© 2024 Zixin Huang

ENHANCING TRUSTWORTHINESS IN PROBABILISTIC PROGRAMMING: SYSTEMATIC APPROACHES FOR ROBUST AND ACCURATE INFERENCE

BY

ZIXIN HUANG

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois Urbana-Champaign, 2024

Urbana, Illinois

Doctoral Committee:

Associate Professor Sasa Misailovic, Chair
Professor Darko Marinov
Professor Sayan Mitra
Assistant Professor Eunsuk Kang, Carnegie Mellon University

ABSTRACT

Probabilistic programming simplifies the encoding of statistical models as straightforward programs. At its core, it employs an inference algorithm which automate the model inference, allowing developers to focus on model creation. Its simplicity has led to its growing application in critical areas such as autonomous driving, privacy modeling, computer networks, and pandemic prediction. However, the flexibility of probabilistic modeling and the scalability to large datasets come at a price of *trustworthiness*: the approximate inference algorithms used by many existing probabilistic programming systems may produce inaccurate results; also the collected data often contain noise, which can violate the model's assumption and cause large deviations in the results. Trustworthiness, therefore, has two key properties: *accuracy*, to ensure results close to the true underlying distribution, and *robustness*, to ensure reliable results amidst data noise.

This dissertation introduces a systematic approach, composed of multiple probabilistic programming systems throughout the probabilistic programming computation stack, to analyze and enhance the trustworthiness of probabilistic programs. We present the results of two-pronged investigation: first, we identify several trustworthiness challenges of the current practice of probabilistic programming algorithms. The examination is supported by the presentation of ASTRA, an experimental testbed for evaluating the robustness of probabilistic programs against data noise, and SixthSense, a system aiding developers in debugging convergence issues in sampling-based approximate inference algorithms.

The second part of the dissertation introduces AQUA, a novel quantized inference algorithm which can achieve better accuracy than existing approximate inference algorithms and scales better than exact inference. Then, the dissertation advances the domain of probabilistic reasoning by moving beyond the conventional focus on computing a single posterior distribution. It presents AURA, an abstract interpretation which provides precise, soundly guaranteed bounds on posterior distributions when they are subjected an infinite set of data perturbations.

To my family, with love.

ACKNOWLEDGMENTS

Getting my PhD has completely changed me, both personally and professionally. Looking back on the beginning of this journey, it feels almost surreal to be concluding this chapter of my life. There were many challenges and moments of doubt, yet here I am, about to achieve my dream, thanks to the help of many of people.

My deepest gratitude goes to my advisor, Sasa Misailovic. We first met in the spring of 2017, when I was still an undergrad at UIUC. Sasa introduced me to the world of probabilistic programming, a field that has captivated me for years. Over the course of seven years, we experienced both highs and lows, and we have witnessed how the probabilistic programming field grew and evolved. We explored new frontiers, unexpected and unforeseen, where the discoveries seemed almost magical at the time. Looking back, I see that Sasa was always the source of encouragement and inspiration, allowing me to push beyond boundaries. Sasa not only shaped my academic journey but also instilled in me the virtues of positivity, resilience, creativity, and pragmatism. These qualities have extended beyond our research and have become core principles in my life.

I also need to thank my PhD committee, Professors Darko Marinov, Sayan Mitra, and Eunsuk Kang. Each one has played an important role in my journey, offering me their feedback and support. Professor Darko Marinov has known me since my undergrad days and all through my PhD. He kindly allowed me to use his space in summer 2017 to work on my first paper, making a great first impression on me about conducting research. Moreover, I must thank Professor Carl Gunter, who guided me to my advisor, Sasa, during my undergrad years as I was searching for research opportunities, marking the beginning of my story.

I would like to thank my labmates: Vimuth Fernando, Saikat Dutta, Keyur Joshi, Jacob Laurel, Yifan Zhao, and Shubham Ugare. I miss the times we spent discussing solutions and inspiring each other. Saikat has been a co-author on many of my papers. He possesses an exceptional research vision and has taught me many useful skills, including scripting and academic writing. Collaborating with Jacob was always enlightening; Jacob's wealth of new ideas, combined with his ability for thorough reasoning, allowed us to solidify ideas and finally bring them to life. Vimuth was always caring and considerate, often brought snacks; while Keyur, my desk neighbor, always shared fun life stories and Altolds. Exploring various great restaurants with Yifan and Shubham was also a memorable experience outside the lab.

I also want to extend my thanks to many student collaborators and longtime friends who were part of my PhD journey. I want to give special thanks to Zitong Zhou, who

collaborated with me on a paper and offered brilliant insights and invaluable feedback on my project. I would also like to thank Enguang Fan, the first undergrad collaborator I had the honor of working with, for his outstanding knowledge in statistics and commitment. Furthermore, I would also like to thank everyone in the Computer Science Department for their contributions in making various aspects of my PhD study enjoyable and seamless.

During my PhD, I did three internships which taught me valuable lessons, each marking an important step in my PhD journey. At Apple, I worked with Alex Horn, where I gained knowledge about distributed computing and cultivated a mindset for establishing rigorous methodologies. At Meta, under the mentorship of Cornelius Aschermann, I gained insights into coverage-guided fuzz testing, and also improved my coding and communication skills. At Nvidia, working with Chris Bate and Josh Park, I discovered my passion in the area of deep learning compilers. I will carry these experiences forward into the next phase of my career.

I am also thankful to various funding sources for supporting my PhD research, including the NSF grants and the Andrew and Shana Laursen Fellowship.

Above all, I am grateful to my parents and my entire family; they have always believed in me and been proud of me. My parents have attentively followed every step and achievement I've made, shaping me into the go-getter I am today. Finally, I would also like to thank Dr. Ziwei Ji for his invaluable support.

TABLE OF CONTENTS

CHAPTER 1 INTRODUCTION 1
- 1.1 Trustworthy Probabilistic Programming 2
- 1.2 Dissertation Goals 5
- 1.3 Contributions 6

CHAPTER 2 ASTRA: UNDERSTANDING THE PRACTICAL IMPACT OF ROBUSTNESS FOR PROBABILISTIC PROGRAMS 9
- 2.1 Introduction 9
- 2.2 Robustness Metrics 11
- 2.3 Robustness Transformations 13
- 2.4 ASTRA 14
- 2.5 Correctness of the Transformations 19
- 2.6 Methodology 20
- 2.7 Evaluation 22
- 2.8 Other Diagnostics for NUTS at Noise Level 10 27
- 2.9 Discussion 28
- 2.10 Related Work 29
- 2.11 Conclusion 29

CHAPTER 3 DEBUGGING CONVERGENCE PROBLEMS IN PROBABILISTIC PROGRAMS VIA PROGRAM REPRESENTATION LEARNING WITH SIXTHSENSE 30
- 3.1 Introduction 30
- 3.2 Example 33
- 3.3 Overview 36
- 3.4 Learning Program Features 39
- 3.5 Program Generation for Training Set Augmentation 43
- 3.6 Methodology 47
- 3.7 Evaluation 50
- 3.8 Sensitivity Analysis 59
- 3.9 Related Work 61
- 3.10 Conclusion 62

CHAPTER 4 AUTOMATED QUANTIZED INFERENCE FOR PROBABILISTIC PROGRAMS WITH AQUA 64
- 4.1 Introduction 64
- 4.2 Preliminaries 66
- 4.3 Inference with Density Cubes 67

4.4 AQUA Optimizations . . . 80
4.5 Methodology . . . 82
4.6 Evaluation . . . 84
4.7 Related Work . . . 87
4.8 Conclusion . . . 89

CHAPTER 5 PRECISE ABSTRACT INTERPRETATION OF PROBABILISTIC PROGRAMS WITH INTERVAL DATA UNCERTAINTY . . . 90
5.1 Introduction . . . 90
5.2 Example . . . 94
5.3 Preliminaries . . . 98
5.4 Concrete Semantics . . . 101
5.5 Abstract Semantics for Data Perturbation . . . 103
5.6 AURA Optimization and Verification Algorithm . . . 105
5.7 Soundness Proofs . . . 114
5.8 Implementation and Optimizations . . . 116
5.9 Methodology . . . 118
5.10 Evaluation . . . 120
5.11 Related Work . . . 125
5.12 Conclusion . . . 127

CHAPTER 6 CONCLUSIONS AND FUTURE WORK . . . 128
6.1 Conclusions . . . 128
6.2 Future Work . . . 128

REFERENCES . . . 131

APPENDIX A ASTRA ADDITIONAL RESULTS . . . 146
A.1 Best MSE Improvement for Different Noise Models . . . 146
A.2 Convergence Scores at Noise Levels 2 and 6 . . . 146

APPENDIX B AURA PROOF AND EXPERIMENT DETAILS . . . 148
B.1 Pseudo-Concavity of Benchmarks . . . 148
B.2 Experimental Setup Details . . . 155
B.3 Query under Data Perturbation for All Feasible Benchmarks . . . 156
B.4 Illustration of Unsound Results from GuBPI . . . 158

CHAPTER 1: INTRODUCTION

Probabilistic programming techniques has greatly simplified Bayesian modeling by separating the process of model development from that of automated inference. These techniques enhance conventional programming languages, introducing constructs for sampling from probability distributions and probabilistic conditioning [1], thereby bringing randomness and Bayesian inference as first-class abstractions.

Benefiting from its interpretability and efficiency, probabilistic programming has become increasingly popular in critical areas, like testing of autonomous vehicles [2], pandemic prediction [3], and security or privacy modeling [4, 5]. This approach allows engineers and scientists, even those with limited knowledge of the underlying inference mechanisms, to express a statistical model as a probabilistic program. They can then query the probabilistic programming systems to assess the probability of specific events, facilitating more informed predictions and decision making processes.

Probabilistic programming systems simplify the complex process of probabilistic inference by hiding the intricate details. However, probabilistic inference is fundamentally hard, and thus most approaches from statistics and machine learning communities rely on approximation techniques. These include sampling-based methods like Markov Chain Monte Carlo (MCMC) [6, 7, 8, 9, 10], alongside optimization-based approaches like variational inference (VI) [11, 12]. However, probabilistic programming systems are not without limitations, particularly in safety-critical applications where reliability is paramount.

Figure 1.1 illustrates the conventional workflow in probabilistic programming, which involves four stages: (1) Initially, the developer starts with a dataset, assuming it follows a normal distribution defined by certain parameters. While the procedure is generic, the figure showcases an example where the dataset, depicted as blue points and denoted as 'Y',

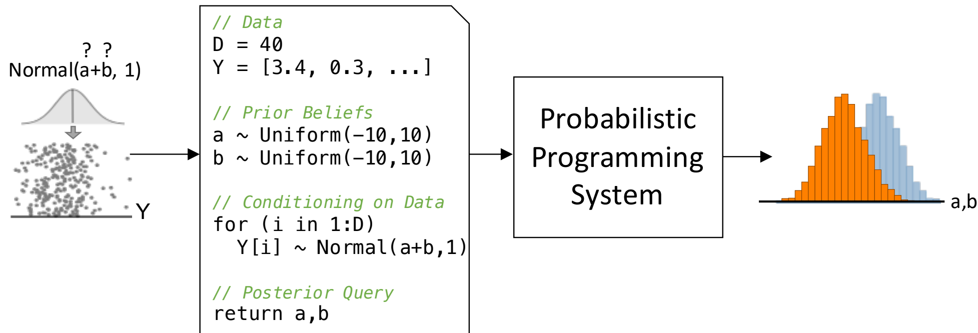


Figure 1.1: Probabilistic Programming Workflow.

follows a normal distribution with parameters ‘a’ and ‘b’. (2) Next, the developer encodes this data and the assumed model into a probabilistic program, which finally returns the distribution of these parameters (‘a’ and ‘b’ in this example). (3) In the third phase, the developer utilizes an existing probabilistic programming system to solve the posterior of the distributions, which at a low level employs inference algorithms like MCMC. (4) In the final stage, the developer queries for posterior distributions of the model’s parameters. While the figure uses blue and orange histograms to depict the posterior distributions of ‘a’ and ‘b’ (a common approach in sampling-based inference algorithms like MCMC), other inference algorithms can represent the posteriors differently.

1.1 TRUSTWORTHY PROBABILISTIC PROGRAMMING

Given the increasing popularity of probabilistic programming in critical areas such as autonomous vehicles and security or privacy modelling, the *trustworthiness* of probabilistic programs is crucial, as their result posterior distributions are often used to make informed decisions, estimations or predictions. Two key properties of trustworthiness in probabilistic programming include *accuracy* and *robustness*. Accuracy refers to the model’s ability to make inferences that closely match the true underlying distributions of the data. Robustness refers to the model’s ability to give reliable results regardless of anomalies in data (e.g. the presence of outliers).

However, each stage in probabilistic programming workflow (as illustrated in Figure 1.1) has potential pitfalls that can compromise trustworthiness of probabilistic programming:

- **Noisy Data** In practical scenarios, data often comes with unexpected noise that may not align with the model’s expectations, challenging the model’s robustness to accurately fit despite data noise. For instance, the motivating example in Chapter 2 shows how a small perturbation in data can lead to a large deviation in the output posterior.

- **Unsuitable Model** During the development of probabilistic programs, developers may construct a model that is not suitable for a chosen inference algorithm. Some model structures like unlikely distributions or intricate dependencies between parameters may cause the non-convergence problem of MCMC or variational inference algorithms, resulting in inaccurate posterior estimation.

- **Approximate Inference** From a more theoretical perspective, the accuracy of approximate inference algorithms, such as MCMC sampling, are only guaranteed to be

accurate when in theory infinite samples are obtained. This condition defines the concept of *convergence*, which is the state where the algorithm's output stabilizes close to the true underlying distribution. In practice, given limited computational resources and time budget, the precision of estimated posteriors from these approximate inference techniques may not meet the requirements in safety critical applications. This limitation is illustrated in Chapter 4, where the analysis highlights inadequacies in the tails of posterior distributions produced by MCMC.

- **Intersecting of above Issues** All the issues mentioned above may intersect. For example, many robustness-improving techniques would make the model more complicated for a chosen inference algorithm to handle, thus resulting in non-convergence issues and giving inaccurate results, observed in the experiments described in Chapter 3.

Related works, while innovative in diverse applications, largely overlook these critical trustworthiness issues of accuracy and robustness in probabilistic programming. Many languages/systems adopt a best-effort approach, leaving key decisions regarding execution time and trustworthiness properties to the developers. Programmers therefore get little guidance on how to enhance the accuracy and robustness of probabilistic programs, and current probabilistic programming systems demand that developers possess a high level of specialization in the intricate details of machine learning, statistics, and programming language semantics.

A main theme of related works is the use of probabilistic programming as a specialized tool for modeling and verifying properties within specific application domain, such as fairness and computer networks. However, these studies typically do not address the trustworthiness issues (namely accuracy and robustness) within the workflow of probabilistic programming itself. Instead, they focus on designing customized programming languages and inference algorithms tailored for the specific applications. Illustrations of such an approach are seen in the works of [13] and [14] which leverage volume computation or concentration inequality, respectively, to verify fairness properties; [15], [16] and [17] focus on network verification, and [18] address security policies. While these methods are able to compute probability of an event and verify a property within their targeted application, many of them fall short in computing entire posterior distributions, and lack the capability to provide guarantees on these distributions. They do not address trustworthiness issues such as the effects of data perturbation on posterior robustness, a concern we have identified above. Moreover, given that these studies employ probabilistic programming as a tool in their analyses, they also inherently face trustworthiness challenges such as inaccurate results or vulnerability to data noise, which could compromise the integrity of their analyses.

Additionally, there has been a growing interest in symbolic inference which guarantees the accuracy of probabilistic program results. For instance, Hakaru [19] and PSI [20] utilizes pure symbolic inference to derive the posterior by calculating its mathematical formula. However, both tools struggle with translating complex real-world programs into closed-form solutions due to the often necessary integration processes, since the computation of the integrals become intractable for larger models. Other studies, such as that involving DICE, focuses only on models with discrete distributions [21], while SPPL limits its application to models that does not contain data observation from continuous distributions [22].

While achieving perfect accuracy using exact, symbolic inference is challenging, another line of study considers abstract interpretation for probabilistic programs [23, 24]. Abstract interpretation for probabilistic programs is more challenging compared to that for deterministic programs due to the inherent uncertainties and the variety of possible events. Many approaches aim to accurately over-approximate the probabilistic programs states, especially their probability distributions, using abstract transformers. An over-approximation of a probabilistic program's distributions includes all the actual distributions, which ensures that if the over-approximated program satisfies certain properties, then the actual program does as well. To analyze over-approximated models, researchers often resort to volume computation, which quantifies the "volume" represented by the probability distributions occupied by the probability distributions within the over-approximated model. Typically this approach uses intervals or bounds to represent the range of possible variable values of parameters in the probabilistic program. However, existing volume computation techniques, such as those mentioned in [4, 13, 25] do not support posterior inference involving data observations. GuBPI [26] supports posterior inference by over-approximating posterior distributions using interval bounds, but it still suffers from imprecision and performance issues.

Furthermore, some approaches are dedicated to developing guarantees for sampling-based approximate algorithms, which include in-depth analysis and verification for algorithms like MCMC or importance sampling [27, 28]. These guarantees typically require the use of a specific variants of the MCMC implementation. As a result, it is not clear how to apply these analysis to many advanced MCMC algorithms like NUTS (No-U-Turn Sampler) [8]. Some other researchers concentrate on language-level properties. For example, Gorinova et al. [29, 30] provide guarantees for specific program transformations and slicing operations. Such technique ensures the transformations are implemented correctly, however, it does not address the correctness of program results nor does it consider data noise or model incompatibility with input data.

Beyond accuracy and robustness, trustworthiness in probabilistic programming also relates to a broader spectrum of concerns. One such concern is prior sensitivity, where changes in

prior parameters can significantly influence the results, highlighting the need for careful prior selection and sensitivity analysis [31, 32]. Additionally, the implementation of probabilistic programming systems themselves may contains software bugs [33, 34]. These topics do not fall directly within the scope of this thesis, which mainly focuses on accuracy and robustness.

1.2 DISSERTATION GOALS

As shown in Figure 1.1, building trustworthy probabilistic programming requires us to investigate into each stage and consider the interaction between the stages. The thesis statement of this dissertation is as follows:

Probabilistic programming can be made more trustworthy by leveraging statistical insights and program analysis techniques, particularly in addressing challenges of robustness to data perturbations and accuracy of inferences.

Diverging from the existing approaches, this dissertation takes a more systematic perspective, which has several unique aspects:

- Focus on Posterior Inference: Distinct from many probabilistic programming studies that do not involve conditional statements or data observations, our approach handles posterior inference, which involves continuous distributions, data observations, and computes the posterior distribution of the parameters by conditioning on data observations.

- Focus on Practical Implementation Issues: We also address the practical challenges faced by developers. For example, when using sampling-based inference algorithms, developers might encounter accuracy issues if the sampling algorithm does not converge within a limited number of iterations. Additionally, the convergence issues can be difficult to debug. Such practical issues are often overlooked in many theoretical studies.

- Model Robustness Against Data Noise: In practice, the collected data can often contain unexpected noise, which may break the assumptions of probabilistic model and significantly affect model accuracy. Our study focuses on improve or guarantee the robustness of these models against data noise. Specifically, we conduct several studies regarding how the model posteriors would change in the presence of data noise, and how to transform the model into a more robust version.

- Leveraging Hardware Capabilities: We also explore hardware acceleration for probabilistic programming, particularly through the use of parallel programming and advanced hardware resources like multi-core CPUs and GPUs, to boost the efficiency of analyses on probabilistic programs.

1.3 CONTRIBUTIONS

The dissertation presents the result of my two-pronged investigation in the field of probabilistic programming. In the first part of this dissertation, I identify multiple challenges related to the trustworthiness of current probabilistic programming practices, focusing on model robustness against data noise (introducing ASTRA) and the convergence of sampling-based inference algorithms (introducing SixthSense):

- **ASTRA**: This dissertation presents ASTRA, which is the first systematic study of effectiveness of robustness transformations on a diverse set of 24 probabilistic programs representing generalized linear models, mixture models, and time-series models. We evaluate five robustness transformations from literature on each model. We quantify and present insights on (1) the improvement of the posterior prediction accuracy and (2) the execution time overhead of the robustified programs, in the presence of three input noise models.

  To automate the evaluation of various robustness transformations, we developed ASTRA – a novel framework for quantifying the robustness of probabilistic programs and exploring the trade-offs between robustness and execution time. Our experimental results indicate that the existing robustness transformations are often suitable only for specific noise models, can significantly increase execution time, and have non-trivial interaction with the inference algorithms. Moreover, we observed that many robustness transformations tend to complicate the model, leading to non-convergence issues with certain inference algorithms, which motivated us to further study the causes of non-convergence.

- **SixthSense**: This dissertation presents SixthSense, a novel approach for predicting probabilistic program convergence ahead of run and its application to debugging convergence problems in probabilistic programs. SixthSense's training algorithm learns a classifier that can predict whether a previously unseen probabilistic program will converge. It encodes the syntax of a probabilistic program as *motifs* – fragments of the syntactic program paths. The decisions of the classifier are interpretable and can

be used to suggest the program features that contributed significantly to program convergence or non-convergence. We also present an algorithm for augmenting a set of training probabilistic programs that uses guided mutation.

  We evaluated SixthSense on a broad range of widely used probabilistic programs. Our results show that SixthSense features are effective in predicting convergence of programs for given inference algorithms. SixthSense obtained accuracy of over 78% for predicting convergence, substantially above the state-of-the-art techniques for predicting program properties Code2Vec and Code2Seq. We show the ability of SixthSense to guide the debugging of convergence problems, which pinpoints the causes of non-convergence significantly better by Stan's built-in warnings.

In the second part of this dissertation, I present the development of probabilistic inference strategies, featuring quantized inference for enhanced precision (introducing AQUA), and abstract interpretation for sound inference (introducing AURA):

- **AQUA**: This dissertation presents AQUA, a novel probabilistic inference algorithm that solves probabilistic programs with continuous posterior distributions. AQUA approximates programs using an efficient quantization of continuous distributions. Namely, it represents the distributions of random variables using quantized value intervals (Interval Cube) and corresponding probability densities (Density Cube). AQUA analysis transforms Interval and Density Cubes to compute the posterior distribution with bounded error. We also present an adaptive algorithm for selecting the size and the granularity of the Interval and Density Cubes.

  We evaluate AQUA on 24 programs from the literature. AQUA solved all of 24 benchmarks in less than 43s, with the median time impressively recorded at 1.35 seconds. It solves all the programs with a high-level of accuracy. When compared with state-of-the-art approximate algorithms like Stan's NUTS and ADVI, AQUA showed superior accuracy. Moreover, AQUA is able to support many continuous programs that exact inference tools such as PSI and SPPL find challenging.

- **AURA**: This dissertation presents AURA, a novel abstract interpretation for obtaining precise sound bounds on the posterior distributions computed by probabilistic programs. In addition to computing bounds on a single posterior distribution, when data observations are fixed constants, AURA allows programmers to specify interval bounds on uncertainty of data observations. Then AURA can abstractly interpret an infinite set of posterior distributions and certify bounds on probabilistic queries over those distributions.

AURA's key algorithmic contribution is to reduce the abstract interpretation of posterior distributions into tractable constrained optimization problems efficiently solvable by gradient ascent, by leveraging the concavity properties of a probabilistic program's log-likelihood. We prove AURA's optimization-based abstract semantics sound and additionally prove that we can combine this precise abstraction with standard abstract transformers in cases where the concavity properties hold for only part of a program. We then show how to use the posterior bounds computed by AURA to bound the probabilities of different queries.

We evaluate AURA on multiple case studies. In particular, we compare AURA's sound bounds on posterior distributions with a recent interval-based approach and an exact inference engine. Our experimental evaluation on 19 programs shows that AURA can efficiently compute distribution bounds (in an order of seconds) for more programs and obtain an order of magnitude more precision than the state-of-the-art approaches. Additionally, AURA's formal certification scales to order of magnitude more observed data than any existing work.

CHAPTER 2: ASTRA: UNDERSTANDING THE PRACTICAL IMPACT OF ROBUSTNESS FOR PROBABILISTIC PROGRAMS

2.1 INTRODUCTION

Probabilistic programming (PP) has recently emerged as a general and flexible approach for Bayesian inference [1, 8, 35]. PP decouples model specification from the inference procedures, and thus allows the users to update their models while automatically applying general inference algorithms for Markov Chain Monte Carlo (MCMC) sampling [36] or Variational Inference (VI) [37]. In recent years, PP has been applied in various real-world machine learning applications, e.g., forecasting [38], recommendations in social networks and predicting user locations [39, 40], rating players in games [41], and COVID-19 modeling [3].

Automatically deploying probabilistic programs on such a diverse set of real-world applications raises the question of how much the results of their inferences change in presence of outliers and other deviations of data from model's assumptions (which we summarily call *noise*). *Robustness* is the property of systems, including probabilistic programs, to remain unaffected by data noise [42]. For instance, many statistical models assume a Gaussian prior or likelihood, but few data points that are far away from the rest can significantly change the inferred mean. In contrast, inference using robust models is more likely to yield posteriors that are not affected by such noise.

