ASTRA: Understanding the Practical Impact of Robustness for Probabilistic Programs (Supplementary Material)

Zixin Huang[1] **Saikat Dutta**[1] **Sasa Misailovic**[1]

[1]Department of Computer Science, University of Illinois Urbana-Champaign, Urbana, Illinois, USA

A CASE STUDIES

A.1 CASE STUDY 1: GENERALIZED LINEAR REGRESSION MODEL

A.1.1 Linear Regression

We study a simple linear regression model $y_{i=1\ldots D}|x_{i=1\ldots D} \sim \mathcal{N}(w_1x_{1i} + w_2x_{2i} + w_3x_{3i} + w_4x_{4i} + w_5x_{5i} + b, \sigma^2)$. Here y_is and x_is are the given dataset and w_j ($j \in \{1, \ldots, 5\}$), b, and σ are latent parameters. Here we apply the transformations, StudentT, Reparam, Reweight and Mixture on the original model.

To evaluate the robustness of the original and transformed models, we first generate the ground truth for the parameters w_j, b from $\mathcal{N}(0, 1)$ and x_is from *Unif*$(-1, 1)$. Then we use these parameters to generate the 500 training data with noise (Outliers). To get a test dataset, we also take 500 samples generated in the same way and add no noise.

Table 1 shows the result for evaluate the original and transformed models using different robustness metrics. We repeat the procedure 5 times and report the average of the metric values. Here we present the result for two noise levels, 0 and 10, where 0 means no noise is in data $\boldsymbol{y}$. We fit all the models using Stan's NUTS, running 4 chains each with 1000 samples. We take the mean of the samples to obtain a point estimate of all the parameters or predicted data.

In Table 1, when noise level $k = 0$, we can see all the metrics are almost at their best value, i.e., the model is accurate. Specifically, all the models have $\text{MSE}_{\text{param}} = 0.002$ (with < 0.0005 rounding error) when $k = 0$. Intuitively this means each of the parameters $\beta_j \in \{w_1, w_2, w_3, w_4, w_5,\ b,\ \sigma\}$ will give the squared error $(\hat{\beta}_j - \beta_j)^2 \approx 0.002$, and thus means $\hat{\beta}_j$, the posterior sample mean after fitting the model, has almost no difference from β_j, the true parameter value we used to generate the data. At noise level $k = 10$, some of the metrics show a less optimal value, which indicates the model is no longer accurate when there is noise presented in the training data. For example, when $k = 10$, for the original model, $\text{MSE}_{\text{param}}$ increases from 0 to 0.30, which intuitively means on average each of the parameters β_j will have the squared error $(\hat{\beta}_j - \beta_j)^2 \approx 0.257$. On contrary, for the Student-T transformed model, on average each β_j will have $(\hat{\beta}_j - \beta_j)^2 \approx 0.012$, much smaller than that from the original model.

All the robustness metrics indicate that the models after StudentT or Reparam are the most robust ones, followed by Reweight, and the Original models is the least robust one. Take the parameter w_1 which has the true value of 1.012, as an example, the Student model gives the mean value 1.010, the Reparam model gives 1.021, the Reweight model gives 0.999, and the Original model gives 0.965.

The reason that Reparam and Student are better than Reweight may be that they have one more layer of hierarchy than Reweight. All these three transformations adds one auxiliary parameter for every datapoint. Reparam and Student has one additional hyperparameter representing the degree of freedom used in the hyperprior of the auxiliary parameters, while Reweight use a flat hyperprior with no additional parameter.

If we use a Beta hyperprior with a hyperparameter for Reweight's auxiliary parameters, i.e., *weight* $\sim$ *Beta*(α, α), factor(*weight*$[i] \cdot d(E_1, \ldots)$.pdf$(y[i])$), Reweight might be able to give a better result than both Reparam and Student. However, it will drastically increase the inference time and is more likely to diverge.

Timing Interestingly, the models may run faster on data with noise than on the clean data. This is probably because the posterior shape is easier to sample from when conditioned on noisy data, but this may not hold for other models. For the parameter w_1, its posterior has an average standard deviation of $1 \cdot 10^{-2}$ with noisy data; while with clean data, the standard deviation is only $3 \cdot 10^{-4}$. Intuitively, with the same prior distribution, when the posterior from noisy data is more spread-out, a random sample generated by MCMC

Accepted for the 39th Conference on Uncertainty in Artificial Intelligence (UAI 2023).

