i \in \{0, 1\}$ and use the logistic regression model

$$\beta_d \sim \mathcal{N}(0, 1), \text{ for } d \in \mathcal{D} \text{ and } y_i \sim \mathcal{B}(S(\sum\nolimits_{d\in\mathcal{D}} \beta_d x_{i,d})),$$

where $S(x) = 1/(1 + \exp(-x))$ and $\mathcal{B}$ is the Bernoulli distribution.

The three different datasets we consider are: *California* housing ($N = 20,650$, $D = 8$; 50 % train, 50 % test) (Pace and Barry, 1997), a regression dataset where the goal is to predict the median house prices for districts of California; *Diabetes* ($N = 768$, $D = 8$; 80 % train, 20 % test) (Smith et al., 1988), a classification dataset on if a person has diabetes or not; and *Stroke* ($N = 4908$, $D = 8$; 80 % train, 20 % test), a classification dataset on if a person will have a stroke or not. For DCC inference, we collect 10^3 HMC samples with 400 burn-in samples for each SLP. The sampling distribution of the SLP weights are plotted in Fig. 14. For RJMCMC, the transition kernel randomly selects a feature dimension d and for that dimension flips the inclusion, i.e. if the feature was previously included it will be excluded and vice versa. For a previously excluded feature, a new coefficient is sampled from the prior. For all other coefficients, a new coefficient is proposed from a standard normal centered at the current value. To ensure a fair comparison to DCC, we collect the same number of samples from RJMCMC as we do for DCC.

F.5 Modelling Radon Contamination in US Counties

As mentioned in the main text, our program encodes different modelling choices for both the intercept(s), α, and the slope parameter(s), β, of the regression relation $y_i = \alpha + \beta\, x_i$. Below we describe in detail the four different modelling choices for the intercept term. The modelling choices for the slope parameter are analogous, except we do not consider using a group-level predictor for β. The full Pyro program for this experiment is shown in Fig. 15.

Pooling. This model corresponds to the SLP denoted "P, P" in Fig. 3.

$$\alpha \sim \mathcal{N}(0, 10), \quad \beta \sim \mathcal{N}(0, 10), \quad f_i = \alpha + \beta\, x_i, \quad \sigma \sim \text{Exponential}(5), \quad y_i \sim \mathcal{N}(f_i, \sigma^2). \tag{29}$$

No pooling. Here $c[i]$ refers to the county index of the ith house. This model corresponds to the SLP denoted "NP, P" in Fig. 3.

$$\alpha_c \sim \mathcal{N}(0, 10) \text{ for each county } c, \quad \beta \sim \mathcal{N}(0, 10), \quad \sigma \sim \text{Exponential}(5), \quad y_i \sim \mathcal{N}(f_i, \sigma^2) \tag{30}$$

with $f_i = \alpha_{c[i]} + \beta\, x_i$.

Hierarchical. This model corresponds to the SLP denoted "H, P" in Fig. 3. We are using a non-centered parameterization to allow for better sampling performance from the HMC sampler.

$$\sigma_\alpha \sim \text{Exponential}(1), \quad \mu_\alpha \sim \mathcal{N}(0, 10), \quad \epsilon_c \sim \mathcal{N}(0, 1) \quad \text{for each county } c, \quad \alpha_c = \mu_\alpha + \sigma_\alpha\, \epsilon_c, \tag{31}$$

$$\beta \sim \mathcal{N}(0, 10), \quad \sigma \sim \text{Exponential}(5), \quad f_i = \alpha_{c[i]} + \beta\, x_i, \quad y_i \sim \mathcal{N}(f_i, \sigma^2) \tag{32}$$

Group-level predictor. This model corresponds to the SLP denoted "G, P" in Fig. 3.

$$\gamma_0 \sim \mathcal{N}(0, 10), \qquad \gamma_1 \sim \mathcal{N}(0, 10), \qquad \sigma_\alpha \sim \text{Exponential}(1), \qquad \epsilon_c \sim \mathcal{N}(0, 1) \quad \text{for each county } c, \tag{33}$$

$$\alpha_c = \mu_{\alpha,c} + \sigma_\alpha \, \epsilon_c, \qquad \beta \sim \mathcal{N}(0, 10), \qquad \sigma \sim \text{Exponential}(5), \qquad y_i \sim \mathcal{N}(f_i, \sigma^2) \tag{34}$$

where $\mu_{\alpha,c} = \gamma_0 + \gamma_1 \, u_c$ and $f_i = \alpha_{c[i]} + \beta \, x_i$.