Robustness transformations have been traditionally custom-designed for specific models, such as linear regression. Some common transformations can however be applied across different model classes, for instance, by replacing Gaussian likelihood with Student-T. However, it remains unknown (1) which robustness transformation to apply to obtain most robust inference result for a given model and noise model, and (2) how off-the-shelf inference algorithms in popular PP languages interact with these transformations– e.g., which ones have higher execution overhead. Most previous works consider only a few examples, without any thorough comparisons or systematic run time measurements. However, these questions become particularly important when the models are deployed in real-world settings where both accuracy (inference results) and execution cost matter.

Example: How Good are the Robust Models? Figure 2.1 presents the posteriors fitted on a corrupted dataset when applying different robustness transformations on the original model. The model $y \sim \mathcal{N}(\beta, 1)$ fits the parameter β as the mean of data y. In the dataset, 80% data points are good (non-noisy) observations centered at 0, while 20% are outliers centered at -10, far from the good observations. We present the posterior distributions of β obtained from original and robust models as different lines on the plot.

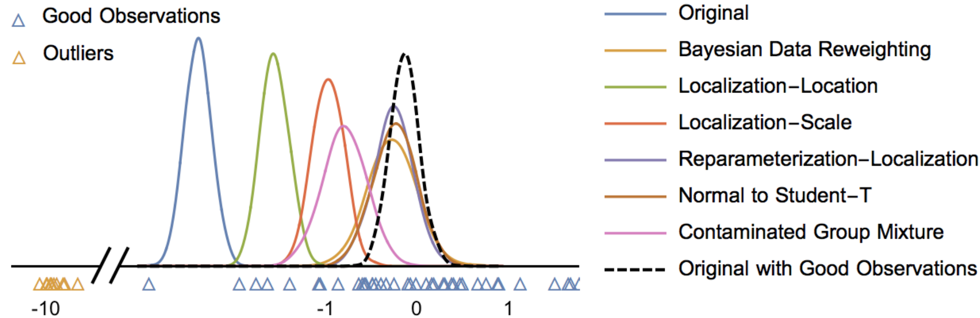


Figure 2.1: Posteriors of Different Robust Models Fitted with Corrupted Data

We observe that all robust models are not equally useful: while some (e.g., Reweighting) yield posteriors that are close to ideal in presence of outliers, some others (e.g., Localization) are not too different than the original (non-robust) model. These observations motivate the need for a systematic study of robustness transformations.

Our Work. The goal of our work is to develop a *systematic understanding* of how various robustness transformations perform in different scenarios through rigorous empirical evaluation on a broad range of subjects. We study the impact on the performance (accuracy) and execution cost of *four factors*: (1) Inference Algorithms, (2) Noise Models and Noise Levels, (3) Model Class, (4) User time budget.

We present the first extensive study of different robustness transformations on 24 probabilistic models from three classes: generalized linear models, mixture models, and time-series models. To help users understand both the practical and fundamental properties of probabilistic robustness transformations, we developed the **ASTRA** framework. ASTRA automatically modifies the program code to apply the robust transformation (and check for its legality) and systematically evaluates different robustness transformations for user-defined input noise models and posterior accuracy metrics. ASTRA then ranks the transformed programs by predictive accuracy. ASTRA is *extensible*: users can easily add new noise models, transformations, and accuracy metrics.

We implemented three common *noise models* for corrupting the datasets (Section 2.6): (1) *Simple Outliers* randomly changes the value of several data points, (2) *Introducing Hidden Groups* corrupts the data by adding a new distribution mode, and (3) *Skewing Data* adds non-symmetric error to most data points to skew the distribution. We also implemented five robustness transformations from literature for each model (we describe them in Section 2.3): (1) *Bayesian Data Reweighting* [43], (2) *Localization* [44], (3) *Robust Reparameterization* combines reparameterization from [45] with localization, (4) *StudentT* transformation of Gaussian variables, and (5) *Contaminated Group Mixture* [46].

We analyze the posterior predictive accuracy of the robustified models and their execution

times using two state-of-the-art inference algorithms: No U-Turn Sampler (NUTS) and Automatic Differentiation Variational Inference (ADVI), implemented in Stan [8].

Results and Insights. Our study yields several interesting insights and observations:

- Different inference algorithms respond differently to each robustness transformation. For instance, for Simple Outliers noise model, Student-T always performs better than Reparameterization for ADVI but for NUTS, Reparameterization outperforms Student-T.
- Robustness transformations can be effective for some noise models – in particular for Simple Outliers – even when 10% of the data has been replaced with outliers the robustness transformations reduce the error by up to 3x, compared to the original program on the same data. However, most transformations do not generalize well across different noise models. For instance, all transformations provide very limited benefits for Hidden Group and Skewed data attacks – this motivates future research to develop novel robustness transformations for these attacks.
- Robustness transformations incur greater overheads for NUTS than ADVI. The run time overheads (over original model) for ADVI range between 1.04x and 7.03x, while for NUTS they are between 1.76x and 14.5x. Hence, some transformations may be impractical in scenarios with tight time budgets. We present more insights in Section 2.7.

Contributions. This chapter makes several contributions:

- **Automated Robustness Evaluation:** We develop ASTRA, a novel automated system that efficiently evaluates the robustness transformations for probabilistic programs.
- **Systematic Evaluation of Robustness:** We present an extensive study of 24 probabilistic programs with multiple robustness transformations, input noise models, and inference algorithms. Our results inform how users select a robustness transformation for their use cases.
- **Insights:** We demonstrate that the robustness transformations can effectively improve predictive accuracy for some models of noisy data, but they may also incur significant execution time overhead. Using ASTRA, we obtained numerous useful insights that are beneficial for both the users and researchers of the probabilistic programming community in particular and AI in general.

  ASTRA is open sourced at `https://github.com/uiuc-arc/astra`.

2.2 ROBUSTNESS METRICS

To evaluate model robustness, we follow the standard methodology in existing research on robust Bayesian modelling [43, 44], by injecting noise in the observed data and computing the relative change in posterior predictive accuracy of the model.

Table 2.1: Robustness Transformations: Original and Transformed Models ($\alpha, \beta, s, \eta, \nu, \rho, \sigma, z$ are parameter variable placeholders; y is a data variable placeholder; F, π are distribution placeholders)

$\beta \sim \pi_\beta(\alpha)$ $y_{i=1}^D \sim F(\beta)$ $\Downarrow$ $w_{i=1}^D \sim Beta(\gamma, \zeta)$ $\beta \sim \pi_\beta(\alpha)$ $y_{i=1}^D \sim F(\beta)^{w_i}$	$\beta \sim \pi_\beta(\alpha)$ $y_{i=1}^D \sim F(\beta)$ $\Downarrow$ $\beta \sim \pi_\beta(\alpha)$ $s \sim Unif(0,1)$ $\eta_{i=1}^D \sim \mathcal{N}(\beta, s)$ $y_{i=1}^D \sim F(\eta_i)$	$\beta \sim \pi_\beta(\alpha)$ $y_{i=1}^D \sim \mathcal{N}(\beta, \sigma)$ $\Downarrow$ $\nu \sim \pi_\nu(\gamma)$ $\beta \sim \pi_\beta(\alpha)$ $y_{i=1}^D \sim \mathcal{T}(\nu, \beta, \sigma)$	$\beta \sim \pi_\beta(\alpha)$ $y_{i=1}^D \sim \mathcal{N}(\beta, \sigma)$ $\Downarrow$ $\nu \sim \pi_\nu(\gamma)$ $\tau_{i=1}^D \sim Gamma(\frac{\nu}{2}, \frac{\nu}{2})$ $\beta \sim \pi_\beta(\alpha)$ $y_{i=1}^D \sim \mathcal{N}(\beta, \frac{\sigma}{\sqrt{\tau_i}} \cdot)$	$\beta \sim \pi_\beta(\alpha)$ $y_{i=1}^D \sim F(\beta, \sigma)$ $\Downarrow$ $\rho_{out}, \eta_{out}, \sigma_{out} \sim \pi(\gamma)$ $\nu \sim \mathcal{N}(\eta_{out}, \sigma_{out})$ $\beta \sim \pi_\beta(\alpha)$ $z_{i=1}^D \sim Bernoulli(\rho_{out})$ $y_i \mid z_i = 0 \sim F(\beta, \sigma)$ $y_i \mid z_i = 1 \sim F(\beta, \sqrt{e^\nu})$
(a) Reweighting	(b) Localization	(c) Normal-to-Student-T	(d) Reparameterization	(e) Cont. Mixture

Given a probabilistic program P, and the observed dataset y (we will also call it *uncorrupted*), we fit P to a *corrupted dataset* y^{Noise} that is generated by injecting noise in y. We use the fitted posterior of P to generate predicted data $\hat{y}$, and we evaluate the *robustness* of this program through the mean squared error (MSE) metric, as

$$MSE(\hat{y}, y) = \frac{1}{D} \sum_{i=1}^{D} (\hat{y}_i - y_i)^2, \tag{2.1}$$

where D is the size of the dataset. Intuitively, *MSE* quantifies by how much the posterior predictive accuracy changes in presence of data corruptions. Computing *MSE* using predictive data is recommended by [47] as the posterior predictive check to evaluate model fitting and is also used by [44] as the predictive R2 metric to evaluate model robustness.

Since the value of *MSE* depends on the scale of data values, we standardize the *MSE*s on the original model, following [48]. Specifically, let $MSE(\hat{y}, y)$ and $MSE(\hat{y}_T, y)$ be the estimated robustness of the original model P and a transformed model P_T, respectively. Then we define the *relative improvement of robustness* of the transformed model as:

$$RIMSE(\hat{y}, \hat{y}_T, y) = MSE(\hat{y}, y) \,/\, MSE(\hat{y}_T, y). \tag{2.2}$$

Intuitively, *RIMSE* denotes the relative improvement of the "robustified" model over the original model. *RIMSE* >1 indicates improved robustness, *RIMSE* of 1 indicates no improvement, and *RIMSE* <1 indicates that the accuracy of robustified model is lower than the original model. In our example (Figure 2.1), the best transformation (Reweight) yields a RIMSE of 5.22, whereas the least useful transformation (Localization-Location) yields a RIMSE of 1.31.

2.3 ROBUSTNESS TRANSFORMATIONS

We describe various robustness transformations for probabilistic models from the literature that we use in our study.

Bayesian Data Reweighting. This transformation changes the contribution of each data sample (observation) by raising its likelihood term in the model to its own weight [43]. The weights are then exposed as latent variables and inferred along with the rest of the model's parameters. During inference, the outliers are automatically assigned lower weights, which improves prediction. Table 2.1(a) presents an example of an original model and its transformed version. The transformation introduces a vector of weights w with a Beta prior. $y_i \sim F(\beta)^{w_i}$ denotes that the likelihood F for each data sample is raised to the power of its weight.

Localization. This transformation allows each likelihood term to depend on its own copy of latent variable [44]. Table 2.1(b) presents an example original and robustified probabilistic models. In the transformed model, there are D local versions of the latent variables: η_i, one for each data point y_i. All the auxiliary local variables are sampled from prior π_η. We use a Gaussian prior for η_i in evaluation, following the examples in the original work [44]. Unlike [44] that designs a specialized E-M algorithm to fit s in the Gaussian prior, we fit s with other parameters via Bayesian inference.

Normal to Student-T. Normal distribution is not robust to outliers or over-dispersed data. An easy alternative is the Student-T distribution [46]. Intuitively, the fatter tail of Student-T can better capture the data points far away from the majority. In this transformation, ASTRA replaces a Normal distribution with Student-T by preserving the location and scale parameters in the program, while adding a new parameter ν as the degree of freedom (DOF). Table 2.1(c) presents the transformation. In the transformed model, ν is from the prior π_ν. Since we may not have prior knowledge, we use a uniform (non-informative) prior for ν.

Reparameterization and Localization of the Scale Parameter. This transformation changes the Gaussian likelihood distribution to an equivalent of Student-T distribution and also localizes the additional parameter τ. Table 2.1(d) presents an example. The transformation adds D parameters τ_i to adjust the standard deviation of the likelihood for each data point. This has similar effects as the Localization transformation. τ_i is from a *Gamma* prior with hyper-parameter ν. If we integrate out all the τ_is, ν will be equivalent to the DOF parameter in the Normal to Student-T transformation [45] (but can be more amenable when sampled with MCMC algorithms). This transformation is only applicable for Normal distributions.

Contaminated Group Mixture. To make the model capture a small amount of corruption in data, we can encode in the model that the data is from a mixture of the original model

and a outlier group [46]. Table 2.1(e) presents an example. With probability $1 - \rho_{out}$, the data point is from the original model; with ρ_{out}, the data point comes from another distribution with a different (likely larger) variance. Benefitting from the outlier group, the contaminated data will not directly affect the original model's parameters. ρ_{out} and the scale of the new group are latent parameters which can adapt to the user's data. To ensure a positive scale parameter, we set the outlier group scale parameter to be $\sqrt{e^{\nu}}$ where ν is another hyper-parameter.

2.4 ASTRA

At a high level, ASTRA takes a probabilistic program P, a dataset y, the desired noise model A, inference algorithm I, and a set of transformations T to apply on P. ASTRA first generates the transformed programs by applying each transformation in T to P. ASTRA then compares each transformed program against the original program and returns the list of transformed programs and their corresponding robustness *scores*, sorted in decreasing order of their robustness.

2.4.1 Probabilistic Program Transformations

Probabilistic Programs. ASTRA takes a probabilistic program (PP) in Stan probabilistic programming language [8] as input, which is to encode a probabilistic model in the form of a program. Figure 2.2 shows the Stan program for the original model in the motivating example (Figure 2.1). The representation is intuitive: the `data` block declares N observations of data y; the `parameters` block declares one parameter `b` in the model; and the `model` block encodes that each data observation is conditional on `b`. Given such a probabilistic program, Stan can automatically apply inference algorithms like MCMC or VI to compute the posterior of parameters.

```
1 data {
2   int<lower=0> N;
3   vector[N] y;
4 }
5 parameters {
6   real b;
7 }
8 model {
9   for (i in 1:N)
10    y[i]~normal(b,1);
11 }
```

Figure 2.2: Example PP

Transformations. To allow automated transformations on the probabilistic program, we use Storm-IR [34] as our internal representation. Storm-IR can represent program 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 a compiler CFG [49]). Since Storm-IR supports multiple languages (e.g., Stan, Pyro, Edward), it allows ASTRA to be language-agnostic. ASTRA first parses the original probabilistic program into abstract syn-

tax tree and converts to Storm-IR. On this IR, searching for the code pattern from Table 2.1 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., $\beta \mapsto$ `b`, $y \mapsto$ `y`) and distributions (e.g., $F \mapsto \mathcal{N}(\texttt{b}, \texttt{1})$). ASTRA uses the identified distributions/variables to instantiate the transformation template and update the program. For example, to apply the Normal-to-Student-T transformation on Figure 2.2, ASTRA will replace the normal distribution on Line 10 with a Student-T distribution, as `student_t(nu,b,1)`, where `nu` is a new parameter for the degree of freedom. ASTRA will also place a uniform prior on `nu`.

After identifying the pattern, ASTRA also 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 2.3 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.

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 $\boldsymbol{w}$[D]	init. weights.
for ($\boldsymbol{i}$ = 1..D)	
$\boldsymbol{w}[\boldsymbol{i}]$:= *Beta*($\boldsymbol{\gamma}, \boldsymbol{\eta}$)	reweighting dist.
...	
for (i = 1..D)	
factor(*DistExpr*$_2$(x_1).pdf($y[i]$) * $\boldsymbol{w}[\boldsymbol{i}]$)	reweighted obs.
return $x_1, \boldsymbol{w}$	posteriors

Figure 2.3: Reweighting Transformation Code Pattern

Localization. Figure 2.4 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.

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)
$\boldsymbol{s}$ **:=** $\boldsymbol{Unif}(0,1)$	new hyper-prior
var $\boldsymbol{\eta}$[D]	localized params.
for ($\boldsymbol{i}$ = 1..D)	
$\boldsymbol{\eta[i]}$:= $\boldsymbol{Normal(x_1, s)}$	new priors
...	
for (i = 1..D)	
factor($DistExpr_2$(...,$\boldsymbol{\eta[i]}$,...).pdf($y[i]$))	localized obs.
return $x_1, \boldsymbol{\eta}$	posteriors

Figure 2.4: Localization Transformation Code Pattern

Normal to Student-T. Figure 2.5 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.

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
$\boldsymbol{\nu}$ **:=** $\boldsymbol{Unif}(\ldots)$	new hyper-prior
...	
for (i = 1..D)	
factor($\boldsymbol{StudentT}(\boldsymbol{\nu}, x_1, x_2)$.pdf($y[i]$))	Student-T dist.
return $x_1, \boldsymbol{\nu}$	posteriors

Figure 2.5: Normal/Student-T Transformation Code Pattern

Reparameterization and Localization of the Scale Parameter. We present the code pattern for this transformation in Figure 2.6. 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.

```
x1 := DistExpr1                                        prior mean
x2 := DistExpr2                                        prior std
...
for (i = 1..D)
  factor(Normal(x1, x2).pdf(y[i]))                     conditioning
return x1, x2                                          Gauss. posterior
                          ⇓
x1 := DistExpr1                                        prior mean
x2 := DistExpr2                                        prior std
ν := DistExpr3                                         new hyper-prior
for (i = 1..D)
  τ[i] := Gamma(ν/2, ν/2)                              robustness factors
...
for (i = 1..D)
  factor(Normal(x1, x2/sqrt(τ[i])).pdf(y[i]))          conditioning
return x1, x2                                          Gauss. posterior
```

Figure 2.6: Reparameterization Transformation Code Pattern

Contaminated Group Mixture. We present the code pattern for this transformation in Figure 2.7. 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.

```
x1 := DistExpr1                                        prior mean
x2 := DistExpr2                                        prior std
...
for (i = 1..D)
    factor ( Dist(μ, σ, ...).pdf (Expr2) )             conditioning
return x1, x2                                          Gauss. posterior
                          ⇓
x1 := DistExpr1                                        prior mean
x2 := DistExpr2                                        prior std
ρ_out := Unif(0, 0.5)                                  outlier probability.
μ_out := DistExpr3                                     outlier mean
s_out := DistExpr4                                     outlier variance
out := LogNormal(μ_out, s_out)                         outlier probability.
...
if(Bernolli(1 − ρ_out))                                mixture
  for (i = 1..D)
    factor(Dist(x1, x2).pdf(y[i]))                     conditioning
else                                                   Gauss. mixture
  for (i = 1..D)
    factor(Dist(x1, sqrt(exp(out)).pdf(y[i]))          conditioning
return x1, x2                                          Gauss. posterior
```

Figure 2.7: Cont. Mixture Transformation Code Pattern

2.4.2 ASTRA Algorithm

Algorithm 2.1 presents ASTRA's main algorithm. First, ASTRA initializes a set, *Results*, for storing the robustness scores of all transformed programs (L.2). ASTRA generates the transformed programs $\boldsymbol{P_T}$ (L.3). Next, ASTRA evaluates the robustness of each transformed program (L.4-13). For each transformed program, $P_T \in \boldsymbol{P_T}$, ASTRA performs the following steps `N` times: it first generates a noisy dataset, y^{Noise}, using the specified noise model A (L.7). It runs the inference algorithm I selected by the user to estimate the latent parameters (or posterior data predictions), $\hat{y}$, in program P using the noisy dataset y^{Noise} (L.8). Besides, the user also specifies other inference specifications such as number of samples (for MCMC) or number of iterations (for VI). The *Infer* method encapsulates this step. ASTRA infers the parameters of the transformed program P_T on the same noisy dataset (L.9), and computes the robustness score using the *RIMSE* metric (L.10).

Algorithm 2.1 ASTRA Algorithm

Input: Program P, Data y, Noise Model A, Inference Algo I, Transformations T
Output: Transformed Programs Ranked by Robustness

1: **procedure** ASTRA(P, y, A, I, T)
2: $\quad Results \leftarrow \emptyset$
3: $\quad \boldsymbol{P_T} \leftarrow ApplyTransforms(P, T)$
4: $\quad$ **for** $P_T \in \boldsymbol{P_T}$ **do**
5: $\quad\quad Score \leftarrow \emptyset$
6: $\quad\quad$ **for** $i \leftarrow 1$ **to** `N` **do**
7: $\quad\quad\quad y^{Noise} \leftarrow A(y)$
8: $\quad\quad\quad \hat{y} \leftarrow Infer(P, y^{Noise}, I)$
9: $\quad\quad\quad \hat{y}_T \leftarrow Infer(P_T, y^{Noise}, I)$
10: $\quad\quad\quad Score \leftarrow Score \cup \{RIMSE(\hat{y}, \hat{y}_T, y)\}$
11: $\quad\quad$ **end for**
12: $\quad\quad Results \leftarrow Results \cup \{(P_T, Avg(Score))\}$
13: $\quad$ **end for**
14: $\quad$ **return** $Sort(Results)$
15: **end procedure**

ASTRA computes the average score (e.g. arithmetic or geometric mean) for the transformed program P_T and appends the result to the *Results* set (L.12). Averaging the scores over multiple runs (and different noisy data-sets) produces a better estimate of the robustness of a transformation. Finally, ASTRA returns the list of transformed programs in descending order of their robustness scores (L.14).

ASTRA also supports other user-specified robustness metrics, which can be specified as a simple function using our python interface. Further, unlike [44]'s approach that uses only synthetic data (simulated from the original model with known parameters) as y, ASTRA allows users to provide the uncorrupted data as y if the true data model is unknown. Given the uncorrupted data y, ASTRA helps users to compare how different models fit to y.

We formally state that the transformations we define in Section 2.4.1 have the semantic effects as proposed in the statistical literature (as summarized in Table 2.1). We leverage Stan's operational semantics from [30].

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 and Stan syntax [30]. For example, let $\Leftrightarrow$ denote the translation relation between a StormIR expression/statement and a Stan expression/statement, then factor translation rule is:

$$\frac{E_{storm} \Leftrightarrow E'_{stan}}{\texttt{factor}(E_{storm}) \Leftrightarrow \textbf{target} = \textbf{target} + \log(E'_{stan})}. \tag{2.3}$$

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 2.1. 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 2.4.1.

Definition 2.2. 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 2.1 where $\boldsymbol{\theta}$ represents all the parameters in the program and y is the data.

Theorem 2.1. 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 [30], we know that there exists a unique end state s for S as $((y, \boldsymbol{\theta}, \textbf{target} \mapsto 0), S) \Downarrow s$ where $s[\textbf{target}] = \log p^*(\boldsymbol{\theta}|y)$. $p^*(\boldsymbol{\theta}|y)$ 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 $s_T[\textbf{target}] = \log p^*_T(\boldsymbol{\theta}|y)$, 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 2.3, 2.4, 2.5, 2.6, and 2.7 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 2.1, and between $p_T^*(\boldsymbol{\theta}|y)$ and $p_T(\boldsymbol{\theta}|y)$.

Table 2.2: Description of Benchmarks

Prog	Name	Description	#Param	#Data	ADVI	NUTS
RA	anova_radon_nopred_chr	Multi-level linear model with set up for ANOVA with Choo-Hoffman Parametrization	88	919	10.38	20.28
RE	electric_chr	Multi-level linear model with varying intercept with Choo-Hoffman Parametrization	100	192	5.65	12.57
RG	flight_simulator_17.3	Varying intercept model	17	281	4.25	12.28
RK	hiv	Multi-level linear model with varying slope and intercept	173	369	7.00	20.33
RL	lightspeed	Linear model with no predictors	2	66	0.68	0.69
RN	pilots	Multi-level linear model with varying intercept and redundant parameterization	17	40	1.57	3.17
RQ	radon_no_pool	Multi-level linear model without pooling	89	919	12.64	13.60
RR	radon.pooling	Multi-level linear model with complete pooling	3	919	7.18	5.53
RU	radon_vary_si	Multi-level linear model with group level predictors	175	919	14.77	33.95
RV	unemployment	Linear model with one predictor	3	57	0.48	1.46
RW	wells_dae	Logitistic regression model	4	3020	14.06	113.85
RX	y_x	Linear model with one predictor	3	919	7.93	9.17
RY	kidscore_momwork	Linear model with discrete predictor	5	434	3.98	9.26
SA	gp-fit-latent	Gaussian process (GP) with exponentiated quadratic kernel and Gaussian likelihood	104	101	150.63	732.07
SB	stochastic-volatility	Moving average model for time-series	503	500	11.52	170.22
SC	gp-fit-pois	GP with exponentiated quadratic kernel and Poisson likelihood	104	101	108.98	697.92
SD	gp-fit-ARD	GP with ARD-parameterized exponentiated quadratic kernel and Gaussian likelihood	105	101	141.25	1188.97
SE	koyck	Geometric lag time-series	4	200	1.88	6.61
MA	normal_mixture_k_prop	Mixture model with unknown locations, scales and mixing proportion	11	1000	9.98	601.02
MB	normal_mixture_k	Mixture model with unknown locations, scales and mixing proportion	9	1000	1.98	80.11
MC	normal_mixture	Mixture model with known scale	3	1000	25.49	27.91
MD	gauss_mix_asym_prior	Mixture model with non-exchangeable priors	5	100	1.75	3.28
ME	gauss_mix_given_theta	Mixture model with known mixing proportion	4	1000	19.81	54.51
MF	gauss_mix_ordered_prior	Mixture model with ordered priors	5	1000	14.08	30.30

2.6 METHODOLOGY

Probabilistic Models. To evaluate the transformations in ASTRA, we obtain a set of 24 probabilistic programs from a popular repository [50] including 13 Regression models, 5 Time-Series/State-Space models, and 6 Mixture models. Table 2.2 presents the details of all probabilistic programs we evaluate using ASTRA, their description, number of parameters and data items, and run times (in seconds) for ADVI and NUTS inference algorithms with Stan. The TimeSeries models start with "S-", Mixture models with "M-", and Regression models with "R-".