Table 1: Evaluation Results for the Linear Regression Model (NUTS)

Transform	(#params)	k	Time (s)	$\text{MSE}_{\text{param}}$	MSE_y	$\hat{R}$	$\hat{R}_{\text{max}}$	pL1	pR2
Original	(7)	0	5.12	0.002	0.000	1.000	1.004	1.00	1.00
		10	4.11	0.257	0.026	1.000	1.003	0.97	1.00
Reparam	(508)	0	214.36	0.002	0.000	1.000	1.004	1.00	1.00
		10	93.44	0.013	0.004	1.000	1.014	0.99	1.00
Reweight	(507)	0	115.35	0.002	0.000	1.000	1.004	1.00	1.00
		10	42.42	0.087	0.010	1.000	1.003	0.98	1.00
Reweight*	(508)	0	1003.16	0.002		1.05	1.12		
		10	1014.40	0.006		1.74	3.53		
Student	(8)	0	36.02	0.002	0.000	1.000	1.003	1.00	1.00
		10	11.06	0.012	0.004	1.000	1.003	0.99	1.00

will be more likely to get accepted.

Different prior distributions also affect the timing. For example, we run the Original model on the same clean data with different priors. For the parameter σ, if we specify the limit <lower=0> and does not specify any other priors, it takes 7.5s; if we leave the limits and allow Stan to reject illegal samples, it is much faster, as 5.12 shown in the table above. This may be because the limits introduce discontinuity to the distribution and make the sampling harder. For other parameters, let w_i^* be the truth value of w_i, and if we use the prior $w_i \sim (w_i^*, 10)$, it takes 4.96s, slightly faster than default; and with $w_i \sim (w_i^*, 100)$, it takes 5.36s and is slower than default.

A.1.2 Poisson Regression

The poisson regression model is $y_i|x_i \sim$ *Poisson*$(\exp(w_1x_{1i} + w_2x_{2i} + w_3x_{3i} + w_4x_{4i} + w_5x_{5i}))$, and the noise model is $y_i|x_i \sim$ *Poisson*$(\exp(w_1x_{1i} + w_2x_{2i} + w_3x_{3i} + w_4x_{4i} + w_5x_{5i} + \epsilon_i))$ where $\epsilon_i \sim \mathcal{N}(0, k \cdot 0.15)$. Different from the other models, the poisson regression seems harder to converge. In the table 2, we present the result for running NUTS with 4 chains for with 10000 iterations. If we use 1000 iterations, most of the transformed models will not converge.

From the results, we can see the Reweight model works best: it is able to give a small MSE for parameters even when the data contains noise. Specifically, in one run, the true value of w_1 is 1.31. With clean data, both the original model and the Reweight transformed model are able to give the correct results. With noise level $k = 10$, the Reweight model gives 1.28; while the original model gives 1.21.

A.1.3 Logistic Regression

For the logistics model, we sort the data based on their probability and then start flipping the label from the ones with the lowest scores, as in [Wang et al., 2018]. We can see the Reweight model gives better results than the Original one, but the difference is not large, because the model of flipping binary labels has little effect on both models.

A.2 CASE STUDY 2: MIXTURE MODEL

A.2.1 gauss_mix_given_theta

The noise model for the mixture model by adding one more noise group with mean $\max(\mu_1, \mu_2) + |\mu_1 - \mu_2|$, meaning that a new group is to the right of the two groups and forms three groups with equal intervals. The new group has probability $p_3 = 0.02k$, and the original two groups have probability $p_1 = (1-p_3)\cdot\theta$ and $p_2 = (1-p_3)\cdot(1-\theta) = 1-p_3-p_1$. In the results, Reparam and Student ranks first, and then Original, which is similar to the linear regression model. However, the Reweight model does not work: it misclassifies two group means to places between the noise group and the group to the right. The other transformations, Reparam and Student are able to filter out the noise group and identify the true groups.

A.3 CASE STUDY 3: TIME SERIES MODELS

A.3.1 koyck

The koyck model is $y_t \sim \mathcal{N}(w_1 + w_2 * x_t + w_3 * y_{t-1}, \sigma)$. We use the Outliers noise model. Notice the form of this model is similar to the linear regression models. The difference is the additional dependency on previous timestamp. The results are also similar: Student and Reparam are the best, then Reweight. All the three transformations are more robust than Original.