To ensure we have a balanced representation of data in each county in both the training and the testing data we apply stratified sampling: for each county, we hold out 20 % of the observations for evaluation. For this dataset we do not run stacking with a validation set because of the limited amount of data available per county. For DCC inference, we collect 2000 HMC samples with 2000 samples for burn-in for each SLP. For RJMCMC, the transition kernel picks a modelling choice of α and β and then samples the local parameters for each modelling choice from the prior. The standard deviation σ is updated separately with a Metropolis-Hastings transition kernel. To ensure a fair comparison between DCC and RJMCMC we collect the same total number of samples.

F.6 Stacked RJMCMC

We conducted further experiments to determine the impact of running stacking on top of RJMCMC, we call this *Stacked RJMCMC*. The results can be viewed in Fig. 16 where we plot the difference in LPPD between RJMCMC and other methods, i.e. $\text{LPPD}_{\text{Diff}} = \text{LPPD}_{\text{Other}} - \text{LPPD}_{\text{RJMCMC}}$. Note, compared to the main paper we here evaluate the difference in LPPD to RJMCMC and not to stacking on top of DCC. This is because we care about investigating the performance improvement relative to RJMCMC. We observe in Fig. 16 that Stacked RJMCMC generally leads to higher LPPD than RJMCMC. The difference is positive for all problem settings. Further, the difference is statisticaly significant under a Wilcoxon-signed rank test in all problems, except for *California* and *Stroke*.

G PAC-Bayes and Stacking

For completeness, we first give a brief introduction into PAC-Bayes which is mostly based on Morningstar et al. (2022). Morningstar et al. (2022) assume a setting in which data is sampled i.i.d. from $\tilde{y}_\ell \sim p_{\text{true}}(\tilde{y})$, that we have a parameterized probability model $p(\tilde{y} \mid \theta)$ and we want to find a mechanism to fit a distribution, $q(\theta)$, over the parameters of the probability model.

The **true predictive risk** is given by

$$\mathcal{P}(q) := -\mathbb{E}_{p_{\text{true}}(\tilde{y})}[\log \mathbb{E}_{q(\theta)}[p(\tilde{y} \mid \theta)]] \tag{35}$$

and in most applications is the quantity we care about in the end because it directly measures the quality of our predictions. However, in practice we cannot evaluate the true predictive risk because we do not have access to the true data generating distribution. The goal of PAC-Bayes methods is then to provide a stochastic upper bound on $\mathcal{P}(q)$ which can be used to train $q(\theta)$ (Masegosa, 2020; Morningstar et al., 2022).

The **empirical predictive risk**

$$\overline{\mathcal{P}}(q) := -\frac{1}{L} \sum_{\ell=1}^{L} \log \mathbb{E}_{q(\theta)}[p(\tilde{y}_\ell \mid \theta]. \tag{36}$$

is an empirical estimate of the true predictive risk. *Ensemble methods (which include stacking) directly minimize the empirical predictive risk.* For example, our stacking objective in Eq. (15) is a particular instantiation of the empirical predictive risk. We will explain the connection in more detail below.

The **true inferential risk**

$$\mathcal{R}(q) := -\mathbb{E}_{p_{\text{true}}(\tilde{y})}[\mathbb{E}_{q(\theta)}[\log p(\tilde{y} \mid \theta)]] \tag{37}$$

is an *upper bound on the true predictive risk* (by applying Jensen's inequality). In the case that our model is well specified, i.e. $\exists \theta'$ s.t. $p_{\text{true}}(\cdot) = p(\cdot \mid \theta')$, then $\operatorname{argmin} \mathcal{P}(q) = \operatorname{argmin} \mathcal{R}(q)$. Hence, when our model is well specified minimizing the true inferential risk is equivalent to minimizing the true predictive risk.

The **empirical inferential risk**

$$\overline{\mathcal{R}}(q) := -\frac{1}{L} \mathbb{E}_{q(\theta)}[\log p(\tilde{y}_\ell \mid \theta)] \tag{38}$$

is the empirical estimate of the true inferential risk. *Minimizing this risk directly is equivalent to maximum likelihood estimation.* By adding an extra regularization term to the empirical inferential risk we get the **PAC-inferential risk**, given by

$$\widetilde{\mathcal{R}}(q; r, \beta) := \mathbb{E}_{q(\theta)}\left[-\frac{1}{L}\sum_{\ell=1}^{L}\log p(\tilde{y}_\ell \mid \theta) + \frac{1}{\beta L}\log\frac{q(\theta)}{r(\theta)}\right] \tag{39}$$

where $r(\theta)$ is a user specified prior distribution on the parameters θ. This is a stochastic upper bound on the true predictive risk.