Automated Inference. We use NUTS [9] and ADVI [51] inference algorithms from Stan [8] – a popular probabilistic programming language – to run each transformed program and compare their relative behaviors. For NUTS, we run each program with 4 chains, 1000

warmup iterations, and 1000 sampling iterations with a timeout of 8 minutes for each chain. We exclude the programs that timed out. For ADVI, we use 10000 iterations and 1000 posterior samples for comparison. For our evaluation, we use Azure VMs, each with 4 cores, 2.3 GHz CPU, and 16 GB RAM.

Robustness Metric. We use the *RIMSE* metric defined in Section 2.2. For each model, we repeat generating noise and inference 5 times and compute the geometric mean of *RIMSE* scores. We chose *RIMSE* instead of other information criteria used in [47] for model selection because there are several challenges for applying them on general probabilistic programs: AIC [52] does not work under strong priors; DIC [53] gives poor results when the distributions are not well summarized by mean; WAIC [54] and Cross-Validation [55] require data partitioning, which is hard to automate for structured models; Bayes factor method [56] only works well for discrete models [57].

Convergence Metric. The sampling-based automated inference may suffer from non-convergence and result in inaccurate estimation of the result. Robustness transformations that introduce new parameters can make the program harder to converge and thus affect its accuracy and robustness. Hence, for evaluation, we also measure the convergence score using Gelman-Rubin Diagnostic [47]. A score significantly larger than 1 indicates non-convergence.

Noise Models. The dataset $\mathcal{D}$ is usually composed of response data (labels) and explanatory data (features) with the same length. In this work, we only add noise to the response data. We select five noise levels for the fraction of perturbed data inputs between 2%, 4%, 6%, 8%, and 10%. Hereon we denote the response data as $\boldsymbol{y}$ and its size as D:

• **Adding Outliers.** We randomly select a subset of data points and add random noise to them. Let $sd(\boldsymbol{y})$ be standard deviation estimated from the original dataset $\boldsymbol{y} = \{y_1, y_2, \dots, y_D\}$, then we simulate the outliers by:

$$z_{i=1\dots D} \sim Bernoulli(k\%) \tag{2.4}$$

$$y_i^{Outliers} | z_i = 1 \sim \mathcal{N}(c \cdot y_i, |y_i| \cdot sd(\boldsymbol{y})) \tag{2.5}$$

where $k\%$ corresponds to the amount of noise, can be specified by the user. The constant $c > 1$ allows us generate outliers far from the typical observations. In our experiment, we let $c = k$. This noise model simulates a scenario where some observations get corrupted due to some exception or failure (e.g., of a sensor, storage, or network).

• **Introducing Hidden Groups.** This strategy introduces a hidden group (with its own mode) that does not agree with the modeling assumptions. The location and scale of the hidden group is controlled by the noise level k. Similar to previous case, we allow the user

to specify the size of data subset to be changed (e.g., our experiments use $c = 20\%$).

$$z_{i=1...D} \sim Bernoulli(c) \tag{2.6}$$

$$y_i^{Hidden_Group} | z_i = 1 \sim \mathcal{N}(y_i + \frac{k}{2} \cdot sd(\boldsymbol{y}), 0.1k) \tag{2.7}$$

• **Skewing data.** Using this strategy, we skew the distribution of data points. Skewing causes the mean of the distribution to shift and lose symmetry. It also makes it harder for the inference strategy to sample using a non-robust model. In this work, we apply positive skew to the datasets – for data $\boldsymbol{y}$ we generate skewed data as follows:

$$y_i^{Skewed} = \left(\frac{y_i - y_{min}}{y_{max} - y_{min}}\right)^{(1+0.1k)} \cdot (y_{max} - y_{min}) + y_{min} \tag{2.8}$$

where k is the noise level for this noise model. We first scale all the data to $[0, 1]$, then we raise it to a chosen power to skew the data, and finally scale it back to $y_{min} = \min_{i=1...D} y_i$ and $y_{max} = \max_{i=1...D} y_i$.

A reproducible version of ASTRA can be conveniently accessed at the following online location: `https://figshare.com/s/38668524113696505ef4`.

2.7 EVALUATION

2.7.1 Performance of Transformations

We apply the following transformations (discussed in Section 2.3): Robust reweighting data (*Reweight*), Localization of location parameter (*Local-Loc*), Localization of scale parameter (*Local-Scale*), Reparameterization and Localization of scale parameter (*Reparam*), Normal to StudentT (*StudentT*) and Contaminated Group Mixture (*Mixture*). To evaluate the transformations using ASTRA, we apply 3 different noise models (Outliers, Hidden Groups, and Skewing) on the datasets obtained from 24 programs.

General Trends of Different Transformations. Figures 2.8 and 2.9 present the geomean of the relative improvement of MSE (RIMSE) by different robustness transformations at different noise levels for ADVI and NUTS algorithms respectively. Each sub-plot presents the results for one noise model. The X-axis represents the noise level while the Y-axis represents the *geometric mean* of RIMSE over all programs. Each line in the plots represent the performance of one transformation. We also present the MSE of the original (non-robust) program at all noise levels below the X-axis (as "Orig MSE"). The robust transformations reduce MSE by the factor represented on the Y-axis (e.g., up to 3.31x for StudentT transformation for Outliers (ADVI)).

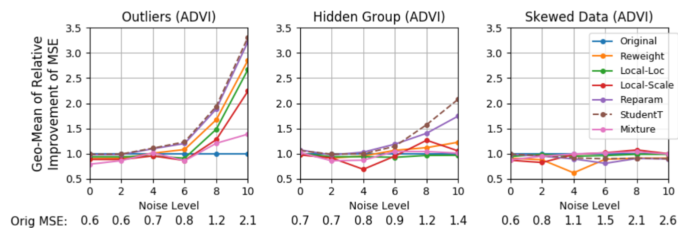


Figure 2.8: Mean Improvement of Transformed Programs at Different Noise Levels (ADVI)

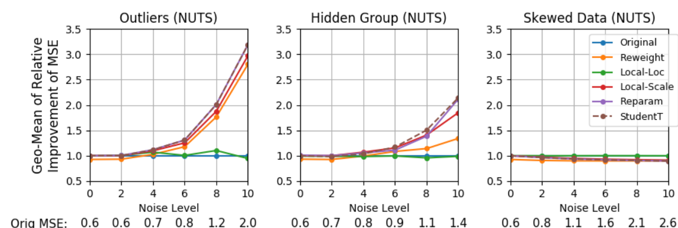


Figure 2.9: Mean Improvement of Transformed Programs at Different Noise Levels (NUTS)

Table 2.3: MSE Improvement at Noise Level 10 (Outliers)

Prog	ADVI	NUTS
RE	256.42 (StudentT)	412.60 (StudentT)
RV	28.04 (StudentT)	31.94 (Reparam)
MC	27.48 (Local1)	1.00 (Original)
SE	14.23 (StudentT)	16.02 (Reweight)
RK	8.41 (StudentT)	9.25 (Reparam)
RN	7.11 (Reparam)	6.25 (Local2)
RU	3.42 (StudentT)	3.75 (StudentT)
RA	3.31 (StudentT)	3.19 (Reparam)
MF	3.27 (StudentT)	2.81 (Reparam)
RQ	3.23 (StudentT)	3.78 (StudentT)
RR	2.95 (Reparam)	3.00 (Reparam)
RX	2.93 (StudentT)	3.18 (Reparam)
SD	2.52 (StudentT)	3.52 (StudentT)
MD	2.21 (Reweight)	6.08 (Reweight)
ME	1.27 (StudentT)	1.41 (Reparam)
RY	1.25 (StudentT)	1.00 (Original)
MB	1.14 (StudentT)	1.22 (StudentT)
RG	1.04 (StudentT)	1.03 (Reweight)
SA	1.02 (Mixture)	1.56 (Reparam)
RW	1.00 (Reweight)	1.00 (Reweight)
SB	1.00 (StudentT)	1.00 (Original)
SC	1.00 (Original)	1.05 (Local1)
RL	1.00 (Original)	1.00 (Original)
MA	1.00 (Original)	1.68 (StudentT)

(R-): Regression, (M-): Mixture, (S-): TimeSeries

Overall, the transformations are most effective for the *Outliers* noise model. The improvements are significantly smaller for *Hidden Group*. For *Skewed Data* noise model, none of the transformations are effective because the noisy samples are harder to distinguish from typical observations. In general, RIMSE increases with higher noise level, showing that the transformations are more helpful when there is more corruption in the data.

Insight 1. *Our results show that most transformations do not generalize well beyond the Outliers noise model and provide limited benefits. Hence, there is a need to develop novel robustness transformations, especially for Hidden Group and Skewed Data noise models.*

For the Outliers and Hidden Group noise models, *StudentT* transformation is the best in most cases, closely followed by *Reparam*. However, Reparam requires inferring many more parameters than StudentT (e.g., for D data points, StudentT transformation adds one more parameter while Reparam adds $D+1$ auxiliary parameters), which increases the run time of inference (see also RQ3).

The Local-Loc and Local-Scale transformations provide less protection from noisy data. Local-Scale may help improve the accuracy with NUTS, but it is likely to diverge when using ADVI (Table 2.4), leading to inaccurate results. One potential cause for this may be that we infer the hyper-parameters for localization transformations in the Bayesian

model using automated inference along with other parameters. It may be possible to obtain a better result by applying the E-M algorithm proposed in [44], which is customized for each model. However, it is unclear how to automatically apply such an algorithm for general probabilistic programs.

2.7.2 Predictive Accuracy Improvement

Table 2.3 presents the RIMSE scores for all programs with the Outliers noise model at noise level 10 for ADVI and NUTS. Each row represents one program. Each column presents the largest improvement of MSE and the name of the transformation that enabled this improvement in parentheses. For example, "256.42 (StudentT)" means that StudentT is the best among all the transformations (and the original one) and yields 256.42x reduction of the MSE of the original program. For the RIMSE scores with other noise models, see Appendix A.1. A larger value means the posterior obtained by the transformation is closer to the posterior based on the non-noisy data. “Local1" stands for Local-Loc and “Local2" stands for Local-Scale. We do not apply Hidden Group noise model on Mixture models and Skewed Data noise model on programs with binary data since they are unsuitable. In summary, when using ADVI, StudentT provides the best improvement on 15 benchmarks, followed by Original which is the best on 3 benchmarks, while the other transformations only lead on fewer than 3 benchmarks each. When using NUTS, Reparam is the best on 8 benchmarks; StudentT is the best on 6 benchmarks; Reweight and Original both dominate 4 benchmarks; Local1 and Local2 both dominate one.

Characteristics for Different Model Categories. Generally, Regression (R-) models show the largest improvement in robustness, while Time-Series models (S-) show the smallest improvement. We observe substantial improvements in most linear regression models (e.g. RE and RV), since most transformations are designed for such models. However, for a logistic regression model (RW), we observe very small improvements (up to 1.00x). This is because this model already has a high tolerance for noise compared to other models, since the noise is limited between 0 and 1 for binary data, and thus makes most transformations redundant.

Most Time-Series models model the auto-correlation within data points or fit a correlation matrix for Gaussian processes. As a result, small noise in the data may not affect the fitted correlation. For instance, we observe that the MSE scores of the original models of SA and SB are not affected as the noise level increases. Further, since the robustness transformations are generally designed for exchangeable data [43], they are unlikely to work well for many Time-Series models. For instance, for models SC and SD, the transformations are not as

Table 2.4: (Geometric-)Mean of Convergence Score (Gelman-Rubin Diagnostic) at Noise Level 10

Transformations	**Outliers**		**Hidden Group**		**Skewed Data**	
	ADVI	NUTS	ADVI	NUTS	ADVI	NUTS
Original	2.12	1.60	1.25	1.00	2.15	1.19
Reweighting	1.40	1.15	1.20	1.01	1.36	1.06
Localized-Loc	3.97	1.44	2.13	1.20	5.07	1.31
Localized-Scale	2.77	1.23	1.88	1.05	5.48	1.19
Reparam-Local	2.15	1.34	1.29	1.13	2.27	1.25
StudentT	1.69	1.36	1.15	1.05	1.87	1.35
Cont. Group Mixture	9.41	–	9.53	–	9.94	–

effective as other models. Unlike other time-series models, the model SE does not model the correlation but describes a regression equation between the past and current observations and thus can benefit more from the robustness transformations.

Mixture models are less robust to outliers than other classes, because they require fitting a large number of parameters, i.e. the locations, scales, and the probabilities of multiple groups. Several robustness transformations could help fit the locations correctly, however, they tend to classify outliers into one of the groups and infer a less accurate scale or probability, which is the case for models MD, MB, and ME. Also, mixture models are more expensive to fit (due to the label switching problem [58]) and thus are likely to diverge when they are not robust to outliers. We observe that at the noise level 10, for the original mixture models, the geomean of the convergence score is 8.94, which is much larger than that of all the original models (2.09). As a result, models like MC and MD can occasionally show a large improvement (up to 27.48x and 6.08x) when the original model diverges due to outliers but the transformed model converges to correct result.

Insight 2. *Overall, we observe the transformations are most useful for most regression models. However, for most time-series models and some classes of regression models, the benefits of transformations are limited since they are already tolerant to input noise. For mixture models, due to the model complexity, transformations generally have less protection against noise, however, they might occasionally protect the original model from divergence.*

Convergence of Transformed Models under Noise. Table 2.4 shows the geometric mean of convergence scores by different transformations averaged by the models with three noise models at noise level 10 when running with ADVI and NUTS.

The convergence score with NUTS is generally better than that with ADVI: for all the transformed programs, the geometric mean of the convergence score for NUTS is 1.33, while for ADVI it is 2.74. We observe that StudentT generally has better average convergence score than Reparam with ADVI. For example, at noise level 10 with Outliers attack, the average convergence score for Reparam is 2.15 while for StudentT it is 1.69 (the lower the

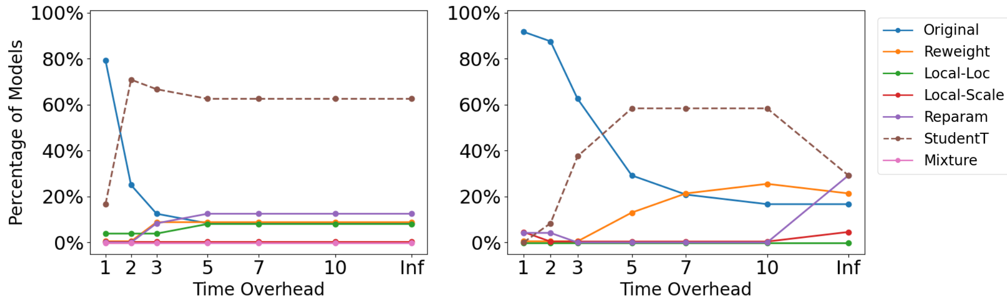


Figure 2.10: (Outliers) ADVI Figure 2.11: (Outliers) NUTS

better). When using NUTS, the heavy-tailed nature of StudentT can make sampling less efficient [45]. Hence, StudentT transformation has *worse* convergence score with NUTS than with ADVI. This also explains why StudentT has slightly higher RIMSE than Reparam when using ADVI, while their RIMSEs are similar using NUTS.

The Local-Loc and Local-Scale transformations introduce a strong dependency between the original parameter and new parameters, as described in [29]. This creates a complex posterior geometry, which is difficult for both algorithms to explore [45]. For instance, for ADVI, at noise level 10, the geometric mean of the convergence score over all models for Local-Loc is 3.69 while for Local-Scale it is 3.18. NUTS does not work well with mixture models (including the Mixture transformation) [58, 59]. For ADVI, *Cont. Mixture* provides best improvements only for two models. Finally, good convergence may not necessarily lead to high accuracy. For example, the Reweighting transformation obtains the best convergence score but only provides the best improvement for four models.

Insight 3. *The performance of a transformation depends on both the inference algorithm and the convergence quality.*

2.7.3 The Overhead of Robustness

Overhead of Transformations for Different Model Categories. Table 2.5 presents the time overhead for different transformations using ADVI and NUTS (over the original program). We divide the benchmarks into three categories: generalized linear models (GLM), Time-Series (TS) and Mixture Models (Mix). The overhead is calculated by dividing the run time of a transformed model by the run time of the original model, and then computing geometric mean over all the benchmarks in the corresponding

Table 2.5: GeoMean Time Overhead

Transforms	ADVI			NUTS		
	GLM	TS	Mix	GLM	TS	Mix
Reweight	2.25x	1.60x	3.13x	6.41x	1.76x	4.88x
Local-Loc	1.73x	1.79x	6.01x	14.75x	3.30x	18.12x
Local-Scale	2.34x	1.81x	6.84x	22.75x	8.00x	30.15x
Reparam	2.47x	1.75x	7.03x	13.05x	3.64x	14.50x
StudentT	1.24x	1.04x	2.21x	6.13x	2.44x	3.15x
Mixture	4.05x	2.70x	–	–	–	–

category. For instance, applying Reweight on GLM is 2.25x times (on average) slower than running the original program with ADVI. NUTS generally has a higher overhead than ADVI. Also, for Mixture Models, the transformations incur the largest overheads among the three model categories, followed by GLM, and Time-Series. The significant increase in execution time for Mixture Models is because the transformations add additional dependency between the parameters in these models, making inference more difficult and slow. On the other hand, since Time-Series Models already have strong dependencies between the parameters, the robust transformations do not affect their execution times much.

Trade-off of Time vs Performance. We evaluate how the choice for best transformation changes (based on posterior predictive accuracy) when the user has limited time budget. For this experiment, we consider different overhead time budgets (from 1x to ∞). For each budget, we filter out transformations that exceed the budget and choose the best transformation among the rest. Figures 2.10 and 2.11 present the results for ADVI and NUTS respectively, for the Outliers noise model. The X-axis represents time budgets. The Y-axis represents percentage of models for which a transformation obtained the best improvement in predictive accuracy. Each line shows the mean across all noise levels for either a transformation or the original model.

For lower time budgets (1-3x), the transformations often produce unacceptable execution overheads, which makes the original model more preferable than the transformed models, especially for NUTS. For ADVI, we observe that StudentT consistently dominates other transformations across all overhead budgets, while Reparam and Reweight assume the second place in most cases and yield best results for similar number of cases. For NUTS, StudentT and Reweight provide better gains than Reparam for overhead budgets of up to 10x. However, for higher budgets, Reparam dominates Reweight and shows closer performance to StudentT.

Insight 4. *Overall, since we observe a larger variance of overheads for NUTS, the users should carefully select a robustness transformation based on the maximum tolerable execution overhead in their applications.*

2.8 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 2.6 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.

For NUTS, the geometric means of the *trajectory divergence* over all the applicable models

Table 2.6: (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 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.

2.9 DISCUSSION

Since *MSE* supports a wide variety of model classes and is easily automated, we only report the evaluation result based on *MSE* in this work. However, our methodology may not tell the entire story about the robustness for all models: (1) it does not take into account the uncertainty in predictions; and (2) it does not use a held-out test set to compute the MSE.

Therefore, after the posterior predictive checks in ASTRA show how well the robustness transformations perform, we suggest ASTRA users to further evaluate the subset of well-performing transformations (semi-automatically) for specific model classes using specialized methods (e.g., predictive on test data, cross-validation, sensitivity analysis, etc.) [47]. These methods can play a complementary role to ASTRA automated analyses.

For example, one may further study the model performance on future observations using the following procedure: if ASTRA shows a poor fit (high *MSE*) using its noise models, then it indicates that prediction of future data is also likely highly inaccurate. If ASTRA shows a good fit (low *MSE*), it means one could also apply other analyses to improve user confidence in the model. In particular, for regression models, one could conduct a cross-validation by splitting the existing data into train/validation/test. For time-series models, one can manually split the data and apply the leave-future out (LFO) cross-validation [60].

We anticipate our work and further automation of applying model robustness transformations and testing for model robustness can lead to future works on 1) general techniques for improving PP robustness, 2) libraries of techniques applicable for specific, but broad, classes of probabilistic models. In addition, we believe that symbolic techniques for robustness analysis and inference (e.g. [31, 61]) can further help improve the reliability of the implementations of robust probabilistic programs. Moreover, since non-convergence issue may

affect the inference accuracy of robust models (also observed in [62]), future studies could more thoroughly investigate the relationship between inference efficiency and robust models.

2.10 RELATED WORK

Robust Probabilistic Modeling. We evaluated various robustness transformations previously proposed in literature [43, 44, 46]. These works did only evaluate small number of programs which makes the generality of their methods unclear. [43] evaluated on six models, and they compared to [44] on a single model. [44] evaluated their method on four models. Also, these works did not the report run time of the robust models. In addition to these approaches, [63] proposed a robust version of KL divergence to make variational inference robust of outliers. Their approach focuses on making the inference more robust instead of the model itself. [64] proposed special measures to improve the robustness of logistic regression models.
Robustness of Neural Networks. Despite their tremendous success in various domains, neural networks are known to be vulnerable to adversarial examples. Researchers have proposed ways to both design attacks for testing the robustness of neural networks [65, 66] and defending against adversarial observations [67, 68, 69]. In this work, we consider multiple attack (or noise) models used previously for probabilistic models. The source of our noise may not necessarily be adversarial but may stem from practical sources such as erroneous measurements, data corruptions, or random failures.

2.11 CONCLUSION

We presented ASTRA – the first system for automatically evaluating the robustness of probabilistic programs against various noise patterns. Our study on 24 benchmarks is the first systematic study of robustness of a diverse set of probabilistic programs. We highlight the benefits of robustness transformations when applied on different models. We show the tradeoffs between the level of robustness and the time overhead for different transformations, which can allow users deploy those robust model versions that best fit their needs.

CHAPTER 3: DEBUGGING CONVERGENCE PROBLEMS IN PROBABILISTIC PROGRAMS VIA PROGRAM REPRESENTATION LEARNING WITH SIXTHSENSE

3.1 INTRODUCTION

Probabilistic programs (PP) express complicated Bayesian models as simple computer programs, used in various domains [70, 71, 72, 73], including the important applications like epidemic modeling [74] and single-cell genomics [75]. Probabilistic languages extend the conventional languages with constructs for sampling from probabilistic distributions (prior), conditioning on data, and probabilistic queries, such as the distribution reshaped by conditioning on the data (posterior) [1]. Probabilistic programming systems (PP systems) compile the programs and compute the results using an efficient inference algorithm, while hiding the intricate details of inference. Most practical inference algorithms are non-deterministic and approximate. For instance, Markov Chain Monte Carlo (MCMC) algorithms [9, 36, 76] run a probabilistic program multiple times (each of which is referred to as an *iteration*) to sample data points from the posterior distribution. They drive today's popular PP systems, such as Stan [8].

MCMC algorithms have a nice theoretical property: in the limit, the samples they generate come from the correct posterior distribution. But, in practice, a user can only execute the algorithm for a finite time budget and hence needs to fine-tune the algorithms to balance between quality of inference and execution time. This complicates development: the programmer needs to write the program in a way that interacts well with the algorithm and select some parameters specific for the inference algorithms. For instance, inference may fail to properly initialize, silently produce inaccurate results, or generate non-independent samples from the posterior distribution. Even identifying and afterward resolving these challenges currently requires significant statistical expertise.

An important property for successful inference is *convergence*, since non-convergence is often a cause of inaccurate (or wrong) result. Convergence means the samples generated by the inference algorithm represent the target distribution. While there exists metrics for convergence (e.g. the Gelman-Rubin diagnostic [47]) in statistic literature, there lacks a comprehensive study of what model features could cause non-convergence. Thus, getting a data-driven understanding of the causes could help developers to debug the non-convergence issues, and does not require expert knowledge. Moreover, the existing convergence diagnostics are *not predictive* – they cannot be determined ahead of time i.e. without running the program. Building prediction model for converges ahead of time would save the time to run programs

(often taking minutes or more). It would also enable a faster program debug/update cycle.

3.1.1 SixthSense

We present SixthSense, the first approach for identifying convergence problems in probabilistic programs ahead-of-run. SixthSense adopts a learning approach: its trains a classifier that can, for a previously unseen probabilistic program and its data, predict whether the program will converge in a specified number of steps (for a given threshold of the Gelman-Rubin diagnostic). The decisions of the classifier are interpretable and can be used to suggest which program features lead to the convergence/non-convergence of the program.

To train such a classifier, SixthSense needs to overcome several challenges that are beyond the big-code techniques studied for conventional languages [77, 78, 79, 80, 81]. First, probabilistic programs are small (20-100 lines of code) compared to conventional programs but their execution is complicated, with conditioning statements for data and non-standard semantics that performs Bayesian inference. Second, due to their relative novelty, there are few publicly-available probabilistic programs that can be used for training. Finally, we should be able to interpret why the programs are predicted to convergence or non-convergence in order to guide developers to debug the non-convergence issues.

Representing Structural, Data, and Runtime Features: To learn a classifier, we embed the syntactic and semantic program features in a numerical vector. To encode program structure, we observe that many snippets of code in probabilistic programs form patterns (sampling from distributions, hierarchical models, relations between variables) that may repeat within the single program or across programs. We identify those patterns as *motifs* – fragments of probabilistic program code, consisting of several adjacent abstract-syntax-tree nodes (e.g., neighboring statements or expressions).