Table 2: Evaluation Results for the Poisson Regression Model (NUTS)

Transform	(#params)	k	Time (s)	$\text{MSE}_{\text{param}}$	MSE_y	$\hat{R}$	$\hat{R}_{\text{max}}$	pL1	pR2
Original	(5)	0	17.052	0.000	$1.65 \cdot 10^5$	1.000	1.001	0.997	1.000
		10	17.248	0.057	$6.58 \cdot 10^{10}$	1.000	1.000	-0.487	-3.307
Reweight	(505)	0	146.864	0.000	$2.48 \cdot 10^4$	1.000	1.000	0.998	1.000
		10	136.662	0.006	$4.65 \cdot 10^9$	1.000	1.011	0.561	0.695

A.3.2 gp-fit-latent

The gp-fit-latent model is

$$\begin{aligned}
\rho &\sim \textit{InvGamma}(5,5) \\
\alpha &\sim \mathcal{N}(0,1) \\
\sigma &\sim \mathcal{N}(0,1) \\
f &\sim \textit{MultivariateNormal}(0, K(x|\alpha,\rho)) \\
y_i &\sim \mathcal{N}(f_i,\sigma) \;\; \forall i \in \{1,\ldots,N\}
\end{aligned}$$

Here K is a exponentiated quadratic kernel. Under the Outliers noise model, Reparam and Student are the best, then Reweight.

B AUTOMATED TRANSFORMATIONS

B.1 STORM-IR

We implement our transformations on an intermediate representation, called Storm-IR, for probabilistic programs [Dutta et al., 2019]. Figure 3 presents the syntax of Storm-IR. Storm-IR is an imperative language with support for standard constructs like arithmetic operations, loops and conditionals, and probabilistic constructs like sampling from distributions (*Dist*) and conditioning on data (*observe*). The Storm framework also provides translators from Storm-IR to other probabilistic programming languages like Stan [Carpenter et al., 2016] and Pyro [Bingham et al., 2018] and vice-versa. Using Storm-IR allows our transformations to be language-agnostic and also leverage a host of different program analyses (e.g. dimensional, interval, and data-flow analysis) which help us implement our transformations easily.

x	$\in$	*Vars*
c	$\in$	*Consts* $\cup \{-\infty, \infty\}$
op	$\in$	$\{+, -, >, ...\}$
Dist	$\in$	{Normal, Uniform, ...}
Type	::=	`Int` | `Float`
Decl	::=	x : Type | x : [c^+]
Expr	::=	c | x | *Dist*.pdf(Expr) | Expr *op* Expr
Stmt	::=	x = Expr
		| for $x \in$ 1..N; { Stmt* }
		| `observe`(*Dist*(Expr*), x)
		| `factor`(Expr)
		| `if` (Expr) `then` Stmt* `else` Stmt*
		| x := *Dist*(Expr*)
		| Decl
Program	::=	Stmt*

Table 3: Syntax of Intermediate Representation

B.2 AUTOMATICALLY TRANSFORMING PROGRAMS

We implement our transformations on an intermediate representation (IR) for probabilistic programs: Storm-IR [Dutta et al., 2019]. Storm-IR represents standard and probabilistic language constructs like sampling from distributions (Dist) and conditioning on data (factor) as a graph with program elements as nodes, and control flow as edges (similar to standard compiler CFG [Allen, 1970]). Since Storm-IR supports multiple languages (e.g., Stan, Pyro, Edward), it allows ASTRA to be language-agnostic and also provides a host of different program analyses (e.g., dimensional, interval, and data-flow analysis) that help us implement transformations easily and correctly.

ASTRA first parses the original probabilistic program into abstract syntax tree and converts to Storm-IR. On this IR, searching for the code pattern from Table 1 amounts to searching for a subgraph that encodes the pattern (e.g., statements corresponding to $\beta \sim \pi_\beta(\alpha)$ and $y_{i=1}^D \sim F(\beta)$; which do not need be adjacent), while remembering the concrete variable names (e.g., $\beta \mapsto \texttt{b}$, $y \mapsto \texttt{data}$) and distributions (e.g., $F \mapsto \mathcal{N}(\texttt{b}, \texttt{s})$). After identifying the pattern, ASTRA checks for the transformation legality and uses the identified distributions/variables to instantiate the transformation template and uses Storm-IR API to update the program graph. ASTRA allows users to implement new transformations on Storm-IR, which is analogous to writing a compiler transformation pass. ASTRA implementation allows applying transformations iteratively on the same program, however, we observed that the combined transformations do not provide additional robustness benefits, while their inference quality suffers from the complicated model.