Morningstar et al. (2022) use results from Burda et al. (2015) to tighten the PAC-inferential risk and introduce **the PAC^M bound**

$$\widetilde{\mathcal{P}}_{M,L}(q; r, \beta) := -\frac{1}{L}\sum_{\ell=1}^{L}\mathbb{E}_{q(\theta^M)}\left[\log\left(\frac{1}{M}\sum_{j=1}^{M}p(\tilde{y}_\ell \mid \theta_j)\right)\right] + \frac{1}{\beta L}\mathrm{KL}(q(\theta) \parallel r(\theta)). \tag{40}$$

Notably, as the PAC-inferential risk, this bound contains a regularization term which is meant to prevent overfitting.

Theorem G.1 (Morningstar et al. (2022))**.** *For all $q(\theta)$ absolutely continuous with respect to $r(\theta)$, $\tilde{y}_\ell \sim p_{true}(\tilde{y})$ i.i.d., $\beta \in (0, \infty)$, $L, M \in \mathbb{N}$, $p(\tilde{y} \mid \theta) \in (0, \infty)$ for all $\{\theta \in \Theta \mid p_{true}(\tilde{y}) > 0\} \times \{\theta \in \Theta \mid r(\theta) > 0\}$, and $\xi \in (0, 1)$, then with probability at least $1 - \xi$,*

$$\mathcal{P}(q) \leq \widetilde{\mathcal{P}}_{M,L}(q; r, \beta) + \psi(p_{true}, \beta, M, L, r, \xi) - \frac{1}{\beta\, M\, L}\log\xi \tag{41}$$

and furthermore (unconditionally)

$$\widetilde{\mathcal{P}}_{M+1,L}(q; r, \beta) \leq \widetilde{\mathcal{P}}_{M,L}(q; r, \beta) \tag{42}$$

where $\widetilde{\mathcal{P}}_{M,L}(q; r, \beta)$ as in Eq. (40) and:

$$\psi(p_{true}, \beta, M, L, r, \xi) := \frac{1}{\beta\, M\, L}\log\mathbb{E}_{p_{true}(\tilde{y}^L)}\mathbb{E}_{r(\theta^M)}\left[\exp(\beta L M \Delta(\tilde{y}^L, \theta^M))\right], \tag{43}$$

$$\Delta(\tilde{y}^L, \theta^M) := \frac{1}{L}\sum_{\ell=1}^{L}\log\left(\frac{1}{M}\sum_{j=1}^{M}p(\tilde{y}_\ell \mid \theta_j)\right) - \mathbb{E}_{p_{true}(\tilde{y})}\left[\log\left(\frac{1}{M}\sum_{j=1}^{M}p(\tilde{y} \mid \theta_j)\right)\right]. \tag{44}$$

For a proof see Appendix C.3 of Morningstar et al. (2022).

G.1 From PAC-Bayes to Regularized Stacking

In order to connect the ideas from Morningstar et al. (2021) with the stacking objective for probabilistic programs we need to define a specific form for the parameterized probability model and the distribution q which is meant to be optimized. We choose the latent variable of our probabilitiy model to be the random variable k which indexes into the SLPs. Our probabilistic model then turns out to be $p(\tilde{y} \mid k) = \rho_k(\tilde{y})$ and our approximate posterior $q(k)$ is a categorical distribution over $\{1, \ldots, K\}$ parameterized by the weights w, i.e. $q(k) = \text{Categorical}(w_1, \ldots, w_K)$. In this formulation the true predictive risk is given by

$$\mathcal{P}(q) = -\mathbb{E}_{p_{\text{true}}(\tilde{y})}[\log\mathbb{E}_{q(k)}[\rho_k(\tilde{y})]] \tag{45}$$

$$= -\mathbb{E}_{p_{\text{true}}(\tilde{y})}\left[\log\left(\sum_{k=1}^{K}w_k\rho_k(\tilde{y})\right)\right] \tag{46}$$

which is equivalent to Eq. (11) in the main paper. The PAC^M bound becomes

$$\widetilde{\mathcal{P}}_{M,L}(q; r, \beta) = -\frac{1}{L}\sum_{\ell=1}^{N}\mathbb{E}_{q(k^M)}\left[\log\left(\frac{1}{M}\sum_{j=1}^{M}\rho_{k_j}(\tilde{y}_\ell)\right)\right] + \frac{1}{\beta L}\mathrm{KL}(q(k) \parallel r(k)). \tag{47}$$