SixthSense learns the set of features from the subset of motifs it identifies in the code. It groups together similar motifs by calculating a low-dimensional representation of the motifs using randomized discrete projections [82]. This way, it can balance the accuracy of prediction and the size of the learned models. We also engineered a set of data features (e.g., means, variances) and the runtime features – diagnostics from early warmup iterations that the inference algorithms compute as they execute. These features cannot be learned by the approaches that focus on static code features [77, 78, 79, 80].

Mutation-Based Program Generation: We present a novel technique based on program and data mutations that produces a diverse set of probabilistic programs with a good balance between converging and non-converging programs, with the goal to augment the training set. Our technique takes a set of seed probabilistic programs as input, analyzes them and applies

a set of pre-defined mutations which aim to change the semantics of generated programs. To obtain better diversity, our algorithm identifies (via locality-sensitive hashing [83]) and discards any mutant that is too similar to the one that was generated before.

Interpretable Predictor Results: For problem diagnosis and debugging of probabilistic programs, it is important to be able to interpret why the algorithm predicted non-convergence. Our learning algorithm leverages random forests for this task. It relates the likely cause of non-convergence to specific statements or expressions in the program code.

3.1.2 Results

In this work, we learn the classifiers for convergence of three popular classes of probabilistic programs: *Regression*, *Time Series*, and *Mixture Models*. We obtained 166 seed probabilistic programs, across the three classes, from an open source repository of Stan programs [50]. For each class, SixthSense generated more than 10,000 mutants with diverse convergence property. We train our classifiers for multiple thresholds of the convergence score (Gelman-Rubin diagnostic) to evaluate the sensitivity of our classifiers.

Our evaluation shows the effectiveness of SixthSense in predicting convergence of probabilistic programs compared to two state-of-the-art learning algorithms for conventional code: Code2Vec [77] and Code2Seq [78]. We measure the prediction quality via *Accuracy* (ratio of sum of True Positives and True Negatives to total tested programs), *Precision* (ratio of True Positives to total classified as Positives) and *Recall* (ratio of True Positives to total actual Positives). Here True Positive is a program that is predicted to converge and it indeed converges; the others are defined analogously.

SixthSense obtains an average Accuracy score across the three model classes of 78% for convergence prediction (with almost equally high precision and recall). SixthSense, with just code features outperforms Code2Vec [77] by 8 percentage points on average and Code2Seq [78] by 5 percentage points on average (for a tight convergence threshold). Moreover, we also show that Accuracy scores increase to over 83% when adding runtime features obtained after just the first 10-200 samples from the warmup stage of the inference algorithm (which is less than 10% of its run-time). SixthSense also has higher precision for all model classes, and recall higher than Code2Vec but similar to Code2Seq. SixthSense's prediction time is less than a second and the model size is modest – less than 20 MB, which is 25-37% smaller than Code2Vec/Code2Seq. We show that the prediction numbers hold across a range of the number of samples the inference would take (between 100 and 1000), and we observe that SixthSense can still achieve a high prediction quality.

We further demonstrate, by studying 40 non-converging programs, that SixthSense can

pinpoint the locations in the code that cause non-convergence for 29 programs (72.5%). In contrast, Stan's runtime warnings point to non-convergence causes in only 5 programs (12.5%).

3.1.3 Contributions

We highlight the main contributions of this chapter:

- ⋆ **SixthSense System[1].** SixthSense is a system for learning to predict convergence of probabilistic programs that aids programmers in pinpointing and understanding the sources of convergence problems in PPs.
- ⋆ **Predicting convergence of probabilistic programs.** We present the first approach for learning predictors for convergence of probabilistic programs based on encoding the structure of probabilistic programs using code motifs.
- ⋆ **Program generation for training set augmentation.** We present a new mutation algorithm for augmenting the training set with PPs that have diverse structural and runtime characteristics.
- ⋆ **Experimental evaluation.** We show that SixthSense predicts convergence for three popular classes of programs, with higher accuracy, precision, and recall than two state-of-the-art approaches (which were originally designed for conventional programs, including tasks like code captioning). In our case study, SixthSense helps pinpoint the likely causes of non-convergence for 29 out of 40 non-converging programs, compared to 5 programs for which Stan's runtime warnings help.

3.2 EXAMPLE

We use a concrete example of a probabilistic program to illustrate how SixthSense works and how we can use it to guide the debugging of probabilistic programs. Figure 3.1 shows two variants of a Mixture model in Stan. A Mixture Model is a probabilistic model that assumes that each observed data point comes from one out of several independent sub-distributions of values. Each sub-distribution has an associated probability (called mixing ratio) of being chosen.

The programs **A** and **B** in Figure 3.1(a), 3.1(b) have several (unknown) parameters: mean *mu* and variance *sigma* of the Normal sub-distribution; *theta* is the mixing ratio of the

[1]SixthSense is publicly available at `https://github.com/uiuc-arc/sixthsense`.

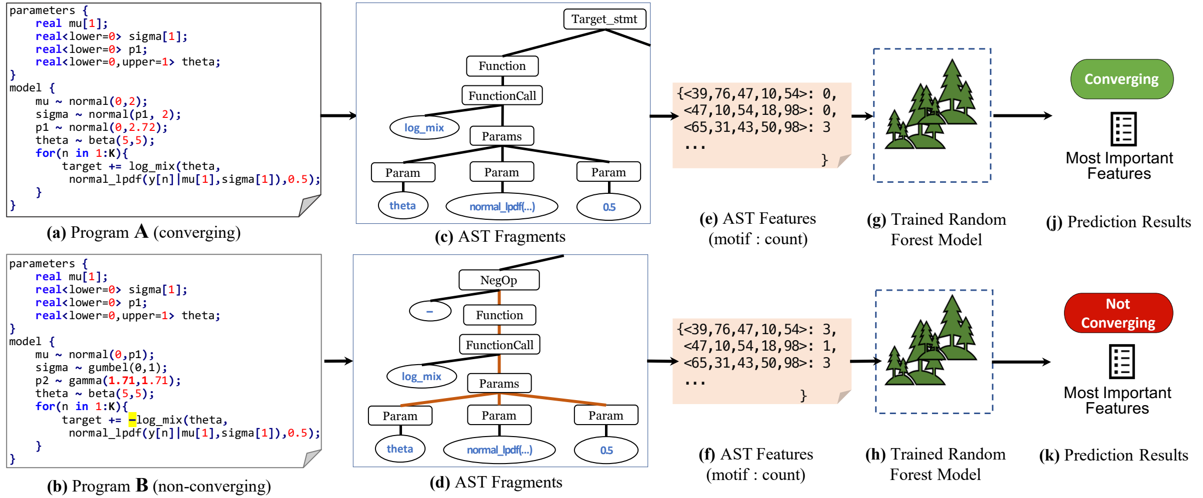


Figure 3.1: An example of two models with different convergence behaviors. We obtain the features from the Abstract Syntax Tree (AST) of source code and data (not shown here). We use them as inputs to the trained Random Forest Model for predicting the label (Converging/Not Converging). We can also obtain the most important features which likely contributed to (non)-convergence.

sub-distributions and *p1* is an auxiliary parameter. The programs also access the array of observations, y, of size K. Each observation in y is assumed to be sampled from one of these two sub-distributions: a Normal distribution (as *normal_ lpdf*) or a uniform distribution (as the constant 0.5). For the program **B**, consider a novice developer, who was confused about Stan's target statement [84], calculated the negative likelihood instead. As an example of how developers typically execute the programs, we invoke Stan with default setting: the NUTS inference algorithm, which employs a 1,000 iterations. In this case, the program **A** converges while the program **B** does not converge.

SixthSense's goal is to predict whether these programs will converge before running the programs, and do so in a small fraction of time. If a program does not converge, the user should know the reason and use the information from SixthSense to debug the program. We next describe how SixthSense computes motifs, trains the predictor, and guides debugging.

Feature Extraction. First, we extract different classes of features for each program in the corpus of mutants. These include *motifs* – representing fragments of the Abstract Syntax Tree (AST), augmented with data features, and run-time features. To extract motifs, we parse each program and construct an AST. Then, starting from each node, we obtain all AST paths of length L by traversing the ancestors of the node. Figures 3.1(c) and 3.1(d) present one sub-tree for the function call statement (in loop) in the programs **A** and **B** respectively and several motifs that SixthSense extracts. The elements in the motif consist of the sequence of node type IDs. The counts of distinct motifs, along with other features

constitute the feature vector of the program. Figure 3.1(e) and 3.1(f) illustrates the motifs (e.g. $\langle 39, 76, 47, 10, 54 \rangle$) and their counts (e.g. $0, 0, 3$).

A good learning algorithm should be able to combine similar motifs and operate only on groups of them. To identify such groups of motifs, we apply *random discrete projections*, a well-known technique for reducing the dimensionality of the feature space. It maps the feature vectors of the node type IDs onto a hash value with a much smaller dimension. The random projections algorithm has a *distance-preserving* property, which means that the similar vectors (even when they are not grouped together) will have similar low-dimensional representations. This property allows us to apply standard learning algorithms on this low-dimensional representation while preserving the similarity of the original motifs.

Computing Reference Solutions and Labels. To compute the program labels ('converging' and 'not-converging'), SixthSense runs them using Stan's MCMC algorithm (NUTS) with the default configuration of 1,000 iterations. For convergence, we calculate a well-known diagnostic called the Gelman-Rubin ($\hat{R}$) statistic [47]. *If the $\hat{R}$ statistic is within a certain bound (close to 1.0), it indicates that the program* converged.

Training. Given a sufficient number of training programs (e.g., an order of $10,000$ programs for each model class), SixthSense extracts the features and obtains the labels for convergence. SixthSense then generates precise and interpretable predictors. We build separate models for predicting convergence for each model class, since models in three classes are significantly different in both semantics and the way they interact with inference algorithms. The model classes are easy to identify manually for users even with minimal expertise or by using simple analytical tools.

Prediction. We use the classifier trained using the batch of Mixture Models for convergence. We use a threshold of 1.05 for the Gelman-Rubin diagnostic (a very tight bound). SixthSense correctly predicts *True* label for program in Figure 3.1(a) and *False* label for program in Figure 3.1(b). The total time required for computing the features and doing the prediction for a single program is less than a second, compared to 53 seconds on average to run a program.

Interpretation and Debugging. Our combination of random projections – which groups very similar motifs together, even if they appear at different locations in the program – and the random forest classification – which has the ability to explain its decisions – proves effective in identifying the parts of the program that impede convergence. Specifically, we can employ SixthSense's random forest classifier to identify top features. When SixthSense predicts non-convergence, the user can debug the program according to the top features.

Now consider the scenario where a novice Stan developer used negative log-likelihood in Stan's target statement, and wrote program **B** (Figure 3.1(b)). SixthSense predicts that **B** does not converge, and gives the topmost feature as the path segment (motif) starting from the negative sign to the parameters in the log-likelihood calculation (function *log_mix*). Figure 3.2 presents this motif. There were three such motifs in program **B** (one for each argument of the *log_mix* function), highly contributing to non-convergence prediction. In contrast, this motif is missing from program **A** (Figure 3.1(a)), and thus has negatively contributed in the converging prediction. This observation validates our earlier intuition about the cause of difference in the nature of two programs and is correctly inferred by our prediction model.

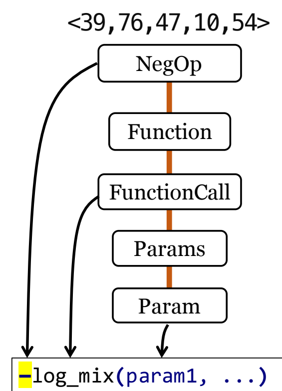


Figure 3.2: Topmost motif in program **B**

It is intuitive for the user to fix a non-converging program (program **B**) by altering the code that corresponds to the top features. In this case, the topmost motif suggests that removing the negative sign would allow program to converge. We describe the debugging process for this example at the end of Section 3.7.2. After applying the change, the user can use SixthSense to predict again, or even iteratively search for a good fix. This iterative debugging would be much faster than running through the full compilation and execution with Stan. At the same time, SixthSense can provide more directed warning messages.

3.3 OVERVIEW

Figure 3.3 shows the architecture of SixthSense. We next describe each of its components.

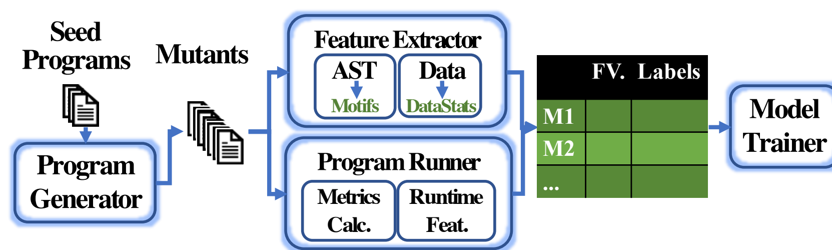


Figure 3.3: SixthSense Training Workflow

Feature Computation. SixthSense's features can be broadly divided into three major groups: (1) automatically-selected AST (Abstract Syntax Tree) based features - motifs -

which represent fragments of the AST; (2) Data Features, and (3) runtime features of the inference algorithm. We present our feature selection and summarization in Section 3.4.
Program Generation. The generator uses the input set of seed probabilistic programs to generate a batch of mutants. We use two sets of transformations to mutate the program: (1) *Expansive Mutations* produce more complex models compared to the original ones (e.g., add a new parameter), and (2) *Reducing Mutations* simplify the models by simplifying arithmetic expressions, removing conditional statements, etc. Our adaptive mutator uses nearest neighbor algorithms to efficiently explore the feature space of the programs. We explain the mutations and the algorithms in Section 3.5.
Program Runner. It runs each generated mutant and collects several statistics such as samples from MCMC iterations and runtimes.
Metric Calculator. Typically, the MCMC algorithms provide samples for each parameter from the posterior distribution. The metric calculator computes the convergence for each parameter using the samples from the posterior.
Model Trainer. Using the syntax, data and runtime features and metrics computed by the previous components, the Model Trainer builds a machine learning model for predicting the behavior of probabilistic models for the given inference algorithm. Here, we used Random Forest Classifier.

We build models to predict, for given metric thresholds, (1) Convergence of the models using static features of model and data, (2) Convergence of the models using static features and run-time diagnostics from initial phases of sampling, and (3) The number of iterations needed for the model to converge.

3.3.1 Convergence

Convergence is an important property for successful inference. In this work, we use the term Convergence to mean "Convergence in distribution". Let us assume each sample is represented by a random variable X_i ($i = 1, 2, \ldots$). We say that a sequence of random variables X_1, X_2, X_3, ... converges in distribution to a random variable X, represented by $X_n \xrightarrow{d} X$, if

$$\lim_{n \to \infty} F_{X_n}(x) = F_X(x), \tag{3.1}$$

for all x at which $F_X(x)$ is continuous. Here, $F_X(x)$ is the cumulative distribution function (CDF) of X.
Gelman-Rubin Diagnostic ($\hat{R}$). The Gelman-Rubin diagnostic analyzes multiple Markov Chains to evaluate the convergence of an MCMC algorithm [47]. The convergence is

computed by comparing the estimated between-chains and within-chain variances for each model parameter. Large differences between these variances indicate non-convergence. Let us assume we have M chains of length N. For a parameter θ, $\{\theta_{mt}\}_{t=1}^{N}$ is the mth simulated chain. Let $\hat{\theta}_m$ and $\hat{\sigma}_m^2$ be the mean and variance of mth chain. Let $\hat{\theta} = (1/M)\sum_{m=1}^{M}\hat{\theta}_m$ be the overall posterior mean. The between-chains and within-chains variances are given by:

$$B = \frac{N}{M-1}\sum_{m=1}^{M}(\hat{\theta}_m - \hat{\theta})^2, \qquad W = \frac{1}{M}\sum_{m=1}^{M}\hat{\sigma}_m^2. \tag{3.2}$$

We can calculate a weight average of B and W: $\hat{V} = \frac{N-1}{N}W + \frac{1}{N}B$. Theoretically, if the distribution of the chains equals the target distribution, or $N \to \infty$, $\hat{V}$ is an unbiased estimator for the marginal posterior variance of θ. Finally, we can compute the Gelman-Rubin diagnostic as follows:

$$\hat{R} = \sqrt{\frac{\hat{V}}{W}} \tag{3.3}$$

In practice, $\hat{R}$ values ≤ 1.1 are considered as good indicators of convergence [47].

3.3.2 Deployment

Once the trainer produces the model, we can use it to predict the convergence of new programs. For a given program and its dataset, SixthSense runs the feature extractor, runs it through the predictor and outputs the convergence label. It also reports on the features that contributed most to the prediction, and relates them back to the source code.

			Type	::=	Int | Float
x	$\in$	*Vars*	Decl	::=	x : Type | x : $[c^+]$
c	$\in$	*Consts* $\cup \{-\infty, \infty\}$	Expr	::=	c | x | Expr *aop* Expr | Expr *bop* Expr
aop	$\in$	$\{+, -, *, /, \hat{}\}$	Stmt	::=	x = Expr | Decl | observe(*Dist*(Expr$^+$),x)
bop	$\in$	$\{=, >, \ldots\}$			| $x \sim$ *Dist*(Expr$^+$) | for $x \in 1..n$;{Stmt*}
Dist	$\in$	{Normal,Uniform,...}			| if (Expr) then Stmt* else Stmt*
ID	$\in$	*String*	Query	::=	posterior(x) | expectation(x)
			Program	::=	Stmt* Query*

Figure 3.4: Syntax of Storm-IR [34]

3.4 LEARNING PROGRAM FEATURES

We present the description of the programs and SixthSense's approach for collecting code, data, and runtime.

Probabilistic Programs Syntax. A probabilistic program is an imperative program with additional constructs for sampling from distributions, conditioning the model on observed data values, and one or more queries for either the posterior distribution or expected value of a parameter. In this work, we use a subset of syntax of Storm-IR [34] for representing probabilistic program, as shown in Figure 3.4.

Representing Program Paths. To understand the causes of non-convergence and for better debuggability, we select a representation that is easy to train and interpret. Existing approaches Code2Vec/Code2Seq [77, 78] aim to predict variable names through natural-language semantics, and they encode the path between any two terminal nodes in the Abstract Syntax Tree (AST). Instead, we encode the sequences of AST nodes with limited length to pinpoint the semantic issues. We formalize our notions:

Definition 3.1. (Abstract Syntax Tree) Similar to [77], we define an AST for a program P as a tuple $\langle N, T, s, \delta \rangle$. N is a set of non-terminal nodes, T is the set of terminal nodes, $s \in N$ is the root node, and $\delta : N \rightarrow (N \cup T)^*$ is a function that maps each non-terminal node to the list of its child nodes, which can be either non-terminal or terminal.

Definition 3.2. (AST Path) An AST path is a path between the nodes in the AST, which starts from one terminal or non-terminal node and ends at another non-terminal node, traversing through the ancestors of each node. We denote an AST path of length L as $\langle N_1, N_2, ..., N_L \rangle$, where $N_i \in \delta(N_{i+1})$, for each $i \in \{1, 2, ..., L-1\}$.

Definition 3.3. (Node-to-Type Mapping) Each node in the AST has an associated node type. We define a function $\tau : (N \cup T) \rightarrow$ Node Types, which maps each node to its corresponding node type. Examples of node types include "Statement", "FunctionCall", "Mulop", etc.

Definition 3.4. (Node Type ID) We define the function ϕ : Node Types $\rightarrow \mathbb{N}$, that maps each distinct node type to a unique identifier, known as the node type ID, represented by a natural number. For example, in Figure 3.1 (c), the function ϕ assigns the same node type ID to the parent nodes of the sub-expressions `theta` and `normal_lpdf(...)`, as they both are instances of the "Param" node type.

Definition 3.5. (Motif) A motif, denoted as m, is a vector that encodes an AST path from a node passing through the ancestors as a numerical vector. Each element in this vector represents the node type ID of a corresponding node in the path. Specifically, for an AST path $\langle N_1, N_2, ..., N_L \rangle$, the motif is a vector $m = \langle I_1, I_2, ..., I_L \rangle$, where $I_i =$

$\phi(\tau(N_i))$, for each index $i \in \{1, 2, ..., L\}$. We refer to the set of all motifs as $\mathbb{M}$.

3.4.1 Extracting Features from Program

Motivation. Two major challenges in efficiently encoding the motifs in a feature vector include (1) the large numbers of different paths that a program may have, and (2) the variability of length between different paths. A general approach to solve both problems is to design a flexible scheme for *dimensionality reduction*, which encodes the rich structures, like our motifs as a smaller set of program properties.

We rest our approach on two observations. First, despite a huge number of possible syntactic paths, *similar motifs repeat often in a single program and across multiple programs*. Therefore, we need to think only about the subsets of all possible paths that appear in the corpus of programs. Second, the variability between motifs is often local, and *many similar (though not-identical) motifs may lead to the same program behaviors*. Therefore, instead of encoding each motif in the feature vector independently, we can group similar motifs and encode only the group.

To reduce the dimensionality of available paths and group together similar motifs, we use *Locality-Sensitive Hashing (LSH)* [85], a hashing technique. While LSH is well-known in data mining for its ability to group similar items with a high probability, it has not been applied to big-code representation. A popular variant of LSH is the *Random Discretized Projections (RDP)* [82]. RDP calculates hash codes that facilitate the grouping of similar items into the same buckets with high probability. These hash codes serve as identifiers for motif groups within the feature vector of the program.

Next we formally define the motif groups and the feature vectors:

Definition 3.6. (Motif Group) Let $\mathbb{M}$ be the set of all motifs, and let $H = [h_1, h_2, \ldots, h_n]$ be a list of n hashing functions. For any motif $m \in \mathbb{M}$, define its hash code as the tuple $c(m) = (h_1(m), h_2(m), \ldots, h_n(m))$. Two motifs $m_1, m_2 \in \mathbb{M}$ have the same hash code, $c(m_1) = c(m_2)$ if and only if $h_i(m_1) = h_i(m_2)$ for every $i \in \{1, 2, \ldots, n\}$.

We then define a motif group, denoted as M, which is a subset of $\mathbb{M}$. A motif group is associated with a specific hash code c_M and contains all motifs in $\mathbb{M}$ that have the hash code c_M:

$$M = \{m \in \mathbb{M} \mid c(m) = c_M\}. \tag{3.4}$$

In our application, RDP utilizes a list of random projection vectors $[r_1, r_2, \ldots, r_n]$, and defines each hashing function $h_i(m)$ to represent the projection of the motif m (which is a numerical vector) onto r_i. This mechanism allows RDP to assign the same hash code to *similar* motifs.

Definition 3.7. (Feature Vector) A program P for a fixed hashing function has a feature vector $\boldsymbol{v} = [v_1, v_2, \ldots, v_k]$. Each element $v_i \in \mathbb{N}$ ($i \in \{1, ..., k\}$) is the sum of counts of the motifs from the motif group M_i in the AST of the program P.

Algorithm for Feature Vector Extraction from a Batch of Programs. Algorithm 3.1 outlines the procedure for extracting the feature vector from a batch of programs. The algorithm processes each program within the batch. Lines 4-8 detail the process of computing the feature vector for an individual program.

In lines 5, the algorithm iterates over the nodes in the AST. At each node, it generates a sequence of nodes by recursively ascending through the parent nodes, up to a depth of L. This process is defined by the function *GetMotifAt* (line 6):

$$GetMotifAt(N, 0) = \varepsilon \tag{3.5}$$

$$GetMotifAt(\varnothing, L) = \varepsilon \tag{3.6}$$

$$GetMotifAt(N, L) = \phi(\tau(N)) :: GetMotifAt(parent(N), L-1) \tag{3.7}$$

Here ε denotes an empty sequence to represent the base cases of the *GetMotifAt* function. The function $\phi(\tau(N))$ retrieves the type ID of the node N, as defined in Definition 3.5. When N is the root node, the $parent(N)$ function returns $\varnothing$, indicating 'no parent' .

The function *RDPSimilarityHash* (line 7) computes a hash key c_M of each motif using the Random Discretized Projections. If the size of the motif is smaller than L (e.g., because the node does not have sufficient number of parents), *PadRight* pads the motif to the maximum size with unused elements. The resulting hash code c_M serves as the unique identifier for the motif group to which the motif belongs (as defined in Definition 3.6). Line 8 is responsible for incrementing the count associated with the motif group each time a similar motif with the same hash code is encountered.

RDP allows a flexible number of projections and the size of bins. These parameters can be tuned to control the granularity of similarity calculations and indirectly impact the size of the feature vector, which will be described shortly.

In line 9, the algorithm assigns the vector $\boldsymbol{v}$ as the feature vector for the program P in the table of feature vectors F. Each row of F corresponds to the feature vector of a single program. Finally, the algorithm returns the table of feature vectors F of all programs in the batch.

3.4.2 Data Features

The nature of the data-set may determine the performance of the probabilistic model when run using an inference algorithm. For instance, in absence of sufficient data, the choice of prior distributions become very important. Similarly, a strong prior with very small variance is unlikely to converge to the correct results in such a scenario [86]. SixthSense computes data metrics like *sparsity* (number of non-zero elements), *auto-correlation* (correlation between values of a time series), *skewness* (asymmetry of the distribution), maximum/minimum variances of the model's prior distributions, and several others for observed and predictor data variables.