Here we present the code patterns of the original and transformed programs for each transformation:

Bayesian Data Reweighting. Figure 1 presents the code pattern demonstrating this transformation. The transformation is applicable on any model with the `factor` statement. During the transformation, the prior distributions of the parameters (x_1) in the model remain unchanged. We introduce the new parameter w (vector), and multiply each $w[i]$ to the log-probability expression of $y[i]$ in factor.

Localization. Figure 2 presents the code pattern for this transformation. This transformation is applicable whenever there is a `factor` statement in a for-loop. First, we introduce the parameter η (vector) as the localized for of x_1. Then we update the factor expression to relate each data point $y[i]$ with an individual realization of the parameter $\eta[i]$. We also initialize the parameter with prior distributions.

Normal to Student-T. Figure 3 presents the code pattern for this transformation. We change an old Normal distribution with a Student-T distribution. This transformation is applicable for normal distributions in `factor` statement.

Reparameterization and Localization of the Scale Parameter. We present the code pattern for this transformation in Figure 4. This transformation is only applicable when there are normal distributions in the `factor` statement. We introduce a new parameter τ (vector), where τ follows a *Gamma* distribution with a newly added hyper-parameter ν. We update the factor expression by dividing the standard deviation x_2 by the inverse square-root of τ.

Contaminated Group Mixture. We present the code pattern for this transformation in Figure 5. The transformation is only applicable when the distribution in the `factor` statement has the location and scale parameters. We introduce a new factor statement that samples from a *LogNormal* distribution for outliers. The model is changed to either sample from the original distribution or the outlier distribution, encoded as an if-then-else statement.

$x_1 :=$ *DistExpr*$_1$	prior
...	
for (i = 1..D)	
factor(*DistExpr*$_2(x_1)$.pdf($y[i]$))	conditioning
return x_1	posterior
⇓	
$x_1 :=$ *DistExpr*$_1$	prior (unchanged)
var w[D]	init. weights.
for (i = 1..D)	
$w[i]$:= *Beta*(γ, η)	reweighting dist.
...	
for (i = 1..D)	
factor(*DistExpr*$_2(x_1)$.pdf($y[i]$) * **$w[i]$**)	reweighted obs.
return x_1, **w**	posteriors

Figure 1: Reweighting Transformation Code Pattern

$x_1 :=$ *DistExpr*$_1$	prior
...	
for (i = 1..D)	
factor(*DistExpr*$_2$(...,x_1,...).pdf($y[i]$))	conditioning on
return x_1	obs. y posterior
⇓	
$x_1 :=$ *DistExpr*$_1$	prior (unchanged)
s := *Unif*(0, 1)	new hyper-prior
var η[D]	localized params.
for (i = 1..D)	
$\eta[i]$:= *Normal*(x_1, s)	new priors
...	
for (i = 1..D)	
factor(*DistExpr*$_2$(...,**$\eta[i]$**,...).pdf($y[i]$))	localized obs.
return x_1, η	posteriors

Figure 2: Localization Transformation Code Pattern

$x_1 :=$ *DistExpr*$_1$	prior mean
$x_2 :=$ *DistExpr*$_2$	prior std
...	
for (i = 1..D)	
factor(*Normal*(x_1, x_2).pdf($y[i]$))	conditioning
return x_1	posterior
⇓	
$x_1 :=$ *DistExpr*$_1$	prior mean
$x_2 :=$ *DistExpr*$_2$	prior std
ν := *Unif*(...)	new hyper-prior
...	
for (i = 1..D)	
factor(***StudentT***(ν, x_1, x_2).pdf($y[i]$))	Student-T dist.
return x_1, ν	posteriors