Note that $\beta = 1$ can be interpreted as the "standard Bayesian" setting as it provides an equal weighting of the prior term $\mathrm{KL}(q(k) \parallel r(k))$ and likelihood term $\sum_{\ell=1}^{N}\mathbb{E}_{q(k^M)}\left[\log\left(\frac{1}{M}\sum_{j=1}^{M}\rho_{k_j}(\tilde{y}_\ell)\right)\right]$. We can rewrite the sum

inside the log as a sum over SLPs, as follows

$$\widetilde{\mathcal{P}}_{M,L}(q;r,\beta) = -\frac{1}{L}\sum_{\ell=1}^{L}\mathbb{E}_{q(k^M)}\left[\log\left(\sum_{k=1}^{K}\frac{|I_k|}{M}\rho_k(\tilde{y}_\ell)\right)\right] + \frac{1}{\beta L}\mathrm{KL}(q(k)\parallel r(k)) \tag{48}$$

where $I_k := \{k_j \mid j = \{1,\dots,M\}, k_j = k\}$ is the set of all parameter samples that are equal to k. Now by the law of large numbers (LLN) as $m \to \infty$ this converges to

$$\widetilde{P}_{\infty,L}(q;r,\beta) = R_\beta(w_{1:K}) = -\frac{1}{L}\sum_{\ell=1}^{L}\log\left(\sum_{k=1}^{K} w_k\rho_k(\tilde{y}_\ell)\right) + \frac{1}{\beta L}\mathrm{KL}(q(k)\parallel r(k)). \tag{49}$$

The first term in this objective is now the stacking objective and the second term is a regularization term pushing the distribution over weights to the prior $r(k)$. In other words, **this is Eq. (15) with an added regularization term.**

There is one caveat with pushing $M \to \infty$, as shown by Morningstar et al. (2021) the slack term ψ that controls the tightness of the bound grows linearly in M. Hence, the bound becomes vacuous for infinite M. However, there are several reasons why in our setting it is still reasonable to work with the objective $\widetilde{P}_{\infty,L}(q;r,\beta)$. First of all, in our specific setting the special case $M = 1$ is the degenerate setting in which we collapse onto a single SLP. This is exactly the behaviour we are trying to avoid in the first place. Additionally, Morningstar et al. (2022) have shown that performance increases with using larger M and show empirically that using $\widetilde{P}_{\infty,L}(q;r,\beta)$ can be beneficial when it is possible to do so. The only practical consideration they mention for increasing M are issues with gradient variance which are not applicable in our setting.

To choose the prior one could be inclined to use the posterior SLP weights $Z_k/\sum_{k'} Z_{k'}$, as they are our Bayesian beliefs about which SLP has generated the data. However, as we have argued in the main text these weights can be very unstable and expensive to estimate. Hence, a reasonable default choice could be to use the discrete uniform distribution. This will further discourage stacking from collapsing onto a single SLP. For the special case of the prior $r(k)$ being chosen to be the uniform distribution the objective further simplifies to

$$R_\beta(w_{1:K}) = \frac{1}{L}\sum_{\ell=1}^{L}\log\left(\sum_{k=1}^{K} w_k\rho_k(\tilde{y}_\ell)\right) - \frac{1}{\beta L}\left(\mathrm{H}[q(k;w_{1:K})] + \log K\right) \tag{50}$$