Algorithm 3.1 Compute Feature Vectors

Input: Batch of Programs *Batch*, Motif depth L

Output: Table of Feature Vectors F

```
1: procedure CalculateFeatures
2:     F ← ∅
3:     for P ∈ Batch do
4:         v = [0, ..., 0]
5:         for node ∈ nodes(AST) do
6:             m ← GetMotifAt(node, L)
7:             c_M ← RDPSimilarityHash(PadRight(m, L))
8:             v[c_M] ← v[c_M] + 1
9:         F(P) ← v
10: return F
```

3.4.3 Runtime Features

For inference algorithms like MCMC, diagnostics from the early stages (warmup) of sampling can often indicate the presence or absence of problems with the model and associated data. Such diagnostics can help in discovering problems earlier so that the users can update their model for more efficient performance. Unfortunately, they are not predictive in nature: manually observing the raw values may not provide a good intuition about the program execution. However, our prediction engine can infer useful information from them.

To validate this intuition, we collect several runtime features from MCMC chains during the early stages of warmup iterations:

- **Posterior Log Density**: Computes the log probability that the data is produced by the model using current set of the parameters
- **Tree Depth**: Tree Depth for the NUTS algorithm
- **Divergence**: Measures the divergence of the simulation from true trajectory.
- **Acceptance Rate**: Acceptance Rate of generated samples
- **Step-size**: Determines the distance between consecutive samples
- **Leapfrog steps**: Number of steps to take for the next sample

- **Energy**: Potential energy of the Hamiltonian Particle

More details about these measures can be found at [87].

3.5 PROGRAM GENERATION FOR TRAINING SET AUGMENTATION

In this section, we describe our approach of generating mutant programs from a corpus of seed probabilistic programs. To produce mutants from the original seed probabilistic programs, we define two kinds of transformations – for code and data.

3.5.1 Code Mutations

Our Code Mutations can be broadly classified into two sets: (1) *expansive mutations*, which make more complicated models from the original one, and (2) *reducing mutations*, which reduce the complexity of the models.
Expansive Mutations. We apply the following mutations:

- **Auxiliary Parameter Creator.** This mutation replaces any distribution and function argument with a parameter and assigns a prior distribution to the parameter. For instance, given a statement of the form $x = normal(a, b)$, this mutation can expand the single statement to a chain of statements: $\sigma = gamma(c_1, c_2); p = normal(a, \sigma); x = normal(p, b)$. Here, a and b can be any expressions, c_1 and c_2 are appropriate constants. We perform interval and dimensional analysis to find the valid set of distributions and values (i.e. not limited to Gamma and Normal distributions).
- **Constant Replacer.** This mutation lifts a constant in the program to a parameter with an appropriate prior distribution. For instance, a constant 0.5 is transformed to parameter sampling from beta distribution (it samples from $[0, 1]$).
- **Dimension Expander.** This mutation expands the dimension of a scalar parameter to match the dimension of any vector expression it is used in. For instance, for a statement, $y = a + b$, where y and b are vectors and a is a scalar parameter, Dimension Expander changes the dimension of a to match the dimension of y and b. We use dimensional and type analysis to ensure the mutation is valid and does not lead to compile time failures.
- **Data to Parameter Transformer.** This mutation randomly replaces a real valued data array with a parameter with the same dimension. It also assigns appropriate prior distribution to the parameter based on range of data values.

Reducing Mutations. The reducing transformations include the following:

- **Arithmetic Simplifier**: this transformation replaces arithmetic expressions with either of the operands or changes the arithmetic operation.
- **Conditional Eliminator**: it replaces conditional statements with either of the branches.
- **Distribution Simplifier**: it replaces complex distributions like Laplace and Weibull with common distributions like Normal or Uniform.
- **Math-Function Call Eliminator**: it replaces common math functions such as *log* and *exp* with constants.
- **Conjugate Replacer**: In probability theory, if the prior and posterior distributions belong to the same probability distribution family, they are referred to as conjugate distributions [88]. The prior distribution is said to be conjugate to the posterior (or likelihood) distribution. This property is particularly important for inference algorithms because the presence of conjugacy between the prior and likelihood distribution simplifies sampling from the posterior distribution, as it can be easily factorized. To explore this property, we consider replacing prior distributions with distributions conjugate to the likelihood when possible.

The first four kinds of transformations have been previously used by Storm [34] for testing PP systems. We added the conjugate replacer as it can simplify the inference.

3.5.2 Data Mutations

Apart from source code transformations, we also added several data transformations. Such transformations help in changing the distribution of values in the data set, which could produce challenging scenarios for the probabilistic model or inference algorithm to work with. The data mutations include scaling by a constant, adding arbitrary noise, Box-Cox transformation [89], scaling to new mean and standard deviation, cube root transform, and random replacement of values with values from the same data set.

3.5.3 Adaptive Algorithm for Mutant Generation

To generate programs with different runtime behaviors, it is essential to explore programs with diverse semantic and syntactic features. Our mutation algorithm achieves this by randomly introducing several mutations to the original program. One prominent approach to ensuring such diversity is through the techniques called Locality Sensitive Hashing (LSH) [85], which is designed to group similar feature vectors with high probability. One of the most popular versions of LSH is the Random Discretized Projections (RDP), which

was also introduced in Definition 3.6 (Section 3.4.1) for motifs. In this section we repurpose RDP for feature vectors which are also numerical vectors like motifs. Utilizing RDP allows us to identify and exclude mutants with highly similar feature vectors. Here we generate the hash code for a feature vector analogously to the motif's hash code, using RDP as outlined in Definition 3.7. Specifically, we consider two feature vectors v_1 and v_2 as *neighbors* if they have the same hash code in RDP. To ensure diversity among generated mutants, we keep only one mutant for each unique hash code.

Algorithm 3.2 Selecting Mutants

```
Input: Seed Programs S, Programs K, Batch-Size B
Output: Program Set progs
procedure SelectMutants
    lsh ← InitializeLSH()
    progs ← ∅
    while | progs |< K do
        for s ∈ S do
            seed ← ChooseSeed(s, progs)
            cand ← GeneratePrograms(seed, B)
            for P ∈ cand do
                v ← lsh.GetFeatureVector(P)
                if lsh.Neighbors(v) = ∅ then
                    lsh.StoreVector(v, P)
                    progs ← progs ∪ {P}
    return progs
```

Algorithm 3.3 Generating Mutants

```
Input: Seed program S, Programs K, Max Changes C
Output: Program Set progs
procedure GeneratePrograms
    progs ← ∅
    i ← 0
    while i < K do
        P′ ← P
        for t ∈ {1..C} do
            m ← chooseMutation()
            P′ ← m.mutate(P′)
        if P′ ≠ P then
            progs ← progs.append(P’)
        i ← i + 1
    return progs
```

Algorithm 3.2 presents the mutant selection algorithm. The inputs for the algorithm are seed probabilistic programs S, the total number K of programs to generate, and the total number B of programs to generate in each batch from each seed program. The algorithm returns the selected mutant programs set *progs* as output. First, we initialize the LSH (Locality Sensitive Hashing) engine using Random Discrete Projections (RDP) hash functions.

In each round, we first choose a seed program $s \in S$ using the *ChooseSeed* function. The *ChooseSeed* function randomly chooses among the original seed program s and the mutants generated in earlier rounds (stored in *progs*). Next, we generate a new batch of programs of size B using *GeneratePrograms*.

For each newly generated program P, we compute its feature vector using *lsh.getFeatureVector* and the number of neighbors among the already generated programs using *lsh.Neighbors*. We select the program only if it has no neighbors in the already selected set of programs. Finally, the algorithm returns the selected set of programs once it has generated the target K programs.

Furthermore, Algorithm 3.3 defines the *GeneratePrograms* algorithm, which is used as a

subroutine of Algorithm 3.2. *GeneratePrograms* is responsible for generating K mutants for a seed program S. For each program, in each iteration, it applies a set of randomly chosen mutations and adds it to the set of new programs. Finally, it returns the set of new programs to the caller. Using this algorithm, we can obtain a diverse set of probabilistic programs with a balance of converging/non-converging behavior.

Generating Semantically Valid Mutants

Maintaining semantic validity of generated programs ensures that we do not generate programs which compile but fail trivially due to semantic errors (e.g. incorrect array indices, dimension mismatches, illegal distribution/function inputs like indefinite matrices). Otherwise, the prediction task becomes simpler and less useful for predicting properties of real programs–which have more complex structure.

To ensure the semantic validity of mutants, we implement several analysis techniques which incorporate domain knowledge. For instance, the *multi_normal* distribution has a co-variance matrix parameter. The constraint on this parameter is that it needs to be positive definite. Otherwise, sampling from the distribution leads to several runtime errors which consequently leads to erroneous samples during inference. To prevent this we use data-flow analysis to identify key computations affecting the covariance matrix in the program and avoid applying mutations to those.

Extension: Generating Harder Benchmarks

We explore an interesting scenario where we bias our mutant generation algorithm to generate harder benchmarks i.e. programs that do not converge or produce inaccurate results. Such programs can serve as useful benchmarks for any developer who wants to evaluate their inference algorithm and test its performance. Next, we outline a heuristic for our approach.

We create a profile of all the mutations by assigning scores to each mutation based on the runtime behavior of the previous batch of programs. Algorithm 3.4 takes the metric label (for convergence or accuracy) for each mutant program in previous batch and the set of mutations *MC* used on each mutant. It initializes all scores with zero. For each mutant program, it decreases the score proportionally if the program did not converge (or was inaccurate) and vice-versa. Finally, it returns the set of scores for the mutations. Given such a mutation profile, we can easily tune *GeneratePrograms* algorithm to choose mutations by assigning higher weights to mutations with more negative scores or higher tendency to produce harder

benchmarks (non-converging or inaccurate). Another possible approach would be to use our learned predictors for convergence and accuracy to select the likely non-converging and/or inaccurate mutants for new batch.

3.6 METHODOLOGY

We present the methodology for collecting seed probabilistic programs and the program features and metrics we compute.

Seed Probabilistic Programs. We collected a corpus of probabilistic programs from the most comprehensive open-source repository of Stan Models [50][2]. Out of total 505 probabilistic programs, we selected the three most common categories: Regression (120 programs), Time-Series (23), and Mixture Models (23, augmented with 3 from [90]). These probabilistic programs come with their datasets.

Algorithm 3.4 Mutation Profiling Algorithm

```
Input: Metrics Met, Mutation Combinations MC
Output: Mutation Profile scores

procedure CREATE MUTATION PROFILE
    scores ← [0, 0, ..., 0]
    for i ∈ {1, ..., Met.length} do
        mt ← Met[i]
        for mut ∈ MC[i] do
            if mt = False then
                scores[mut] ← scores[mut] − 1/MC[i].length
            else
                scores[mut] ← scores[mut] + 1/MC[i].length
return scores
```

Inference Engine and Sampling. NUTS, the default inference engine of Stan [91]. We executed all probabilistic programs using 4 MCMC chains with 1000 iterations each for warmup phase and sampling. This configuration is default for Stan. We also checked the eventual convergence by running the programs for many more iterations. We used 100,000 as the maximum iteration number (the convergence metrics do not change significantly even for 10^6 iterations for the seed probabilistic programs).

Feature Extraction. We used a Python based implementation of Randomized Discretized Projection (RDP) [92]. We configured the hyper-parameters of the RDP algorithm as P=5 and bin-width B=5, which worked well to reduce the dimensionality of the vector space.

Random Forests. We used Random Forests Classifier from Scikit-Learn package in Python

[2] The number of publicly available probabilistic programs in public sources is low compared to conventional languages. This is in part due to the novelty of these languages and expertise required to create and interpret those programs. As a further challenge, probabilistic program systems like Stan require not only a well-defined program, representing the statistical model, but also a corresponding dataset to fit the model to. Unfortunately, many Stan programs on GitHub lack this essential dataset. Furthermore, most of the publicly available programs are tuned to converge to their available datasets.

for training. We use 5-fold cross validation for training. We extract top features using TreeInterpreter [93].

Execution Setup. We performed the mutant generation and feature computation on an Intel Xeon 3.6 GHz machine with 6 cores and 32 GB RAM. We used Azure Batch Scheduling Service to run all the programs and metrics computations. We capped the MCMC execution under 240 minutes.

3.6.1 Baselines

We compare SixthSense to three baselines: The first, **Code2Vec** [77], and the second, **Code2Seq** [78], are state-of-the-art predictors based on Deep Neural Networks for big-code. They were originally used to predict function names from code. We adapted these systems to do classification for each threshold of convergence, by extracting *path contexts* (subsets of paths similar to our motifs) form the code. The third baseline, the **majority label classifier**, assigns the most likely label observed on the training set to all programs during prediction. Since it consistently (and naively) uses the same majority label of the training set as its prediction, it helps indicate the prediction "hardness" when the training set is imbalanced.

3.6.2 Metrics

We describe various metrics that we use in our evaluation.

Accuracy. Accuracy is a classification metric defined as the ratio of correct predictions (the sum of True Positives and True Negatives) to the total number of tested programs:

$$\textit{Accuracy}\text{ score} = \frac{\text{True Positives} + \text{True Negatives}}{\text{Total Num. of Tested Programs}} \tag{3.8}$$

Higher accuracy values, closer to 1, indicate better model performance, with 1 denoting perfect classification and 0 indicating none are classified correctly. Accuracy is well-suited for balanced datasets where the number of converging and non-converging programs in the test set is approximately equal.

F1 score. The F1 score measures a different aspect of classification performance. It is the

harmonic mean of Precision and Recall:

$$Precision = \frac{\text{True Positives}}{\text{True Positives} + \text{False Positives}} \tag{3.9}$$

$$Recall = \frac{\text{True Positives}}{\text{True Positives} + \text{False Negatives}} \tag{3.10}$$

$$F1 \text{ score} = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}} \tag{3.11}$$

Intuitively, Recall (also True Positive Rate) measures the proportion of positive instances correctly identified by a classification model amongst all positive instances; while Precision measures the proportion of correctly identified positive instances amongst all instances classified as positive. Both measures are the better when closer to 1, but there is a trade-off between Precision and Recall. For training and cross-validation, we use F1 score as the optimization metric.

AUC. AUC [94] represents the area under the receiver operating characteristic curve (ROC). The ROC curve is a plot of Recall (True Positive Rate) against the Fallout (False Positive Rate) for *all* the thresholds. The threshold refers to the cutoff value used to interpret a predicted score (ranging from 0 to 1) as positive or negative labels.

$$Fallout = \frac{\text{False Positives}}{\text{False Positives} + \text{True Positives}}. \tag{3.12}$$

One point on the ROC curve represents one pair of Recall(t) and Fallout(t) for a specific threshold t. One can calculate AUC as

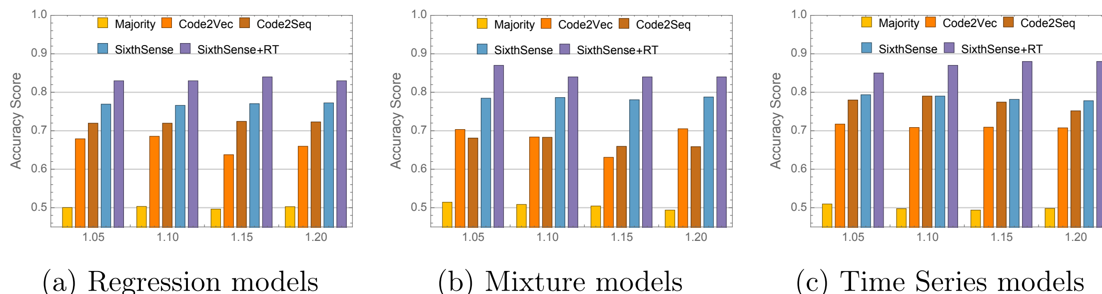
$$AUC = \int Recall(t)\ d(Fallout(t)). \tag{3.13}$$

Ideally, we want the Fallout to be low (close to 0) and Recall to be high (close to 1). AUC presents the trade-off between Recall and Fallout, which the F1 score does not directly quantify. AUC can be interpreted as the probability that our classifier ranks a positive case higher than a negative case. The value of AUC ranges from 0 to 1. A random classifier for balanced data will result in AUC= 0.5. Unlike the Accuracy Score, the AUC score is useful for measuring the performance of classifiers with imbalanced data [95].

3.6.3 Evaluation Experimental Setup

Training and Test Sets. We generate a corpus of mutants programs for each seed probabilistic program using the approach discussed in Section 3.5.3. We create a *test-train split* for every seed program in the following way: (1) *Test set* consists of a single seed program and *all* its mutants; (2) *Training set* contains all other seeds and mutants. Thus, the training is not aware of any mutants of the test seed program. For each such split, we train a classifier using the training set and evaluate its performance (using the metrics below) on the test set. With this strategy we obtain metrics for each split (each representing one seed program and its mutants). Finally, we compute the *average* performance across the splits.

Training a predictor by leaving out each probabilistic program and its mutants in test set allows us to stress-test the predictor. We choose this evaluation strategy because the number of original seed probabilistic programs in each class is low compared to conventional big-code datasets. Every seed probabilistic program represents a different statistical model and using this strategy helps us evaluate the sensitivity of the classifiers for each such model.
Classification Scores. We used Precision, Recall, F1, Accuracy, and AUC [94] to evaluate the performance of the learned classifier. They range between 0 and 1 (higher better). We use the same metric for all the baselines. Specifically, we report Accuracy for the scenario when the test set has balanced labels, and when dealing with potential imbalanced labels, we report Precision and Recall, or F1 score, in conjunction with AUC to provide a comprehensive evaluation of the classification performance.



(a) Regression models (b) Mixture models (c) Time Series models

Figure 3.5: SixthSense Prediction Accuracy for Convergence (Measured Using the Gelman-Rubin Diagnostic)

3.7 EVALUATION

3.7.1 Predicting Convergence of Inference

Figure 3.5 presents the prediction scores for SixthSense when predicting convergence of

MCMC algorithms (NUTS in this case). The Y-axis shows the accuracy scores for each prediction model (higher is better). The X-axis shows the four thresholds (1.05-1.2) of the convergence metric, Gelman-Rubin diagnostic, that we considered in our evaluation. We chose this range to test how general the prediction can be as the individual program labels change. For each threshold, we plot the accuracy scores of our prediction model (SixthSense) together with Code2Vec, Code2Seq and a Majority Label Classifier, as vertical bars in different colors. We evaluated the trained model on a held-out test set (see Section 3.6.3).

Comparison with Code2Vec/Code2Seq. Figure 3.5 shows that SixthSense, with solely AST motifs is better than Code2Vec and Code2Seq (see also the ablation study in Section 3.8). The results show that SixthSense's learned classifiers have an accuracy score close to 0.8. These prediction rates are already useful for the user because it helps them avoid wasting time for compiling and running programs which would likely not converge. Our training algorithm is able to learn classifiers that generalize well across different thresholds.

For Regression and Mixture model programs, SixthSense has consistently better accuracy than the other approaches across all thresholds. For the tightest convergence bound $\hat{R} = 1.05$, its accuracy is by 5 percentage points higher than the alternatives for Regression, and 8 percentage points higher for Mixture. For Time Series programs, the accuracy scores of SixthSense is by 1 percentage point higher than Code2Seq.

Table 3.1: Precision (**P**) and recall (**R**) ($\hat{R}$=1.05)

Class	**6s-AST**		**Code2Vec**		**Code2Seq**	
	P	**R**	**P**	**R**	**P**	**R**
Regression	0.71	0.71	0.63	0.69	0.66	0.72
Mixture	0.77	0.74	0.67	0.67	0.67	0.72
TimeSeries	0.79	0.75	0.69	0.74	0.74	0.77

Table 3.2: AUC scores ($\hat{R}$=1.05)

Class	**6s**	**6s+RT**	**Code2Vec**
Regression	0.82	0.88	0.73
Mixture	0.84	0.90	0.74
TimeSeries	0.86	0.89	0.79

Table 3.1 presents the precision and recall for $\hat{R} = 1.05$. SixthSense exhibits consistently higher precision over Code2Vec (8 to 10 percentage points) and Code2Seq (5 to 10 percentage points). SixthSense also has higher recall than Code2Vec (1 to 7 percentage points), while the recalls of SixthSense and Code2Seq are comparable (within 2 percentage points). Recall that the precision/recall are averaged over those for different splits and can be more sensitive to small and unbalanced splits.

Table 3.2 shows the AUC scores for SixthSense, SixthSense with runtime features and Code2Vec. Code2Seq does not provide its probability of predictions, which prevents us from computing its AUC score. The results show that SixthSense improves in AUC score over Code2Vec for all classes.

The prediction accuracy, prediction, and recall from Tables 3.1 and 3.2 persist for higher thresholds of $\hat{R}$.

Comparison to Majority Label Classifier. Figure 3.5 shows the comparison of SixthSense to a naive Majority Label Classifier, which has the classification accuracy of 0.5. It indicates the significant level of improvement of SixthSense over the uninformed random choice.

Predicting with Warm-up Runtime Features. Figure 3.5 presents the impact of SixthSense's AST features augmented with runtime features (Section 3.4.3) sampled from the first 200 iterations of the warmup stage (at this point Stan still does not issue warnings for our programs). Recall, the results of these iterations are dropped by the inference algorithm, as in this phase the mixing of the MCMC chains has just begun. However they can be useful in addition to code features: they help improve the prediction by further 6 percentage points for Regression and Timeseries, and 8 percentage points for Mixture models ($\hat{R} = 1.05$).Table 3.2 also shows the improvement in AUC of both AST and Run-Time features over the AST-only version of SixthSense. However, note that collecting run-time features still requires compiling the program and starting its execution. While this time differs among the systems and datasets, it may be non-trivial, as is the case for Stan (e.g. around 30 seconds for compilation). This time may be an important factor when deciding to use a runtime-predictor for different PP systems. We also present a feature ablation study in Section 3.8.

3.7.2 Debugging Non-Converging Programs

Table 3.3: Debugging Non-Converging Programs

Class	**#M.**	**6s Upd.**	**Stan Warn.**	**Stan Upd.**
Regression	14	11	4	2
Mixture	13	9	4	1
TimeSeries	13	9	4	2

When SixthSense's learned model predicts that a probabilistic program will not converge, two natural follow-ups are (1) ask which part of the program is likely culprit for non-convergence and (2) how many iterations would be sufficient to run the program to converge, if it converges.

Debugging Approach. We interpret the outcomes SixthSense predicts, and leverage the AST features and the random forests to help pinpoint which part of the program leads to non-convergence.

To obtain the set of programs, we randomly selected 40 probabilistic programs from our *test sets*, equally across the three model classes, which SixthSense correctly identified as non-converging for 1000 iterations. For each program, we obtained the most important features from the learned random forest. We selected *top-5 features* (motifs) and inspected the probabilistic program to identify whether the parts of the motifs contains the culprit of non-convergence. The top-5 features typically only cover 5% of all the motifs, which means SixthSense points to a relatively small scope to debug.

We make up to two manual updates to each probabilistic program by making changes only to the AST elements identified by the motifs or the referenced observed data. These changes represent simple semantic modifications that a user of probabilistic program might make as they explore various possible models for their data. We simulate a *try and check* interactive search with these localized transformations. For instance, SixthSense identified a constant array in a regression equation as one of the top motifs. Converting that constant into a parameter made the probabilistic program converge. Some of our attempted updates include changing the variance (constant) of a distribution, changing the distribution for a parameter, changing a parameter to a constant, and removing mathematical functions (e,g. *abs, log*) when they are redundent.

After transforming the probabilistic program, we run inference to see if it converges. We further check if the probabilistic program become accurate (or correct) after the fix, since non-convergence often causes inaccurate (or wrong) result. For each probabilistic program, we apply accuracy tests from Bayesian model checking [47, Ch.6]: we compute the mean squared error to compare the new result from the probabilistic program to its correct data and also do visual inspection on the result density plot to check if it matches the correct distribution. Multiple student authors inspected the updates and agreed that these changes followed the protocol described above.

Results. Table 3.3 presents the results for this debugging application. Column 1 (**Class**) presents the classes of randomly sampled probabilistic programs. Column 2 (#**M.**) presents the number of mutant programs we randomly selected from each class. Column 3 (**6s Upd.**) presents the number of programs that we manually updated to converge using the method above. Column 4 (**Stan Warn.**) presents the number of programs which Stan issued a warning during sampling. Column 5 (**Stan Upd.**) presents the number of programs for which Stan's warnings helped update the program to converge.

Overall, we were able to identify the problem and let 29 updated programs converge out

of 40 programs. Specifically, we corrected 16 programs by replacing a parameter indicated by SixthSense with a constant; corrected 6 by simplifying mathematical functions, 3 by changing constants in distributions, 2 by converting constants to parameters, and 2 by changing distributions for parameters. All the code elements we changed were pointed by top three motifs SixthSense returned. For 11 programs that we were not able to update, we believe that the programs correction would require more complex changes than those we specified in setup above.

Out of 29 updated, now converging programs, we ran SixthSense again. It correctly predicted that 21 will converge (with 8 from Regression, 8 from Time Series and 5 from Mixture); this is, interestingly, close to the prediction rates from Section 3.7.1. This illustrates that SixthSense can be useful in the iterative debugging loop.

These results demonstrate the advantage of interpretability SixthSense's learned model. Using *motifs* from the AST as features and a simple learning model (random forests) helps the user easily identify key program components which affect the runtime behavior of a probabilistic program. In comparison, identifying such important features is hard for other complex neural network-based models and might require more low-level handling of the learned model. In particular, Code2Vec and Code2Seq do not provide a way to interpret how their prediction worked.