Figure 3: Normal/Student-T Transformation Code Pattern

$x_1 :=$ *DistExpr*$_1$	prior mean
$x_2 :=$ *DistExpr*$_2$	prior std
...	
for (i = 1..D)	
factor(*Normal*(x_1, x_2).pdf($y[i]$))	conditioning
return x_1, x_2	Gauss. posterior
⇓	
$x_1 :=$ *DistExpr*$_1$	prior mean
$x_2 :=$ *DistExpr*$_2$	prior std
ν := *DistExpr*$_3$	new hyper-prior
for (i = 1..D)	
$\tau[i]$:= *Gamma*(ν/2, ν/2)	robustness factors
...	
for (i = 1..D)	
factor(*Normal*(x_1, x_2**/sqrt($\tau[i]$)**).pdf($y[i]$))	conditioning
return x_1, x_2	Gauss. posterior

Figure 4: Reparameterization Transformation Code Pattern

C CORRECTNESS OF THE TRANSFORMATIONS

We formally state that the transformations we define in Section B.2 have the semantic effects as proposed in the statistical literature (as summarized in Table 1). We leverage Stan's operational semantics from [Gorinova et al., 2019].

Given a program P in StormIR language, the StormIR translator will translate P into a Stan program S with equivalent semantics. There exists an one-to-one correspondence between StormIR expressions/statements and Stan expression/statements, by the definition of StormIR syntax (Appendix A.1) and Stan syntax [Gorinova et al., 2019]. For example, let $\Leftrightarrow$ denote the translation relation between a

Table 4: MSE Improvement for Each Program at Noise Level 10 with ADVI

Prog	Outliers	Hidden Group	Skewed
RE	256.42 (StudentT)	1.62 (Reparam)	1.00 (Original)
RV	28.04 (StudentT)	2.78 (Reparam)	1.00 (Original)
MC	27.48 (Local1)	- -	1.00 (Original)
SE	14.23 (StudentT)	3.10 (StudentT)	1.03 (Mixture)
RK	8.41 (StudentT)	3.69 (StudentT)	1.00 (Original)
RN	7.11 (Reparam)	2.06 (Local2)	1.00 (Original)
RU	3.42 (StudentT)	2.05 (StudentT)	1.41 (Local2)
RA	3.31 (StudentT)	2.08 (StudentT)	1.00 (Original)
MF	3.27 (StudentT)	- -	1.23 (Reparam)
RQ	3.23 (StudentT)	2.32 (StudentT)	1.26 (Local1)
RR	2.95 (Reparam)	2.02 (StudentT)	1.14 (Local1)
RX	2.93 (StudentT)	1.94 (StudentT)	1.14 (Local1)
SD	2.52 (StudentT)	12.31 (StudentT)	1.00 (Local1)
MD	2.21 (Reweight)	- -	1.16 (Local1)
ME	1.27 (StudentT)	- -	1.13 (Local1)
RY	1.25 (StudentT)	1.00 (Original)	1.00 (Original)
MB	1.14 (StudentT)	- -	1.77 (Local1)
RG	1.04 (StudentT)	- -	- -
SA	1.02 (Mixture)	2.04 (StudentT)	1.03 (Local1)
RW	1.00 (Reweight)	- -	- -
SB	1.00 (StudentT)	1.00 (Local1)	1.00 (Local1)
SC	1.00 (Original)	1.42 (Local1)	1.00 (Original)
RL	1.00 (Original)	1.00 (Original)	4.51 (Local2)
MA	1.00 (Original)	- -	2.19 (Local1)

Table 5: MSE Improvement for Each Program at Noise Level 10 with NUTS

Prog	Outliers	Hidden Group	Skewed
RE	412.60 (StudentT)	4.59 (Reparam)	1.00 (Local1)
RV	31.94 (Reparam)	2.73 (Reparam)	1.00 (Original)
MC	1.00 (Original)	- -	1.01 (Reparam)
SE	16.02 (Reweight)	3.15 (Reparam)	1.00 (Original)
RK	9.25 (Reparam)	5.65 (StudentT)	1.00 (Original)
RN	6.25 (Local2)	5.54 (Reparam)	1.00 (Original)
RU	3.75 (StudentT)	2.26 (StudentT)	1.00 (Local1)
RA	3.19 (Reparam)	2.14 (StudentT)	1.00 (Original)
MF	2.81 (Reparam)	- -	1.05 (Local1)
RQ	3.78 (StudentT)	2.34 (StudentT)	1.00 (Original)
RR	3.00 (Reparam)	1.94 (StudentT)	1.00 (Local1)
RX	3.18 (Reparam)	2.07 (StudentT)	1.00 (Local1)
SD	3.52 (StudentT)	11.23 (Reparam)	1.00 (Local1)
MD	6.08 (Reweight)	- -	2.10 (Reweight)
ME	1.41 (Reparam)	- -	1.00 (Original)
RY	1.00 (Original)	1.05 (Reparam)	1.67 (Local1)
MB	1.22 (StudentT)	- -	2.15 (Local1)
RG	1.03 (Reweight)	- -	- -
SA	1.56 (Reparam)	2.42 (StudentT)	1.04 (Local1)
RW	1.00 (Reweight)	- -	- -
SB	1.00 (Original)	1.00 (StudentT)	1.00 (Reweight)
SC	1.05 (Local1)	1.36 (Reweight)	1.00 (Original)
RL	1.00 (Original)	1.01 (Local1)	1.01 (Local1)
MA	1.68 (StudentT)	- -	1.94 (Local1)