where H denotes the Shannon entropy.

```
def radon_model(log_radon, floor_ind, county, num_counties, uranium):
    alpha_choice = pyro.sample(
        "alpha_choices", dist.Categorical(torch.ones(4) / 4), infer={"branching": True}
    )
    if alpha_choice == 0:
        # Pooled model
        alpha = pyro.sample("alpha", dist.Normal(0, 10))
    elif alpha_choice == 1:
        # County specific intercepts
        with pyro.plate("num_alpha", num_counties):
            alpha = pyro.sample("alpha", dist.Normal(0, 10))
        alpha = alpha[..., county]  # Shape: (num_counties,) -> (num_data,)
    elif alpha_choice == 2 or alpha_choice == 3:
        if alpha_choice == 2:
            # Partially pooled model
            mean_a = pyro.sample("mean_a", dist.Normal(0, 1))
        elif alpha_choice == 3:
            # Uranium context
            gamma_0 = pyro.sample("gamma_0", dist.Normal(0, 10))
            gamma_1 = pyro.sample("gamma_1", dist.Normal(0, 10))
            mean_a = gamma_0 + gamma_1 * uranium

        std_a = pyro.sample("std_a", dist.Exponential(1))
        with pyro.plate("num_alpha", num_counties):
            z_a = pyro.sample("z_a", dist.Normal(0, 1))
        alpha = mean_a + std_a * z_a
        alpha = alpha[..., county]  # Shape: (num_counties,) -> (num_data,)

    beta_choice = pyro.sample(
        "beta_choices", dist.Categorical(torch.ones(3) / 3), infer={"branching": True}
    )
    if beta_choice == 0:
        # Pooled model
        beta = pyro.sample("beta", dist.Normal(0, 10))
    elif beta_choice == 1:
        # County specific slopes
        with pyro.plate("num_beta", num_counties):
            beta = pyro.sample("beta", dist.Normal(0, 10))

        beta = beta[..., county]  # Shape: (num_counties,) -> (num_data,)
    elif beta_choice == 2:
        # Partially pooled model
        mean_b = pyro.sample("mean_b", dist.Normal(0, 1))
        std_b = pyro.sample("std_b", dist.Exponential(1))
        with pyro.plate("num_beta", num_counties):
            z_b = pyro.sample("z_b", dist.Normal(0, 1))
        beta = mean_b + std_b * z_b
        beta = beta[..., county]  # Shape: (num_counties,) -> (num_data,)

    theta = alpha + beta * floor_ind
    sigma = pyro.sample("sigma", dist.Exponential(5))
    with pyro.plate("data", log_radon.shape[0]):
        pyro.sample("ys", dist.Normal(theta, sigma), obs=log_radon)
```

Figure 15: Pyro program for the radon model.

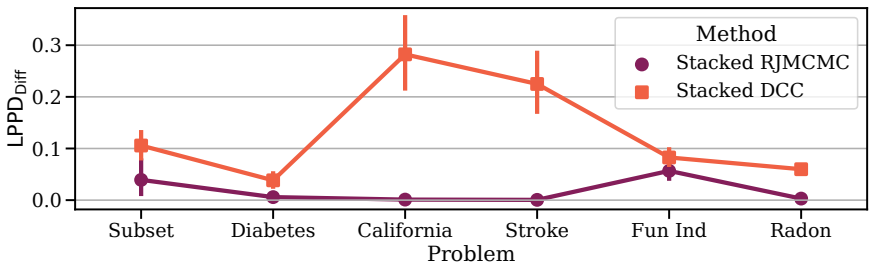


Figure 16: Difference in LPPD between RJMCMC and other methods (higher is better).

```
def distinct(y):
    model1 = pyro.sample("model1", dist.Bernoulli(0.5))
    if model1:
        z = pyro.sample("z1", dist.Normal(0, 1))
        with pyro.plate("data", y.shape[0]):
            pyro.sample("obs", dist.Normal(z, 0.62), obs=y)
    else:
        z = pyro.sample("z2", dist.Normal(0, 1))
        with pyro.plate("data", y.shape[0]):
            pyro.sample("obs", dist.Normal(z, 2.0), obs=y)

def overlap(X, y):
    model1 = pyro.sample("model1", dist.Bernoulli(0.5))
    if model1:
        w = pyro.sample(
            "w1", dist.Normal(0, 1).expand([2]).to_event(1),
        )
        mean = w @ X[:, [0, 2]].T
    elif model2:
        w = pyro.sample(
            "w2", dist.Normal(0, 1).expand([2]).to_event(1),
        )
        mean = w @ X[:, [0, 3]].T
    with pyro.plate("data", X.shape[0]):
        pyro.sample("obs", dist.Normal(mean, 1.0), obs=y)

def dominating(X, y):
    model1 = pyro.sample("model1", dist.Bernoulli(0.5))
    if model1:
        w = pyro.sample("w", dist.Normal(0, 1))
        fs = w * X
    else:
        sin_w = pyro.sample("sin_w", dist.Normal(0, 1))
        fs = torch.sin(sin_w * X)
    sigma = pyro.sample("sigma", dist.Gamma(1, 1))
    with pyro.plate("data", X.shape[0]):
        pyro.sample("obs", dist.Normal(fs, sigma), obs=y)
```

Figure 17: Pyro program for experiments in Sec. 6.1