Comparison to Stan's runtime warnings. Compared to Stan's runtime warnings, SixthSense motifs reveal more fine-grained patterns that hinder convergence. For most of the non-converging programs (29 out of the 40 in this experiment), Stan did not issue a warning (beyond the low $\hat{R}$ value at the end of inference) The 12 warnings issued by Stan only have regards to function domains. Seven out of 12 were not related to non-convergence. For instance, one program returns "*Warning: normal_lpdf: Scale parameter is -0.0799029, but must be* > 0." Changing the scale parameter limits does not help. Instead SixthSense identifies the fix that is not at this location.

The remaining 5 Stan runs indicate non-convergence and can help with updating the program. However, they were not as helpful in locating the causes as SixthSense. One example where both SixthSense and Stan indicated problem is in the program with the expression $normal(exp(w0) + sqrt(abs(w1)) * x1 + w2 * x2, s)$. Stan warned about the overflow in the first argument of *normal*, disregarding its sub-expressions. SixthSense traced the problem to the *sqrt* and *abs* sub-expressions that indeed helped fix the non-convergence, by simplifying the function expressions.

Example: Practical Guide to Debugging Probabilistic Programs with SixthSense. SixthSense provides a practical approach for users to diagnose and fix convergence issues effectively. Here we work through the debugging process using the example from Sec-

tion 3.2. In this example, a novice Stan developer has created a probabilistic program labeled **B** (Figure 3.1(b)). SixthSense predicts that program **B** does not converge and outputs the top five motifs contributing to its prediction.

The top motifs and their corresponding AST types for program **B** are shown in Table 3.4. Different from Figure 3.1(c) where motifs are numerical vectors symbolizing SixthSense's internal representation, SixthSense eventually prints its finding to users via the list of AST types (shown in the second column of Figure 3.4). Recall that motifs are represented by ascending through parent nodes, therefore, the lists of AST types provided to the users are in the reverse order of the motifs to facilitated easier interpretation.

Table 3.4: Top Motifs for Program **B**

Motif	List of AST Node Types
$\langle 39, 76, 47, 10, 54 \rangle$	"NegOp-Function-FunctionCall-Params-Param"
$\langle 47, 10, 54, 18, 98 \rangle$	"Stmt-Target-NegOp-Function-FunctionCall"
$\langle 65, 31, 43, 50, 98 \rangle$	"Stmt-Prior-Limits-Value-Int"
$\langle 31, 91, 39, 76, 40 \rangle$	"Distr-Params-Param-MulOp-Value"
$\langle 10, 54, 18, 78, 55 \rangle$	"Block-Stmt-Target-NegOp-FunctionCall"

Note that three out of these five motifs (the first, second and fifth motifs) share the "NegOp" node (encoded as "54"), suggesting it may be a primary cause of the non-convergence issue. Given this information, the user may consider removing the negative sign or altering it. After applying this change, before running the full execution of the program (which may take minutes and may not give an accurate result if the program does not converge), the user can use SixthSense again to predict convergence. In this example, after removing the negative sign, SixthSense's subsequent prediction is convergence. The user will notice that SixthSense no longer highlights any path segments with NegOp, indicating that removing NegOp indeed solves the issue.

Furthermore, the third and fourth motifs provide valuable insights. The motif "Statement-Prior-Limits-Val-Int" highlights that too many prior distributions limited within a truncated domain may cause non-convergence. Meanwhile, the motif "DistrExpr-Params-Param-MulOp-Val" suggests that sophisticated arithmetic operations in the prior can impede smooth sampling and lead to non-convergence. Although these might not be the main problem here – fixing them could entail big changes to the model, like removing several latent parameters or changing the arithmetic structure of the model – they are still constructive indicators for potential issue sources.

Since different motifs can indicate various potential modifications to the program, SixthSense users are encouraged to consider which changes offer the greatest potential in resolving the convergence issue while preserving the model's integrity. If SixthSense does not predict

convergence after a fix, the user can also iteratively explore alternative fixes suggested by the other top motifs until SixthSense predicts convergence. As an illustrative example, if a user observes that the topmost motif includes the FunctionCall (`log_mix`), the user may choose to replace `log_mix` with a semantically equivalent if-then-else statement since it does not alter the underlying model at all. However, after such a change, SixthSense still predicts non-convergence for the modified program which indicates to the user that further changes are needed.

3.7.3 Prediction For Different Iteration Counts

Table 3.5: F1 Scores for Different Iterations ($\hat{R}$=1.05)

Class \ Iterations	100	400	600	800
Regression	0.76	0.75	0.75	0.75
Timeseries	0.75	0.78	0.77	0.81
Mixture	0.75	0.74	0.74	0.74

Table 3.6: AUC Scores for Different Iterations($\hat{R}$=1.05)

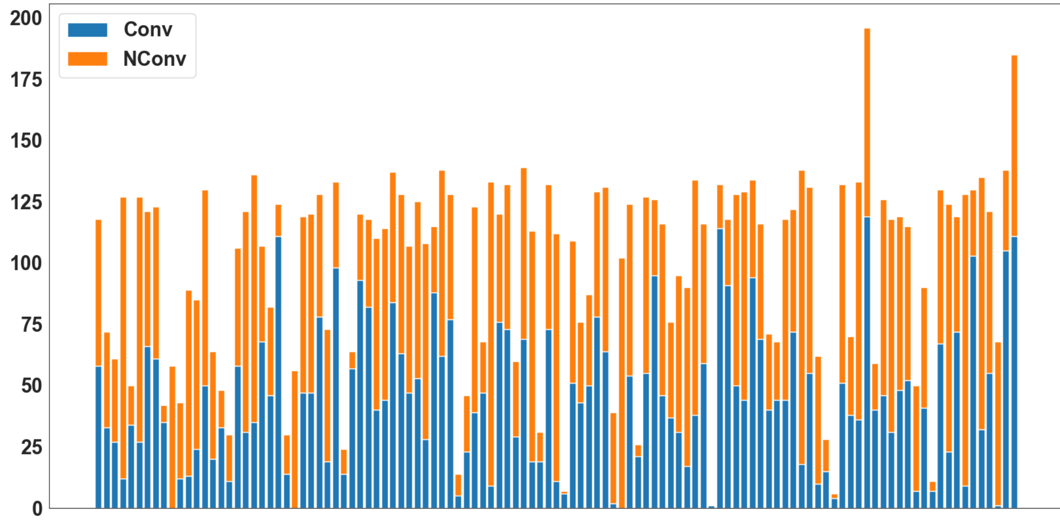
Class \ Iterations	100	400	600	800
Regression	0.78	0.80	0.79	0.80
TimeSeries	0.76	0.78	0.79	0.78
Mixture	0.75	0.74	0.74	0.75

We explore the generality of SixthSense by conducting predictions when the number of MCMC iterations is fewer than the original 1000. Table 3.5 summarizes the results. Each cell presents the F1 score (averaged over 3 runs) for different number of iterations. In this experiment we fixed the Gelman-Rubin diagnostic threshold to 1.05. Table 3.6 presents the AUC scores for the same experiment.

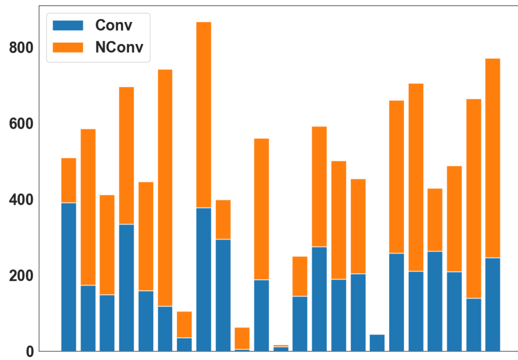
The results shows that even for different distributions of positive and negative labels, SixthSense performs well. Although the labels of individual probabilistic programs change, sometimes significantly (e.g., for 100 iterations), the F1 scores remain consistently high. For AUC we observe a similar trend.

3.7.4 Characteristics of Generated Mutants

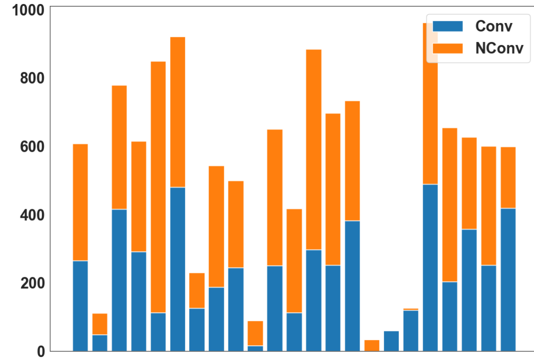
Figures 3.6a, 3.6c, 3.6b show the distribution of converging/non-converging mutants in the program set generated by our mutation algorithm (Algorithm 3.2), after running 1000 iterations. Y-Axis presents the total number of mutants. Each bar gives the count of mutants



(a) Regression ($\hat{R} = 1.05$)



(b) Mixture ($\hat{R} = 1.05$)



(c) Time Series ($\hat{R} = 1.05$)

Figure 3.6: Distribution of Converging Mutants for All Seeds. Blue programs are converging, orange are non-converging.

Table 3.7: Time Taken by Different Phases in SixthSense

Model Class	Programs	Mutants	Mutant Gen.	Code Run	Feature Calc.	Training	Test
Regression	121	8960	2h5m	23h57m	1h5m	23m	0.21s
Mixture Models	24	8524	2h	48h2m	1h42m	20m	0.22s
Time Series	27	8703	1h31m	39h33m	1h16m	21m	0.22s

from each seed probabilistic program, marking converging and non-converging separately. In total, we generated 30,000 mutants, with around 10,000 mutants per class.

These plots illustrate the effectiveness of our mutation algorithms. For most seed programs, SixthSense was able to generate a well-balanced set of semantically correct mutants

– for over 60% of the 166 seeds, the ratio between converging and non-converging programs is between 0.3 and 0.7. The total number of semantically-correct generated mutants per program is generally high (more than 70% on average across three classes), which is particularly important for the classes that have fewer seed programs like MixtureModels and TimeSeries.

When using a larger $\hat{R}$ threshold, more seed programs will be marked as converging. In those cases, our Algorithm 3.4 is particularly helpful in generating harder mutants to get around 50% non-converging cases. One might also observe in Figure 3.5 that, the Majority Label Classifier, which reflects the ratio of converging or non-converging cases, always has its accuracy around 50%.

Several seed programs pose difficulty to the mutation algorithm to generate semantically valid programs. The common features include computations on array indices, custom functions, or positive definite matrices. However, the number of these programs is relatively small: less than 10% Regression models, 17% Mixture models, and 22% Time Series models.

3.7.5 Training and Testing Times and Prediction Model Sizes

Table 3.7 presents the statistics of different parts of SixthSense's execution. Each row presents one class of models. Columns 2 and 3 present the number of the original models from the Stan repository and the number of generated mutants. Column 4 presents the time to generate all the mutants. Column 5 presents the time to run the generated programs and collect the execution statistics in the logs (we reuse the information about the model runs when re-training the data). This includes the time to run each program using NUTS algorithm in Stan for 100,000 iterations for one chain and 1000 iterations for four chains, and computing the accuracy and convergence metrics. Further, it also includes the time to collect the run-time features, as discussed in Section 3.4.3 for 10-100 iterations. Column 6 shows the time to compute the program's static features (Section 3.4.1). Finally, Column 7 and 8 present the time to train and test our random forest models.

Executing the seed and mutated programs is the most expensive step in the training. Running the sampling algorithms for 100,000 iterations for each model takes significant time, especially for more complex models. The average training time per threshold for SixthSense is less than a minute, for Code2Vec about 6 minutes (6x slower) and Code2Seq over two hours (120x slower), as it tries to learn the features during training. Computing the AST motifs and data features takes less than 1 second on average per instance for SixthSense. Code2Vec and Code2Seq take about 0.3 seconds per instance (but without data features).

Table 3.8 presents the size (in MBs) of the gzip-compressed prediction models for the three

tools. SixthSense's models are 25-37% smaller than Code2Vec and Code2Seq.

Table 3.8: The Size of Prediction Models

Class	SixthSense	Code2Vec	Code2Seq
Regression	19M	29M	26M
Mixture	17.7M	20.3M	23M
TimeSeries	16.3M	24M	25.3M

3.8 SENSITIVITY ANALYSIS

We present various sensitivity analyses of SixthSense to justify our design choices.

3.8.1 Feature Ablation Study

Table 3.9 shows the Accuracy score for convergence predictions when trained with different combinations of feature groups (AST features, AST and data features, and all features). Runtime features are from 200 warmup iterations. The AST features (motifs) alone contribute a major portion to the Accuracy scores in all cases. Data features do not have much impact on these models. Runtime features, after a certain number of iterations further improve prediction (they are in fact a strong predictor, but do not establish a relation with the program code). Obtaining runtime statistics comes at a cost of compiling and running the program, which can be over 30 seconds for Stan.

3.8.2 Impact of the Noisy Labels on the Prediction

Noisy labels, where binary labels are randomly flipped, pose a common challenge in binary classification tasks. To evaluate the robustness of our predictions, we intentionally introduced label noise into the training set at various noise levels and conducted experiments using two approaches: one involving a *robust* version of SixthSense that employs a technique called Rank Pruning [96], which is known for its ability to enhance the training for binary classification when labels are noisy, and the other using the basic version of SixthSense without Rank Pruning. Importantly, Rank Pruning seamlessly integrates with SixthSense.

Table 3.10 shows the Accuracy scores for the different model classes for several noise levels (1-5%). For each noise level, the **R** (Robust) column shows the scores when trained using the Rank Pruning algorithm and the **B** (Basic) column shows the scores for the basic SixthSense. Even in the presence of significant training noise, our learning approach maintains high

Table 3.9: Ablation Study ($\hat{R}$=1.05)

Class	A	A+D	A+RT	A+D+RT
Regression	0.77	0.77	0.83	0.83
Mixture	0.78	0.78	0.87	0.87
TimeSeries	0.79	0.79	0.84	0.85

Table 3.10: Training with Noisy Labels ($\hat{R}$=1.05)

Label Flip Pr.	**1%**		**3%**		**5%**	
Model Class	**R**	**B**	**R**	**B**	**R**	**B**
Regression	0.765	0.760	0.763	0.765	0.760	0.764
Mixture	0.772	0.784	0.774	0.782	0.783	0.785
TimeSeries	0.786	0.789	0.794	0.781	0.781	0.788

Accuracy scores. For instance, the performance of Mixture Models remains almost constant (close to 78%), whether Rank Pruning is applied or not, even when 5% labels are wrong.

3.8.3 Motif Ablation Study

We performed a sensitivity study to determine the importance of different *motifs* obtained from AST in prediction. First, we looked at different motif sizes. For three motif sizes (5, 10, 20) on the threshold $\hat{R} = 1.05$, we do not see a significant increase in the Accuracy score. This reflects that even smaller motifs obtained from probabilistic programs can be very effective for predicting their runtime behavior. Therefore, we used Motif size of 5 in all our experiments.

We select only non-overlapping motifs from the features set (i.e motifs with no common nodes or sequence of nodes in AST) and use them for prediction. Next, we repeat the same experiments, but with smaller randomly sampled subsets of non-overlapping motifs. Tables 3.11 shows the results for this study for convergence and accuracy prediction respectively. First column shows the model class. Columns 2-5 show the F1 scores when we use 10%,40%,80%, and 100% of the non-overlapping motifs. The final column shows the F1 scores that we obtain when using all original motifs (as in Sections 3.7.1 and 3.7.2).

Table 3.11: Using Motif Subsets ($\hat{R}$=1.05)

Class	0.1	0.4	0.8	1.0	All
Regression	0.60	0.70	0.72	0.72	0.77
Mixture	0.62	0.69	0.72	0.73	0.78
TimeSeries	0.61	0.73	0.73	0.77	0.79

We observe that the scores drop slightly compared to the original scores when using only non-overlapping motifs. The scores gradually deteriorate when using smaller subsets

of non-overlapping motifs. This shows that although removing some overlapping motifs might help reduce the feature space and improve the training time, the user still needs to pay the penalty of worse predictions.

3.8.4 Other Sensitivity Studies

We also performed other sensitivity studies on the features and generated programs. This included the following experiments: sub-sampling motif subsets for each program, using different LSH configurations to remove syntactically similar programs from the training set, and increasing the motif depth. However, these experiments did not show substantial deviation from the F1 scores we obtained in the main experiments.

3.9 RELATED WORK

Probabilistic Programming. Probabilistic programming offers a means to encode intricate statistical models into straightforward computer programs, serving as a powerful tool to capture and analyze uncertainty. Recently, probabilistic programming languages (PPLs) and their underlying inference systems have gained significant interest from research and industry [1, 8, 61, 70, 97, 98, 99, 100, 101, 102]. Typically, PPLs (e.g., Stan) only provide simple runtime diagnostics and timing information as they run. In contrast, SixthSense is a predictive data-driven approach that complements these efforts.

The prior debugging approach for PPLs [103] requires augmenting Bayesian network representation with additional labels and extending the inference algorithm. However, its applicability is limited as state-of-the-art PP systems typically do not use Bayesian network representation. In contrast, our approach learns program features for debugging without altering the inference algorithm. Other existing tools [33, 34] target lower-level implementation bugs in probabilistic programming systems. Meanwhile, the statistics community explores enhancing model robustness against data noise [104], while our work addresses non-convergence issues in model inference.

Several recent approaches have explored the nature of regression tests in probabilistic and machine learning applications such as the causes and fixes for flaky tests [105, 106], usage of seeds in tests [107], and speeding up expensive regression tests [108].

Predicting Program Properties from Big-Code. Much attention has recently been devoted to uses of machine learning to analyze and predict various program properties. Notable examples include predicting variable names/types via statistical program models [79],

predicting patches [109], summarizing code [80, 110], API discovery [77, 111], and bug detection/repair [112, 113, 114]. However, all of these works apply learning to conventional programs (C/Java/Javascript), obtained from massive code repositories. Moreover, many of these approaches predict static program properties (e.g., names/types), rather than execution properties like convergence. While some of these approaches benefit from the natural-language semantics of identifiers [77, 78], we are interested in semantics of the program itself, which are better represented by the sequence of AST nodes.

We also present how to augment the corpus of programs with diverse programs via guided mutation. While our approach bears similarity to data augmentation in machine learning [115, 116, 117], probabilistic programs have complex structure defined by many syntactic (and often semantic) rules.

Predicting Algorithm Performance. Researchers developed machine learning approaches that predict hardness of NP-hard problems (e.g., SAT, SMT, ILP) [118, 119, 120]. These works are complementary and their syntax and semantics are considerably simpler than for probabilistic programs. Researchers also proposed models for performance of other machine learning architectures [121, 122, 123, 124], but their techniques and applications are orthogonal to ours.

Transformer-based Code Generation. Recently, new advances in transformer-based neural networks have demonstrated substantial improvment in code generation quality. Popular examples include AlphaCode [125], ChatGPT [126] and Codex [127] that can generate a program in seconds for a given natural language prompt. Other systems like [128] are designed for edit-time code completion in IDEs. These systems report a high result performance, which is comparable to humans. While these approaches can help with program generation or property prediction, training these systems usually requires a large code corpus, which is not available for probabilistic programs. Further, it is unclear how to use these systems to predict semantic properties of programs (like convergence).

3.10 CONCLUSION

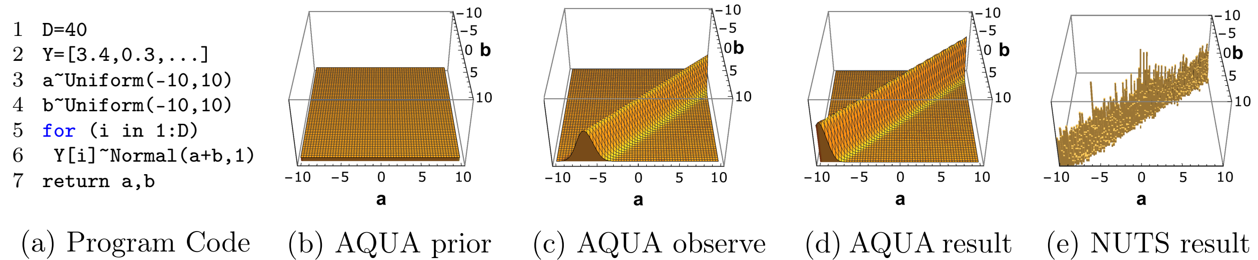
We presented SixthSense, a novel approach and system, which predicts convergence for probabilistic programs and helps guide the debugging of convergence issues. Our results demonstrate the effectiveness of SixthSense in extracting features from probabilistic programs and learning a prediction model. When compared to the state-of-the-art techniques, SixthSense achieves a significant improvement in accuracy, exceeding an accuracy score of 78% for predicting convergence. SixthSense exhibits the potential to pinpoint the causes of non-convergence issues, which offers practical support for software developers and scientists

in addressing convergence issues and enhance the reliability of probabilistic programming practices.

Supplementary Information. SixthSense is publicly avaliable at `https://github.com/uiuc-arc/sixthsense`.

CHAPTER 4: AUTOMATED QUANTIZED INFERENCE FOR PROBABILISTIC PROGRAMS WITH AQUA

4.1 INTRODUCTION



(a) Program Code (b) AQUA prior (c) AQUA observe (d) AQUA result (e) NUTS result

Figure 4.1: Example program and AQUA probabilistic density v.s. NUTS histogram

Many modern applications (e.g., in machine learning, robotics, autonomous driving, medical diagnostics, and financial forecasting) need to make decisions under uncertainty. Probabilistic programming languages (PPLs) offer an intuitive way to model uncertainty by representing complex probabilistic models as simple programs [8]. They expose randomness and Bayesian inference as first-class abstractions by extending standard languages with statements for sampling from probability distributions and probabilistic conditioning. The underlying programming system then automates the intricate details of the probabilistic inference.

Probabilistic inference is a computationally hard problem. Most current approaches that emerged from the statistics and machine learning communities applied aggressive numeric approximations, such as Markov Chain Monte Carlo sampling (MCMC) or Variational Inference (VI). However, these approaches often cannot obtain the level of accuracy that is required in applications such as algorithmic fairness [13], security/privacy [4, 129], sensitivity analysis [31, 130], or software testing [20].

Symbolic techniques for inference have been resurging as a more accurate alternative. They use a symbolic representation of the model's state (e.g., elementary functions, piecewise-linear functions, or hypercubes), and compute the posterior distribution algebraically [19, 20, 22] or closely approximate programs using volume computation [4, 13, 25]. However, these approaches are often limited by the classes of programs they can solve. For instance, continuous programs pose a major challenge for these approaches due to integrals in posterior calculation. State-of-the-art symbolic solvers cannot solve many integrals exactly (often, the integrals do not have a closed form). Similarly, volume computation approaches have a limited support for continuous distributions (e.g., do not allow for conditioning on continuous random variables) and/or compute the probability of a single event, not the entire posterior

distribution. *An intriguing research question is how to approximate multi-dimensional continuous distributions in a principled manner that allows for more expressive programs and can solve programs that are out of reach of existing tools for exact inference.*

AQUA. We present AQUA, a novel system for symbolic inference that uses quantization of probability density function for delivering scalable and precise solutions for a broad range of probabilistic programs. AQUA's inference algorithm approximates the original continuous program via an efficient quantization of the continuous distributions by using multi-dimensional tensor representations that we call *Interval Cube and Density Cube.* The Interval Cube stores the quantized value ranges of variables in the probabilistic program. The Density Cube approximates the joint posterior distribution by recording the probability of each hypercube contained in the interval cube.

AQUA's analysis transforms the symbolic state to compute quantized approximate posterior distribution.We derive the bounds for the approximation error (due to the quantization and integration) and show that our inference converges in distribution to the true posterior. We also present an adaptive algorithm for automatically selecting the granularity of the Interval and Density Cubes.

Example. Figure 4.1a presents a probabilistic program that represents the distribution of two random variables. In the program, we have two random variables `a` and `b`, each having `Uniform` *prior* distribution (Lines 3-4). We then condition the model on 40 data points `Y`, assuming that each point is normally distributed with the mean `a+b` (Lines 5-6). We finally query for the joint *posterior* distribution (i.e., the distribution of latent variables `a` and `b` after observing the data on Line 6).

Figure 4.1 presents AQUA's results: 4.1b shows the prior of the two variables, 4.1c shows the likelihood (observation) on a single data point, and 4.1d shows the posterior distribution. On each plot, the X-axis and Y-axis represent `a` and `b` values, and the Z-axis values are the probability densities computed by AQUA. AQUA computes the result in 0.76s, whereas an MCMC based inference algorithm (NUTS) produces a less accurate posterior within the same amount of time (Figure 4.1e).

Evaluation. We evaluate our implementation of AQUA on a set of 24 probabilistic programs from the literature. We compare AQUA with exact inference – PSI [20] and SPPL [22] – and approximate inference – MCMC and VI implemetations in Stan [8]. We show that AQUA can solve programs that are out of reach for PSI and SPPL. Our results show AQUA solved all benchmarks in less than 43s (median 1.35s). It is significantly more accurate than VI for all programs (for the Kolmogorov-Smirnov metric). AQUA is substantially more accurate than MCMC for 10 programs, even when MCMC is given substantially more time

to complete. We also present a case study that shows AQUA can precisely capture the tails of the distribution of robust models.

Contributions. This chapter makes the following contributions:

- **Inference Algorithm:** We present AQUA, a novel inference algorithm that works on general, real-world probabilistic programs with continuous distributions based on quantization and symbolic computation.