StormIR expression/statement and a Stan expression/statement, then factor translation rule is:

$$\frac{E_{storm} \Leftrightarrow E'_{stan}}{\texttt{factor}(E_{storm}) \Leftrightarrow \mathbf{target} = \mathbf{target} + \log(E'_{stan})}.$$

It states that StormIR's factor statement is translated to an assignment to a special variable **target** in Stan (it by convention contains unnormalized log-posterior), where the expression E_{storm} was recursively translated to E'_{stan}. Rules for other statements are similar.

Definition 1. *We denote as P any StormIR program on which ASTRA can apply a transformation T to get a transformed program P_T according to the Transformation Code Pattern shown in Section B.2.*

Definition 2. *We denote as $p(\boldsymbol{\theta}|y)$ and $p_T(\boldsymbol{\theta}|y)$ the posteriors from the original and the transformed program using the transformation T defined in Table 1 where $\boldsymbol{\theta}$ represents all the parameters in the program and y is the data.*

x_1 := *DistExpr*$_1$	prior mean
x_2 := *DistExpr*$_2$	prior std
...	
for (i = 1..D)	
factor (*Dist*(μ, σ, ...).pdf (*Expr*$_2$))	conditioning
return x_1, x_2	Gauss. posterior
⇓	
x_1 := *DistExpr*$_1$	prior mean
x_2 := *DistExpr*$_2$	prior std
ρ_{out} := *Unif*(0, 0.5)	outlier probability.
μ_{out} := *DistExpr*$_3$	outlier mean
s_{out} := *DistExpr*$_4$	outlier variance
out := *LogNormal*(μ_{out}, s_{out})	outlier probability.
...	
if(*Bernolli*($1 - \rho_{out}$))	mixture
for (i = 1..D)	
factor(*Dist*(x_1, x_2).pdf($y[i]$))	conditioning
else	Gauss. mixture
for (i = 1..D)	
factor(*Dist*(x_1, sqrt(exp(*out*)).pdf($y[i]$))	conditioning
return x_1, x_2	Gauss. posterior

Figure 5: Cont. Mixture Transformation Code Pattern

Theorem 1. *If the distribution of the program P is equivalent (up to a unique normalizing constant) to $p(\boldsymbol{\theta}|y)$ then the distribution and P_T is equivalent (up to a unique normalizing constant) to $p_T(\boldsymbol{\theta}|y)$.*

We sketch the proof next. We first translate the programs P and P_T to equivalent Stan programs S and S_T, respectively, as discussed above. By Stan's operational semantics presented in Gorinova et al. [2019], we know that there exists a unique end state s for S as $((y, \boldsymbol{\theta}, \mathbf{target} \mapsto 0), S) \Downarrow s$ where $s[\mathbf{target}] = \log p^*(\boldsymbol{\theta}|y)$. $p^*(\boldsymbol{\theta}|y)$ is the unnormalized posterior which uniquely defines the posterior as $p(\boldsymbol{\theta}|y) \propto p^*(\boldsymbol{\theta}|y)$. Similarly, S_T results in the unique end state s_T which has $s_T[\mathbf{target}] = \log p^*_T(\boldsymbol{\theta}|y)$, and $p_T(\boldsymbol{\theta}|y) \propto p^*_T(\boldsymbol{\theta}|y)$. Since P and S are equivalent, and P_T and S_T are equivalent, we can next apply structural induction on the Stan statements that are defined in each rule from Figures 1, 2, 3, 4, and 5 to derive the posterior distributions of each original and transformed program, as $p^*(\boldsymbol{\theta}|y)$. and $p^*_T(\boldsymbol{\theta}|y)$, respectively. For each, we can immediately verify that there is an equivalence relation between $p^*(\boldsymbol{\theta}|y)$ and $p(\boldsymbol{\theta}|y)$ defined in Table 1, and between $p^*_T(\boldsymbol{\theta}|y)$ and $p_T(\boldsymbol{\theta}|y)$.