- **Quantization with Interval and Density Cubes:** Our analysis defines symbolic transformers on the abstract state consisting of the Interval and Density Cubes. We also present theoretical bounds on the quality of approximation.

- **Inference Algorithm Optimizations:** We present algorithm extensions that automatically refine the size/granularity of the analysis to satisfy a given precision threshold and aggressively reduce the analysis overhead of local variables.

- **Evaluation:** Our experiments show that AQUA is more accurate than approximate inference algorithms (Stan's MCMC/VI) and supports programs with conditioning on continuous distributions that are out of reach of exact inference tools (PSI and SPPL).

4.2 PRELIMINARIES

Language Syntax and Semantics. Figure 4.2 describes the syntax of a probabilistic program using an imperative, first-order intermediate representation, drawing from Storm-IR [33, 34]. It has statements for sampling from distributions[1] and conditioning on data with `factor` and `observe`.

$x \in$ Vars
$c \in$ Consts
$op \in \{+, -, *, >, \ldots\}$
$d \in$ {`Normal`,`Uniform`,...}

```
E  :=  c | x | E[E*] | E op E |d(E*).pdf(E*) |f(E*)
S  :=  x = E |x ~ d(E*) |factor(E) |observe(d(E*),x)
       | if (E) S* else S* | for x ∈ 1..N; {S*}
P  :=  S+; return x+
```

Figure 4.2: Syntax of AQUA's language

The language semantics are standard, inspired by those presented by Gorinova et al. [30].

[1] We support common continuous distributions including Normal, Uniform, Exponential, Beta, Gamma, Student-T, Laplace, Triangular, and any mixture of the above distributions.

Table 4.1: Correspondence of Symbolic and Concrete Analysis

Concrete	Symbolic
Value $\sigma(x) : \mathbb{R}$	Interval Cube $\sigma^{\#}(x) : \mathbb{I}$
Density $\sigma(\mathcal{L}) : \mathbb{R}^{+}$	Density Cube $\sigma^{\#}(P) : (\mathbb{R}^{+})^{\Pi_{i=1}^{N} M_i}$
State : $\mathcal{P}(\textbf{Vars} \mapsto \mathbb{R}) \mapsto \mathbb{R}^{+}$	**Astate** : $\mathcal{P}(\textbf{Vars} \mapsto \mathbb{I}) \mapsto (\mathbb{R}^{+})^{\Pi_{i=1}^{N} M_i}$
$[\![E]\!]$: **State**$\mapsto\mathbb{R}$	$[\![E]\!]^{\#}$: **Astate**$\mapsto\mathbb{I}$
$[\![S]\!]$: **State**$\mapsto$**State**	$[\![S]\!]^{\#}$: **Astate**$\mapsto$**Astate**

In summary, a probabilistic program evaluates the *posterior probability density function*. Our operational semantics for a program defines its effect on the program state, σ, which maps variables to values. A value V can either be a constant c or an array of values $[c_1, c_2, ...]$. The notations $\sigma(x)$ and $\sigma(x \mapsto V)$ denote accessing and updating a variable x respectively. We refer to the return variables of the program as the *global variables*, and the others as *local variables*. We allow local variables to have discrete distributions (e.g. Bernoulli), as long as the density of the global variables are Lipschitz continuous. We define a special variable $\mathcal{L} \in \mathbb{R}^{+}$ which tracks the *unnormalized posterior density* of the probabilistic program. We initialize $\sigma(\mathcal{L})$ to 1.0 at the start of the program.

Probability Density. We review several basic terms from the probability theory. Let $\boldsymbol{x}$ be the set of variables with values in V, and $\mathcal{D}$ be the set of observed data points. Then, the posterior probability density function is $p(\boldsymbol{x} \mid \mathcal{D}) : V \rightarrow \mathbb{R}$, such that $\int_{\boldsymbol{x} \in V} p(\boldsymbol{x} \mid \mathcal{D}) d\boldsymbol{x} = 1$. The distribution $p(\boldsymbol{x} \mid \mathcal{D})$ can be calculated from the *unnormalized probability density function* $f(\boldsymbol{x}, \mathcal{D}) : V \rightarrow \mathbb{R}$, by $p(\boldsymbol{x} \mid \mathcal{D}) = \frac{1}{z} f(\boldsymbol{x}, \mathcal{D})$, where z is the normalizing constant: $z = \int f(\boldsymbol{x}, \mathcal{D}) d\boldsymbol{x}$. If $\boldsymbol{x}_{-i}$ contains all the variables in $\boldsymbol{x}$ excluding x_i, we define the *marginal probability density function* of x_i as $p(x_i \mid \mathcal{D}) = \int p(\boldsymbol{x} \mid \mathcal{D})\ d\boldsymbol{x}_{-i}$. Hereon, we omit data symbol $\mathcal{D}$ to write $p(\boldsymbol{x})$ and $f(\boldsymbol{x})$ when clear from the context. In the semantics, $f(\boldsymbol{x})$ is represented by $\sigma(\mathcal{L})$.

4.3 INFERENCE WITH DENSITY CUBES

4.3.1 Notation and Basic Definitions

We represent the closed, bounded set $\{x \in \mathbb{R} \mid a \leq x \leq b\}$ with its lower-bound $a \in \mathbb{R}$ and upper-bound $b \in \mathbb{R}$. We denote this abstraction as an **interval** $I = [a, b] \in \mathbb{R}^2$. We refer to the lower and upper bound of I as $\underline{I}$ and $\overline{I}$, respectively ($\underline{I}, \overline{I} \in \mathbb{R}$).

A probabilistic program lifts a normal program operating on single values to a *distribution* over values. Hence, a probabilistic program represents a joint distribution over its variables. For our symbolic analysis, to represent the quantized values of variables, we define tensors

of intervals which we will refer to as *Interval Cube*. We also assign a probability density to each interval in the Interval Cube. We will refer to this assignment of densities as *Density Cube*. If there are N variables in the program, the Density Cube will be an N-dimensional tensor.

Definition 4.1 (Interval Cube)**.** We represent the value of a variable x with Interval Cube, $\boldsymbol{I}^x_{M_1,M_2,\ldots,M_N}$ where $[M_1, M_2, \ldots, M_N]$ represents the shape of the Interval Cube and each $M_i \in \mathbb{N}$ is the *number of intervals* (*splits*) along the i-th dimension. Each element of $\boldsymbol{I}^x_{M_1,M_2,\ldots,M_N}$ is a single interval. We let $\mathbb{I}$ be the set of all Interval Cubes. For a constant c, we denote its Interval Cube as $[c]$, meaning a singleton interval with both lower and upper bounds being c.

Example 4.1. *Suppose a program has $x \sim$ Beta(2,2), meaning that x following a Beta distribution. Beta distribution has bounded support $0 \leq x \leq 1$, and thus we consider splitting the possible values into, say, 10 equal-length intervals: $\boldsymbol{I}^x_{10} = [[0, 0.1], [0.1, 0.2], \ldots, [0.9, 1]]$.*

If the support is unbounded, e.g. $x \sim$ Normal(0,1), where the Normal distribution has infinite support, we will truncate the support into a bounded interval, and ensure the probability that x being out of this interval is small. For example, we may consider $-12 \leq x \leq 12$ with $Pr(x < -12$ or $x > 12) = 3.6 \cdot 10^{-33}$, and then split the interval into 10 equal-length intervals: $\boldsymbol{I}^x_{10} = [[-12, -9.6], [-9.6, -7.2], \ldots, [9.6, 12]]$. In practice, AQUA will adaptively select the bounded interval (Algorithm 4.2).

Definition 4.2 (Shape of the Interval Cube)**.** To simplify the notation, we hereon denote the shape of the hypercube as $\boldsymbol{M} = [M_1, M_2, \ldots, M_N]$ and each index in the hypercube is $\boldsymbol{m} \in \mathbb{M}$, $\mathbb{M} = \{[m_1, \ldots, m_N] \mid m_i \in [1, \ldots, M_i],\ i \in \{1, \ldots, N\}\}$. We write $\boldsymbol{K} = \boldsymbol{M}_1 \odot \boldsymbol{M}_2$ as the element-wise product (Hadamard product) operation on two shape vectors, namely $K_i = M_{1i} \times M_{2i}$, $i \in \{1, \ldots, N\}$. We will use the operation in the computation of multi-dimensional Interval Cubes (see Example 4.3). We use $\boldsymbol{m}_1$ to denote the index of a Interval Cube with shape $\boldsymbol{M}_1$, $\boldsymbol{m}_1 = [m_1, \ldots, m_N]$, $m_i \in \{1, \ldots, M_{1i}\}$, and similarly we use $\boldsymbol{m}_2$ for index in $\boldsymbol{M}_2$, and $\boldsymbol{k}$ for index in $\boldsymbol{K}$.

Definition 4.3 (Density Cube)**.** For a given probabilistic program *Prog* with N sampled variables, $\boldsymbol{x} = \{x_1, \ldots, x_N\}$, we define the Density Cube with shape $\boldsymbol{M} = [M_1, \ldots, M_N]$ as $\boldsymbol{P}^{Prog}_{\boldsymbol{M}}$, where

$$\boldsymbol{P}^{Prog}_{\boldsymbol{M}}(\boldsymbol{m}) = p_{\boldsymbol{m}}, \text{ for each index } \boldsymbol{m} \in \mathbb{M}, \tag{4.1}$$

and $p_{\boldsymbol{m}}$ denotes the value of the unnormalized probability density function at the lower bound of the corresponding interval in the Interval Cube. The densities at the lower bound

of intervals will help us do numerical integration for posterior calculation. Here we use the density at the lower bound for convenience. Using the upper bound or the midpoint will give the same accuracy guarantee (Theorem 4.1). Further, $\boldsymbol{P}_{\boldsymbol{M}}^{Prog} \in (\mathbb{R}^+)^{\prod_{i=1}^N M_i}$, and $p_{\boldsymbol{m}} \in \mathbb{R}^+$.

Example 4.2 (Density Cube for a Single Variable)**.** *In Example 4.1 where* $\boldsymbol{I}_{10}^x = [[0, 0.1], [0.1, 0.2], \ldots, [0.9, 1]]$*, the corresponding Density Cube is* $\boldsymbol{P}_{10}^x = [0, 0.54, \ldots, 0.54]$*, which are the densities of Beta(2,2) at each interval's lower bound. When there are sufficiently many splits, the discretized* $\boldsymbol{P}_{10}^x$ *will converge to the true density.*

Example 4.3 (Density Cube for Multiple Variables)**.** *Suppose we have a program defining two variables:* $x_1 \sim$ *Beta(2,2),* $x_2 \sim$ *Beta(2,2). If we apply 10 equal-length splits for* x_1 *and 5 equal-length splits for* x_2*, two Interval Cubes will be defined:* $\boldsymbol{I}_{10}^{x_1} = [[0, 0.1], [0.1, 0.2], \ldots, [0.9, 1.0]]$ *and* $\boldsymbol{I}_5^{x_2} = [[0, 0.2], [0.2, 0.4], \ldots, [0.8, 1.0]]$*. The corresponding Density Cube* $\boldsymbol{P}_{10,5}^P$ *will have the shape* 10 *by* 5*, so that each element* $\boldsymbol{P}_{10,5}^P([m_1, m_2])$*,* $m_1 \in \{1, \ldots, 10\}$*,* $m_2 \in \{1, \ldots, 5\}$*, stores the approximate joint density when* $x_1 \in \boldsymbol{I}_{10}^{x_1}(m_1)$ *and* $x_2 \in \boldsymbol{I}_5^{x_2}(m_2)$*. For example,* $\boldsymbol{P}_{10,5}^P([10, 5]) = 0.5184$ *corresponds to* $x_1 \in \boldsymbol{I}_{10}^{x_1}(10) = [0.9, 1.0]$ *and* $x_2 \in \boldsymbol{I}_5^{x_2}(5) = [0.8, 1.0]$*. The value* 0.5184 *is the exact joint density for* $x_1 = 0.9$ *and* $x_2 = 0.8$*, and is the approximate joint density for other* x_1 *and* x_2 *in their intervals.*

For easy implementation and explanation, in AQUA we represent x_1*'s Interval Cube as* $\boldsymbol{I}_{10,1}^{x_1}$ *and* x_2*'s Interval Cube as* $\boldsymbol{I}_{1,5}^{x_2}$*, by reshaping their intervals along different dimensions. The shape of* $\boldsymbol{P}_{10,5}^P$ *will simply be element-wise product (Hadamard product) of the individual shape vectors,* $[10, 1] \odot [1, 5]$*. In fact, we represent the intervals of all the* Sampled Variables *on different dimensions. Sampled Variables are variables initialized by sampling statements (e.g.* $x_1 \sim \text{Beta}(2, 2)$*), not deterministic assignments (e.g.* $x_3 = x_1 + x_2$*).*

Definition 4.4 (Symbolic Domain)**.** Our *symbolic state* has two components, a map from variables to Interval Cubes, and a Density Cube representing the joint density approximation. Let **Var** denote the set of variables, and $\mathcal{P}$ be the power set, the domain of the symbolic state is $\Sigma = \mathcal{P}(\mathbf{Var} \mapsto \mathbb{I}) \mapsto (\mathbb{R}^+)^{\prod_{i=1}^N M_i}$ a symbolic state $\sigma^\# \in \Sigma$ will have the form

$$\sigma^\# = \left\langle \{x_1 \mapsto \boldsymbol{I}_{\boldsymbol{M}_1}^{x_1}, x_2 \mapsto \boldsymbol{I}_{\boldsymbol{M}_2}^{x_2}, \ldots\}, P \mapsto \boldsymbol{P}_{\boldsymbol{M}}^{Prog} \right\rangle. \quad (4.2)$$

The symbolic domain expresses a piecewise constant interpolation of the joint probability density at a program point. Hereon, we refer to the set of all the variables in the state $\sigma^\#$ as $\boldsymbol{x} = \{x_1, x_2, \ldots, x_N\}$. The shape vectors $\boldsymbol{M}_1$, $\boldsymbol{M}_2, \ldots$ will remain the same throughout the program.

$$
\begin{aligned}
[\![E]\!]^\# &\mapsto (\textbf{Astate} \ \mapsto \text{Interval Cube}) \\
[\![x]\!]^\# &\coloneqq \lambda\sigma^\#.\sigma^\#(x) \\
[\![c]\!]^\# &\coloneqq \lambda\sigma^\#.[c] \\
[\![E_1[E_2]]\!]^\# &\coloneqq \lambda\sigma^\#.\text{let } [c,c] = [\![E_2]\!]^\#\sigma^\# \text{ in } [\![E_1[c]]\!]^\#\sigma^\# \\
[\![E_1 \ op \ E_2]\!]^\# &\coloneqq \lambda\sigma^\#.\text{let } \boldsymbol{I}^{E_1}_{\boldsymbol{M}_1} = [\![E_1]\!]^\#\sigma^\#,\ \boldsymbol{I}^{E_2}_{\boldsymbol{M}_2} = [\![E_2]\!]^\#\sigma^\#,\ \boldsymbol{K} = \boldsymbol{M}_1 \odot \boldsymbol{M}_2 \\
&\qquad \text{in } \boldsymbol{I}^{E_1 \ op \ E_2}_{\boldsymbol{K}},\ \text{where } \boldsymbol{I}^{E_1 \ op \ E_2}_{\boldsymbol{K}}(\boldsymbol{k}) = \boldsymbol{I}^{E_1}_{\boldsymbol{M}_1}(\boldsymbol{m}_1) \ op \ \boldsymbol{I}^{E_2}_{\boldsymbol{M}_1}(\boldsymbol{m}_2) \\
[\![d(E_1,\ldots,E_{n-1}).\text{pdf}(E_n)]\!]^\# &\coloneqq \lambda\sigma^\#.\text{let } \boldsymbol{I}^{E_1}_{\boldsymbol{M}_1} = [\![E_1]\!]^\#\sigma^\#,\ldots,\boldsymbol{I}^{E_n}_{\boldsymbol{M}_n} = [\![E_n]\!]^\#\sigma^\#,\ \boldsymbol{K} = \bigodot_{i=1}^{n} \boldsymbol{M}_i, \\
&\qquad \text{in } \boldsymbol{I}^{\text{dpdf}}_{\boldsymbol{K}},\ \text{where } \boldsymbol{I}^{\text{dpdf}}_{\boldsymbol{K}}(\boldsymbol{k}) = d_\text{pdf}(\boldsymbol{I}^{E_n}_{\boldsymbol{M}_n}(\boldsymbol{m}_n), \boldsymbol{I}^{E_1}_{\boldsymbol{M}_1}(\boldsymbol{m}_1), \ldots, \boldsymbol{I}^{E_{n-1}}_{\boldsymbol{M}_{n-1}}(\boldsymbol{m}_{n-1}))
\end{aligned}
$$

Figure 4.3: Analysis of Expressions

4.3.2 Analysis

We approximate the posterior density function of variables in our symbolic states. Table 4.1 presents the correspondence of the objects in concrete semantics to symbolic states. While a concrete state has a single valuation of variables and evaluates to a single density value, our symbolic state stores all possible variable values in Interval Cube and the corresponding joint probability density in Density Cube. As the concrete semantics for a expression maps state to values, the symbolic semantics map symbolic state to Interval Cube; and as the concrete semantics for a statement map state to state, our symbolic semantics map symbolic state to symbolic state.

Analysis of Expressions. The symbolic transformer $[\![E]\!]^\#$ on an expression E takes a symbolic state $\sigma^\#$: **Astate** as input, and outputs an Interval Cube. Figure 4.3 presents the rules. We explain two important cases in detail:

- $[\![E_1 \ op \ E_2]\!]^\#$: For the arithmetic/boolean operation on two Interval Cubes, which may not always have the same shape, the resulting Interval Cube needs to contain all possible value combinations. Specifically, for $\boldsymbol{I}^{E_1}_{\boldsymbol{M}_1}$ with shape $\boldsymbol{M}_1 = [M_{11}, \ldots, M_{1N}]$ and $\boldsymbol{I}^{E_2}_{\boldsymbol{M}_2}$ with shape $\boldsymbol{M}_2 = [M_{21}, \ldots, M_{2N}]$, the result $\boldsymbol{I}^{E_1 \ op \ E_2}_{\boldsymbol{K}}$ has shape $\boldsymbol{K} = [K_1, \ldots, K_N]$ with $K_i = M_{1i} \times M_{2i}$ to capture all the combinations of elements from $\boldsymbol{I}^{E_1}_{\boldsymbol{M}_1}$ and $\boldsymbol{I}^{E_2}_{\boldsymbol{M}_2}$. If $\boldsymbol{M}_1$ and $\boldsymbol{M}_2$ are not of the same length, we reshape both $\boldsymbol{I}^{E_1}_{\boldsymbol{M}_1}$ and $\boldsymbol{I}^{E_2}_{\boldsymbol{M}_2}$ to have the same dimension, by letting some M_{1i} or M_{2i} to have value 1. We let the arithmetic or boolean operation on the interval pairs be $\boldsymbol{I}^{E_1}_{\boldsymbol{M}_1}(\boldsymbol{m}_1) \ op \ \boldsymbol{I}^{E_2}_{\boldsymbol{M}_2}(\boldsymbol{m}_2) \coloneqq [\underline{\boldsymbol{I}^{E_1}_{\boldsymbol{M}_1}(\boldsymbol{m}_1) \ op \ \boldsymbol{I}^{E_2}_{\boldsymbol{M}_2}(\boldsymbol{m}_2)}, \overline{\boldsymbol{I}^{E_1}_{\boldsymbol{M}_1}(\boldsymbol{m}_1) \ op \ \boldsymbol{I}^{E_2}_{\boldsymbol{M}_2}(\boldsymbol{m}_2)}]$. We handle the case with multiple intervals analogously. This operation on multiple Interval Cubes can be implemented efficiently with the *broadcast* function in tensor libraries.
- $[\![d(E_1, \ldots, E_{n-1}).\text{pdf}(E_n)]\!]$: Similar to arithmetic operator, we apply the mathematical density $d_\text{pdf}(_)$ of the distribution d whose parameters (e.g., mean, location, shape or

$$
\begin{aligned}
[\![S]\!]^{\#} &\mapsto (\textbf{Astate} \mapsto \textbf{Astate}) \\
[\![\text{skip}]\!]^{\#} &\coloneqq \lambda\sigma^{\#}.\sigma^{\#} \\
[\![S_1; S_2]\!]^{\#} &\coloneqq \lambda\sigma^{\#}.[\![S_2]\!]^{\#}([\![S_1]\!]^{\#}\sigma^{\#}) \\
[\![x = E]\!]^{\#} &\coloneqq \lambda\sigma^{\#}.\text{let } \boldsymbol{I} = [\![E]\!]^{\#}\sigma^{\#} \text{ in } \sigma^{\#}(x \mapsto \boldsymbol{I}) \\
[\![x \sim d(E_1, \ldots, E_n)]\!]^{\#} &\coloneqq \lambda\sigma^{\#}.\text{let } \boldsymbol{P}_{\boldsymbol{M}_0} = \sigma^{\#}(P),\ \boldsymbol{I}_{\boldsymbol{K}}^{\text{dpdf}} = [\![d(E_1,\ldots,E_n).\text{pdf}(x)]\!]^{\#}\sigma^{\#},\ \text{in} \\
&\qquad \text{let } \boldsymbol{M} = \boldsymbol{M}_0 \odot \boldsymbol{K},\ \text{in } \sigma^{\#}(P \mapsto \boldsymbol{P}'_{\boldsymbol{M}}), \\
&\qquad \text{where } \boldsymbol{P}'_{\boldsymbol{M}}(\boldsymbol{m}) = \boldsymbol{P}_{\boldsymbol{M}_0}(\boldsymbol{m}_0) \cdot \underline{\boldsymbol{I}_{\boldsymbol{K}}^{\text{dpdf}}(\boldsymbol{k})},\ \text{for all } \boldsymbol{m} = \boldsymbol{m}_0 \odot \boldsymbol{k}, \\
&\qquad \boldsymbol{m}_0 \in \{[m_{01}, \ldots, m_{0N}] \mid m_{0i} \in \{1, \ldots, M_{0N}\}\}, [M_{01}, \ldots, M_{0N}] = \boldsymbol{M}_0, \\
&\qquad \boldsymbol{k} \in \{[k_1, \ldots, k_N] \mid k_i \in \{1, \ldots, K_N\}\},\ [K_1, \ldots, K_N] = \boldsymbol{K} \\
[\![\text{factor}(E)]\!]^{\#} &\coloneqq \lambda\sigma^{\#}.\text{let}\boldsymbol{P}_{\boldsymbol{M}_0} = \sigma^{\#}(P),\ \boldsymbol{I}_{\boldsymbol{K}} = [\![E]\!]^{\#}\sigma^{\#},\ \boldsymbol{M} = \boldsymbol{M}_0 \odot \boldsymbol{K} \\
&\qquad \text{in } \sigma^{\#}(P \mapsto \boldsymbol{P}'_{\boldsymbol{M}}),\ \text{where } \boldsymbol{P}'_{\boldsymbol{M}}(\boldsymbol{m}) = \boldsymbol{P}_{\boldsymbol{M}_0}(\boldsymbol{m}_0) \cdot \underline{\boldsymbol{I}_{\boldsymbol{K}}(\boldsymbol{k})} \\
&\qquad \text{where } \boldsymbol{P}'_{\boldsymbol{M}},\ \boldsymbol{P}_{\boldsymbol{M}_0} \text{and } \boldsymbol{P}_{\boldsymbol{K}} \text{ are as above} \\
[\![\text{observe}(d(E_1, \ldots, E_{n-1}), E_n)]\!]^{\#} &\coloneqq \lambda\sigma^{\#}.[\![\text{factor}(d(E_1,\ldots,E_{n-1}).\text{pdf}(E_n))]\!]^{\#}\sigma^{\#} \\
[\![\text{if } (E) \text{ then } \{S_1\} \text{ else } \{S_2\}]\!]^{\#} &\coloneqq \lambda\sigma^{\#}.\left([\![\text{factor}(E); S_1]\!]^{\#}\sigma^{\#}\right) \sqcup \left([\![\text{factor(1-}E); S_2]\!]^{\#}\sigma^{\#}\right) \\
[\![\text{for } (i \text{ in } E_1..E_2)\ S]\!]^{\#} &\coloneqq \lambda\sigma^{\#}.[\![i{=}E_1; \text{if } (i{\le}E_2)\text{then}\{S; \text{for } (i \text{ in } E_1{+}1..E_2)S\}\text{else}\{\text{skip}\}]\!]^{\#}\sigma^{\#}
\end{aligned}
$$

Figure 4.4: Analysis of Statements

variance) are intervals obtained by evaluating $E_1, \ldots, E_{n-1}$, and it takes the intervals of E_n for which we seek the density. We denote the shape of the result Interval Cube as $\boldsymbol{K}$, which is computed from the element-wise product of the shapes of the input Interval Cubes.

Analysis of Statements. Figure 4.4 presents the transformers $[\![S]\!]^{\#}$ on statements S, which takes an abstract state $\sigma^{\#}$: **Astate** as input, and outputs an abstract state. We explain two important rules where we modify Density Cube (the remaining statements are standard or rely on these two rules):

• $[\![x \sim d(E_1, \ldots, E_n)]\!]^{\#}$, $[\![\text{factor}(E)]\!]^{\#}$: We first evaluate d.pdf(_) of the expressions into an Interval Cube, and multiply the current Density Cube with the lower bound of intervals from the Interval Cube. Then at the lower bound of each interval, the density is the same as the one from concrete semantics (Lemma 4.1). Intuitively, we discretize the density function and use the density at the lower bound to represent each interval. For convenience, our discretization uses the density at the lower bound. Using the density at the upper bound or the midpoint is also possible, and our accuracy guarantee (Theorem 4.1) still holds.