D BEST MSE IMPROVEMENT FOR DIFFERENT NOISE MODELS

Tables 4,5 present the best MSE improvements for ADVI and NUTS across different noise models and programs. The cells with "–" mean that the noise model is not applicable to the data in the program.

E CONVERGENCE SCORES AT NOISE LEVELS 2 AND 6

Tables 6 and 7 present the convergence scores at noise levels 2 and 6. We observed a similar overall trend in convergence scores across different noise levels.

Table 6: (Geometric-)Mean of Rhat at Noise Level 2

Transformations	**Outliers**		**Hidden Group**		**Skewed Data**	
	ADVI	NUTS	ADVI	NUTS	ADVI	NUTS
Original	1.75	1.05	1.16	1.00	2.43	1.08
Reweighting	1.33	1.11	1.19	1.01	1.40	1.03
Localized-Loc	3.40	1.38	2.18	1.13	4.15	1.21
Localized-Scale	4.24	1.43	1.85	1.03	4.47	1.05
Reparam-Local	2.02	1.25	1.25	1.02	2.36	1.15
StudentT	1.66	1.41	1.22	1.00	1.72	1.34
Cont. Group Mixture	7.17	–	8.77	–	8.43	–

Table 7: (Geometric-)Mean of Rhat at Noise Level 6

Transformations	**Outliers**		**Hidden Group**		**Skewed Data**	
	ADVI	NUTS	ADVI	NUTS	ADVI	NUTS
Original	1.79	1.46	1.32	1.00	1.65	1.04
Reweighting	1.34	1.19	1.17	1.00	1.30	1.01
Localized-Loc	3.86	1.34	2.83	1.16	3.77	1.25
Localized-Scale	3.04	1.38	2.05	1.04	3.75	1.10
Reparam-Local	2.01	1.34	1.32	1.00	2.35	1.18
StudentT	1.56	1.26	1.19	1.01	1.97	1.37
Cont. Group Mixture	8.99	–	8.48	–	7.86	–

F OTHER DIAGNOSTICS FOR NUTS AT NOISE LEVEL 10

Here we present two other diagnostics for NUTS, the effective sample size (ESS) and the trajectory divergence.

Table 8 presents the ESS at noise level 10 for every 4x1000 samples after warmup, under a timeout of 8 minutes for each chain. A small ESS also indicates the lack of convergence.

Table 8: (Geometric-)Mean of ESS at Noise Level 10

Transformations	**Outliers**	**Hidden Group**	**Skewed Data**
Original	2482.04	2526.45	2144.90
Reweighting	2422.91	2289.96	2397.99
Localized-Loc	876.01	1067.06	1179.52
Localized-Scale	1114.61	1467.49	842.73
Reparam-Local	1707.49	2296.89	2029.92
StudentT	1338.56	2673.49	1786.98
Cont. Group Mixture	-	-	-

For NUTS, the geometric means of the *trajectory divergence* over all the applicable models for each transformation at each noise level is smaller than 0.01, with 90% of the models have trajectory divergence being 0. Such a small trajectory divergence portion does not indicate any issue of concerns.

References

Frances E Allen. Control flow analysis. *ACM Sigplan Notices*, 5(7), 1970.

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*, 2018.

Bob Carpenter, Andrew Gelman, Matt Hoffman, Daniel Lee, Ben Goodrich, Michael Betancourt, Michael A Brubaker, Jiqiang Guo, Peter Li, Allen Riddell, et al. Stan: A probabilistic programming language. *JSTATSOFT*, 20(2), 2016.

Saikat Dutta, Wenxian Zhang, Zixin Huang, and Sasa Misailovic. Storm: program reduction for testing and debugging probabilistic programming systems. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 729–739. ACM, 2019.

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

Chong Wang, David M Blei, et al. A general method for robust bayesian modeling. *Bayesian Analysis*, 13(4):1159–1187, 2018.