• $[\![\text{if } (E) \text{ then } \{S_1\} \text{ else } \{S_2\}]\!]^{\#}$: We first solve the results from two branches one conditioning on E and the other on $1 - E$. The true boolean expressions evaluate to 1 and false to 0 in our analysis, and we get the interval cubes for E and $1 - E$ from expression rules (Figure 4.3). We then *Join* the result states by adding up the Density Cubes from both branches.

Definition 4.5 (Joins)**.** *Join* ($\sqcup$) adds the Density Cubes from two states. Formally, $\sigma_1^\# \sqcup \sigma_2^\# = \sigma_1^\#(P \mapsto \boldsymbol{P}_{\boldsymbol{M}}^{Prog})$, where each element in $\boldsymbol{P}_{\boldsymbol{M}}^{Prog}$ at location $\boldsymbol{m}$ is $\sigma_1^\#(P)(\boldsymbol{m}) + \sigma_1^\#(P)(\boldsymbol{m})$, with $\boldsymbol{m} = [m_1, m_2, \ldots, m_N],\ m_i \in \{1, \ldots, M_i\},\ \boldsymbol{M} = [M_1, M_2, \ldots, M_N]$. Since we already initialized the global variables with their Interval Cube, $\sigma_1^\#$ and $\sigma_2^\#$ should have the same variables and Interval Cubes. Then the joint probability density P is changed to the sum of the densities from both states. Similarly, we can define *Meet* ($\sqcap$) by product of $\sigma_1^\#(P)(\boldsymbol{m})$ and $\sigma_1^\#(P)(\boldsymbol{m})$.

Algorithm. Algorithm 4.1 takes as input a probabilistic program *Prog*, the shape vector $\boldsymbol{M}$ where each element M_i is the number of intervals for variable x_i, and the interval bounds $\boldsymbol{C}$ (optional). In Section 4.4, we describe an adaptive scheme to automatically search for a proper $\boldsymbol{C}$ for the analysis.

First, it initializes the joint probability density P with the single interval [1.0] (Line 2). Then, it splits the value domain for each x_i in *SampledVars*, which are variables sampled from a prior distribution $x_i \sim d(E_1, \ldots, E_n)$ and not from deterministic assignments, into M_i equi-length intervals in C_i (in the function *GetInitIntervals*, Line 3-4). M_i is the i-th element in $\boldsymbol{M}$, and C_i is the i-th element in $\boldsymbol{C}$.

The algorithm follows the analysis rules to get the state at the end of the program (Line 5). Then it computes the joint probability density estimation $\hat{f}$, as a piecewise function of $\sigma^\#(P)$ (Line 6).

Definition 4.6 (Concretization of Symbolic States)**.** Define γ as the concretization function, s.t. $\gamma(\sigma^\#) = \hat{f}$, where $\hat{f}(\boldsymbol{x}) = \sigma^\#(P)(\boldsymbol{m})$ if $\boldsymbol{x} \in \bigotimes_{i=1}^N [\underline{\boldsymbol{I}_{M_i}^{x_i}(\boldsymbol{m})}, \overline{\boldsymbol{I}_{M_i}^{x_i}(\boldsymbol{m})}) \subset \mathbb{R}^N$ for any $\boldsymbol{m}$, and 0 otherwise. Intuitively, $\hat{f}$ is a piecewise constant interpolation of $\sigma^\#$.

The result $\hat{f}(\boldsymbol{x})$ is an approximation of the true unnormalized probability density function $f(\boldsymbol{x})$. In the concrete domain, the posterior probabilistic density function is calculated as $p(\boldsymbol{x}) = \frac{1}{z} f(\boldsymbol{x})$, but the integration $z = \int f(\boldsymbol{x}) d\boldsymbol{x}$ is often intractable. We compute our approximation $\hat{z}$ using integration on the piecewise function:

Definition 4.7 (Integration for Normalizing Constant)**.** Suppose there are N sampled variables $\boldsymbol{x}$ in the program, and let $\boldsymbol{C} = \bigotimes_{i=1}^N [a_i, b_i] \subset \mathbb{R}^N$ for each $x_i \in [a_i, b_i] \subset \mathbb{R}$ be the bounded domain used in the analysis ($\bigotimes$ represents the Cartesian Product on intervals on $\mathbb{R}$). We initialize $\sigma^\#[\boldsymbol{x}] = \boldsymbol{C}$ in the analysis. Then $z = \int_{\boldsymbol{C}} f(\boldsymbol{x})\ d\boldsymbol{x}$ is approximated with $\hat{z}$ $= \int_{\sigma^\#[\boldsymbol{x}]} \hat{f}(\boldsymbol{x})\ d\boldsymbol{x} = \sum_{\boldsymbol{m} \in \mathbb{M}} (\prod_{i=1}^N (\overline{\boldsymbol{I}_{M_i}^{x_i}(\boldsymbol{m})} - \underline{\boldsymbol{I}_{M_i}^{x_i}(\boldsymbol{m})}) \cdot \boldsymbol{P}_{\boldsymbol{M}}^P(\boldsymbol{m}))$.

The algorithm finally computes the posterior and the marginals for every variable (Lines 7-9). When the program has N variables, and each variable has the same number of intervals M, Algorithm 1 has the time complexity $\mathcal{O}(N \cdot M^N)$ and space complexity $\mathcal{O}(M^N)$.

Algorithm 4.1 Posterior Interval Analysis Algorithm

1: **procedure** POSTERIORANALYSIS($Prog$, $\boldsymbol{M}$, $\boldsymbol{C}$)
2: $\sigma^{\#}_{init} \leftarrow \{P \mapsto [1]\}$ ▷ Initialize
3: **for** [**do**]$x_i \in SampledVars(Prog)$
4: $\sigma^{\#}_{init}[x_i] \leftarrow GetInitIntervals(x_i, M_i, C_i)$
5: $\sigma^{\#} \leftarrow [\![Prog]\!]\sigma^{\#}_{init}$ ▷ Apply analysis rules
6: $\hat{f}(\boldsymbol{x}) \leftarrow PiecewiseFunc(\sigma^{\#}(P))$
7: $\hat{z} \leftarrow \int_{\sigma^{\#}[\boldsymbol{x}]} \hat{f}(\boldsymbol{x})\ d\boldsymbol{x};\quad \hat{p}(\boldsymbol{x}) \leftarrow \frac{1}{\hat{z}}\hat{f}(\boldsymbol{x})$ ▷ Posterior
8: **for** [**do**]$x_i \in SampledVars(Prog)$
9: $Marginal[x_i] \leftarrow \frac{1}{z}\int_{\sigma^{\#}[\boldsymbol{x}_{-i}]} \hat{p}(\boldsymbol{x})\ d\boldsymbol{x}_{-i}$ ▷ Marginalize
10: **return** ($\hat{p}$, $Marginal$)

Example 4.4 (Analysis Example)**.** *We use the following example to show how analysis works. It is a simplified version of the example in Section 4.1.*

```
1 a ~ Uniform(0,4)
2 b ~ Uniform(0,4)
3 c = a + b
4 observe(Normal(c, 1), 5)
5 return a, b
```

Figure 4.5: Analysis Example

Before the analysis starts, we initialize the Density Cube as $\sigma^{\#}_{init}(P) = [1.0]$*, which is a scalar. We also initialize the Interval Cubes of* a *and* b *as:*

$$\sigma^{\#}_{init}(a) = \begin{array}{|c|c|c|c|}\hline \texttt{[0, 1]} & \texttt{[1, 2]} & \texttt{[2, 3]} & \texttt{[3, 4]} \\ \hline\end{array} \tag{4.3}$$

$$\sigma^{\#}_{init}(b) = \begin{array}{|c|}\hline \texttt{[0, 1]} \\ \hline \texttt{[1, 2]} \\ \hline \texttt{[2, 3]} \\ \hline \texttt{[3, 4]} \\ \hline\end{array} \tag{4.4}$$

In this example we use 4 equal-length splits per sampled variable. AQUA infers the variable support $[0, 4]$ *from the Uniform priors. The user can specify the different number of intervals or interval bounds. Also, note that the Interval Cubes for* $\sigma^{\#}(a)$ *and* $\sigma^{\#}(b)$ *are on different dimensions. This will later us calculate the joint density or dependent expression.*

Then we go over the program to apply the analysis rules:

Line 1. *`a ~ Uniform(0,4)` defines the prior distributions for variables* a*. It times the initial Density Cube* $\sigma^{\#}_{init}(P)$ *with the density of* a *being at the lower bounds of the 4 intervals [[0, 1], [1, 2], [2, 3],[3, 4]] respectively:*

$(\llbracket \texttt{a\~{}Uniform(0,4)} \rrbracket \sigma^{\#}_{init})(P) =$ (4.5)

1/4	1/4	1/4	1/4

The first 1/4 approximates the probability density when $a \in [0, 1]$ *and the second 1/4 approximates the density when* $a \in [1, 2]$*, and so on. Denote the result state as* $\sigma^{\#}_{1}$*. Besides the Density Cube* $\sigma^{\#}_{1}(P)$ *,* $\sigma^{\#}_{1}$ *also contains the Interval Cubes* $\sigma^{\#}_{1}(a) = \sigma^{\#}_{init}(a)$ *and* $\sigma^{\#}_{1}(b) = \sigma^{\#}_{init}(b)$*.*

Line 2. `b ~ Uniform(0,4)` *defines the prior distributions for variables* b*. Denote the density of* b *at the lower bounds of the intervals as*

$P^{b} =$ (4.6)

1/4
1/4
1/4
1/4

then the result Density Cube representing the joint density of a *and* b *will be:*

$(\llbracket \texttt{b\~{}Uniform(0,4)} \rrbracket \sigma^{\#}_{1})(P) =$ (4.7)

1/16	1/16	1/16	1/16
1/16	1/16	1/16	1/16
1/16	1/16	1/16	1/16
1/16	1/16	1/16	1/16

where the element at column i *and row* j *is calculated from* $(\sigma^{\#}_{1}(P))(i) \cdot P^{b}(j) =$ *1/4*1/4. It represents the density when* a *is in the* i*-th interval of* $\sigma^{\#}_{1}(a)$ *and* b *is in the* j*-th interval of* $\sigma^{\#}_{1}(b)$*. Intuitively the rows correspond to the intervals of* a *and the columns correspond to the intervals of* b*. We let different sampled variables occupy different dimensions, e.g.* $\sigma^{\#}_{1}(a)$ *has shape* $\boldsymbol{M}_{a} = (4, 1)$ *while* $\sigma^{\#}_{1}(b)$ *has shape* $\boldsymbol{M}_{b} = (1, 4)$*. Then the shape of the Density Cube for the joint density is simply* $\boldsymbol{M}_{a} \odot \boldsymbol{M}_{b} = (4, 4)$*. Let the result state be* $\sigma^{\#}_{2}$ *with* $\sigma^{\#}_{2}(P)$ *given above and* $\sigma^{\#}_{2}(a) = \sigma^{\#}_{1}(a)$ *and* $\sigma^{\#}_{2}(b) = \sigma^{\#}_{1}(b)$*.*

Line 3. `c = a + b` *specifies a dependent variable* c*. The Interval Cube of* c *and the corresponding Density Cube after the statement will be:*

$(\llbracket \texttt{c=a+b} \rrbracket \sigma^{\#}_{2})(c) =$ (4.8)

[0, 2]	[1, 3]	[2, 4]	[3, 5]
[1, 3]	[2, 4]	[3, 5]	[4, 6]
[2, 4]	[3, 5]	[4, 6]	[5, 7]
[3, 5]	[4, 6]	[5, 7]	[6, 8]

$$([\![\texttt{c=a+b}]\!]\sigma_2^{\#})(P) = \begin{array}{|c|c|c|c|}\hline \texttt{1/16} & \texttt{1/16} & \texttt{1/16} & \texttt{1/16} \\ \hline \texttt{1/16} & \texttt{1/16} & \texttt{1/16} & \texttt{1/16} \\ \hline \texttt{1/16} & \texttt{1/16} & \texttt{1/16} & \texttt{1/16} \\ \hline \texttt{1/16} & \texttt{1/16} & \texttt{1/16} & \texttt{1/16} \\ \hline \end{array}, \tag{4.9}$$

where in Interval Cube the interval at column i and row j is calculated from the i-th interval of $\sigma_2^{\#}(a)$ and j-th interval of $\sigma_2^{\#}(b)$. For example, `[2, 4]` *at column 3 row 1 is from $a \in$* `[2, 3]` *and $b \in$* `[0, 1]`*, by adding the lower and upper bounds respectively. Then, each element of $([\![\texttt{c=a+b}]\!]\sigma_2^{\#})(c)$ has its corresponding probability density in $([\![\texttt{c=a+b}]\!]\sigma_2^{\#})(P)$ at the same position: for example,* `[2, 4]` *at column 3 row 1 has the corresponding density in $([\![\texttt{c=a+b}]\!]\sigma_2^{\#})(P)$ at column 3 row 1 being* `1/16`*. Let the result state be $\sigma_3^{\#}$ with $\sigma_3^{\#}(c)$ given above and $\sigma_3^{\#}(P) = \sigma_2^{\#}(P)$, $\sigma_3^{\#}(a) = \sigma_2^{\#}(a)$, and $\sigma_3^{\#}(b) = \sigma_2^{\#}(b)$.*

Line 4. `observe(Normal(c, 1), 5)` *means that the observed data (which is* `5`*) follows a Normal distribution with mean being c and variance 1. In Bayesian terminology, it defines a likelihood function as Normal_pdf$(5, c, 1)$. To simplify the notation, we let*

$$lik(c) = Normal_pdf(5, c, 1) = \frac{1}{\sqrt{2\pi}} e^{-\frac{(5-c)^2}{2}} \tag{4.10}$$

Since c can take multiple intervals, the result Density Cube is

$$([\![\texttt{observe(Normal(c, 1), 5)}]\!]\sigma_3^{\#})(P) =$$

$$\begin{array}{|c|c|c|c|}\hline \texttt{1/16*}lik(0) & \texttt{1/16*}lik(1) & \texttt{1/16*}lik(2) & \texttt{1/16*}lik(3) \\ \hline \texttt{1/16*}lik(1) & \texttt{1/16*}lik(2) & \texttt{1/16*}lik(3) & \texttt{1/16*}lik(4) \\ \hline \texttt{1/16*}lik(2) & \texttt{1/16*}lik(3) & \texttt{1/16*}lik(4) & \texttt{1/16*}lik(5) \\ \hline \texttt{1/16*}lik(3) & \texttt{1/16*}lik(4) & \texttt{1/16*}lik(5) & \texttt{1/16*}lik(6) \\ \hline \end{array}, \tag{4.11}$$

where each entry is by replacing c with the lower bound of each interval in $\sigma_3^{\#}(c)$. Note that the density is accurate at the lower bound of each interval, e.g. when $a = 2$ (the lower bound of the third interval in $\sigma_3^{\#}(a)$) and $b = 0$ (the lower bound of the first interval in $\sigma_3^{\#}(b)$) the result density will be exactly `1/16*lik(2)` *(the element in the third column and first row in the result Density Cube). Using the density for other values in the interval may result in a bounded error. The error can go to 0 when the number of splits goes to infinity (Theorem 4.1). Let the result state be $\sigma_4^{\#}$ with $\sigma_4^{\#}(P)$ given above and other variables unchanged.*

Line 5. `return a, b` *will let AQUA output the normalized joint density and marginalized densities of a and b. AQUA will first approximate the unnormalized joint density with a*

function $\hat{f}(a,b)$, which is constructed by the piecewise constant interpolation (Definition 4.6) of $\sigma_4^{\#}(P)$. Recall the columns and rows of $\sigma_4^{\#}(P)$ correspond to the values of a and b stored in $\sigma_4^{\#}(a)$ and $\sigma_4^{\#}(b)$. Then AQUA normalize $\hat{f}(a,b)$ to get the normalized joint density. To get the marginalized densities of a, AQUA integrates $\hat{f}(a,b)$ over b. This is equivalent to summing up $\sigma_4^{\#}(P)$ along the rows, interpolating and normalizing. Similarly, to get the marginalized densities of b, AQUA integrates $\hat{f}(a,b)$ over a, which is equivalent to summing up $\sigma_4^{\#}(P)$ along the columns, interpolating and normalizing.

4.3.3 Formal Guarantee of Accuracy

In this section we formally derive how well the symbolic state $\sigma^{\#}$ approximates the joint unnormalized density function f and the posterior density function p. To simplify the presentation, we use $\underline{\boldsymbol{x}}^{(\boldsymbol{m})} = [\underline{\boldsymbol{I}_{M_1}^{x_1}(\boldsymbol{m})}, \ldots, \underline{\boldsymbol{I}_{M_N}^{x_N}(\boldsymbol{m})}]$ for all variables, and analogously for $\overline{\boldsymbol{x}}^{(\boldsymbol{m})}$.

Definition 4.8. We write $\gamma(\sigma^{\#}) =_Q f$ if $\sigma^{\#}(P)(\boldsymbol{m}) = f(\boldsymbol{x})$ when $\boldsymbol{x} = \underline{\boldsymbol{x}}^{(\boldsymbol{m})}$, which means the abstract transformers are exact at the lower bounds. Further, if f is μ-Lipschitz continuous, namely $\mid f(\boldsymbol{x}_1) - f(\boldsymbol{x}_2) \mid \leq \mu \|\boldsymbol{x}_1 - \boldsymbol{x}_2\|$, we write $\gamma(\sigma^{\#}) =_Q^{\mu} f$.

In Lemma 4.1, to allow the posterior distributions to be Lipschitz continuous, we put an additional restriction on our programs: if an expression follows a discrete distribution, it must be a branching condition, e.g. `b ~ Uniform(0,1); if (b > 0.5){...}` where `b > 0.5` is discrete. The result posterior distribution of the statement will be a mixture of conditional distributions and thus is continuous. For example, it is acceptable to have the conditional statement `if (b > 0.5) {x ~ Normal(0,1)} else {x ~ Normal(1,1)}`. We do not provide formal guarantee for the programs with discrete distributions elsewhere, nevertheless, AQUA may still run on those programs and give results with small error.

Lemma 4.1 (Discretization Error)**.** The error of discretization is $\mid \hat{f}(\boldsymbol{x}) - f(\boldsymbol{x}) \mid \leq \mu \cdot \max_{\boldsymbol{m}} \|\overline{\boldsymbol{x}}^{(\boldsymbol{m})} - \underline{\boldsymbol{x}}^{(\boldsymbol{m})}\|$ if $\boldsymbol{x} \neq \underline{\boldsymbol{x}}^{(\boldsymbol{m})}$, and if $\boldsymbol{x} = \underline{\boldsymbol{x}}^{(\boldsymbol{m})}$ the error is 0.

Lemma 4.1 shows that at any program point, the error is bounded if we use the analysis result $\gamma(\sigma^{\#}) = \hat{f}$ as an approximation of joint density function f, and the error will reduce when the number of intervals is increased.

Proof of Lemma 4.1. We need to show that $\gamma(\sigma^{\#}) =_Q^{\mu} f$ at any program point. The proof is by structural induction on Expressions and Statements.

First we prove for all expressions E, $(\llbracket E \rrbracket^{\#}\sigma^{\#})(\boldsymbol{m}) = [(\llbracket E \rrbracket\sigma)(\underline{\boldsymbol{x}}^{(\boldsymbol{m})}), (\llbracket E \rrbracket\sigma)(\overline{\boldsymbol{x}}^{(\boldsymbol{m})})]$, and further $\llbracket E \rrbracket\sigma$ about variable $\boldsymbol{x}$ is μ_E-Lipschitz continuous if $\llbracket E \rrbracket\sigma$ is not the condition in if-then-else. We assume the function $\llbracket _ \rrbracket$ binding σ and $\llbracket _ \rrbracket^{\#}$ binding $\sigma^{\#}$ have the highest precedence, so hereon we omit the parentheses around them. The proof is by structural induction on ***expressions:***

Base case for expressions: for constants, $[\![c]\!]^{\#}\sigma_0^{\#}(\cdot) = [c, c] = [[\![c]\!]\sigma_0, [\![c]\!]\sigma_0]$, where $[\![c]\!]\sigma_0 = c$ is a constant and thus is 0-Lipschitz continuous. For variables, $[\![x]\!]^{\#}\sigma_0^{\#}(\boldsymbol{m}) = [\underline{I}^x, \overline{I^x}] = [[\![x]\!]\sigma_0(\underline{\boldsymbol{x}}^{(\boldsymbol{m})}), [\![x]\!]\sigma_0(\overline{\boldsymbol{x}}^{(\boldsymbol{m})})]$ and $[\![x]\!]\sigma_0$ about $\boldsymbol{x}$ is 1-Lipschitz continuous.

Inductive steps for expressions:

1. $[\![E_1[E_2]]\!]^{\#}$: E_2 must be evaluated to a constant, in the form of $[c, c]$. Then $[\![E_1[E_2]]\!]^{\#} = [\![E_1[c]]\!]^{\#}$, and we evaluate $E_1[c]$ in $\sigma^{\#}$ as an individual variable.

2. $[\![E_1 \ op \ E_2]\!]^{\#}\sigma_0^{\#}$: each element in the result Interval Cube is $[\![E_1 \ op \ E_2]\!]^{\#}\sigma_0^{\#}(\boldsymbol{m}) = [\underline{\boldsymbol{I}}^{E_1} \ op \ \underline{\boldsymbol{I}}^{E_2}, \overline{\boldsymbol{I}^{E_1}} \ op \ \overline{\boldsymbol{I}^{E_2}}] = [[\![E_1]\!]\sigma_0(\underline{\boldsymbol{x}}^{(\boldsymbol{m})}) \ op \ [\![E_2]\!]\sigma_0(\underline{\boldsymbol{x}}^{(\boldsymbol{m})}), [\![E_1]\!]\sigma_0(\overline{\boldsymbol{x}}^{(\boldsymbol{m})}) \ op \ [\![E_2]\!]\sigma_0(\overline{\boldsymbol{x}}^{(\boldsymbol{m})})]$. Because $[\![E_1]\!]\sigma_0$ and $[\![E_2]\!]\sigma_0$ about $\boldsymbol{x}$ are Lipschitz continuous (by inductive hypothesis), and *op* is one of +,-,*, $[\![E_1 \ op \ E_2]\!]\sigma_0$ about $\boldsymbol{x}$ is Lipschitz continuous. If *op* is /, we only allow E_2 to be a non-zero constant. If *op* is >, we represent the result *True* and *False* with 0 and 1 respectively. If $[\![E_1 > E_2$ is not the condition in the conditional statements, we require the operands of > to be independent of $\boldsymbol{x}$ and thus be Lipschitz continuous.

3. $[\![d(E_1,\ldots).\text{pdf}(E_n)]\!]^{\#}\sigma_0^{\#}$: Let $\boldsymbol{I}_{M_i}^{E_i} = [\![E_i]\!]^{\#}\sigma_0^{\#}$ for $i \in \{1, 2, \ldots, n\}$, and $[\![d(E_1, \ldots).\text{pdf} \ (E_n)]\!]^{\#} \ \sigma_0^{\#} = \boldsymbol{I}_{\boldsymbol{K}}^{\text{dpdf}}$. Then each element in the Interval Cube $\boldsymbol{I}_{\boldsymbol{K}}^{\text{dpdf}}(\boldsymbol{k}) = [d_\text{pdf} \ (\underline{\boldsymbol{I}_{M_n}^{E_n}(\boldsymbol{m}_n)}, \underline{\boldsymbol{I}_{M_1}^{E_1}(\boldsymbol{m}_1)}, \ldots), d_\text{pdf} \ (\overline{\boldsymbol{I}_{M_n}^{E_n}(\boldsymbol{m}_n)}, \overline{\boldsymbol{I}_{M_1}^{E_1}(\boldsymbol{m}_1)}, \ldots)] = [d_\text{pdf} \ ([\![E_n]\!]\sigma_0(\underline{\boldsymbol{x}}^{(\boldsymbol{k})}), [\![E_1]\!]\sigma_0(\underline{\boldsymbol{x}}^{(\boldsymbol{k})}), \ldots), d_\text{pdf} \ ([\![E_n]\!]\sigma_0(\overline{\boldsymbol{x}}^{(\boldsymbol{k})}), [\![E_1]\!]\sigma_0(\overline{\boldsymbol{x}}^{(\boldsymbol{k})}), \ldots)] = [[\![d_\text{pdf}(E_n, E_1, \ldots)]\!]\sigma_0(\underline{\boldsymbol{x}}^{(\boldsymbol{k})}), [\![d_\text{pdf}(E_n, E_1, \ldots)]\!] \sigma_0(\overline{\boldsymbol{x}}^{(\boldsymbol{k})})]$. For all the distributions in this language, $d_$pdfs about $\boldsymbol{x}$ are Lipschitz continuous.

Then we prove the lemma for ***statements:*** as the *base case*, we initialize $f(\cdot) = 1$, and $\sigma^{\#} (P)(\cdot) = 1$, so $\gamma(\sigma^{\#}) =_Q^0 f$ (f is 0-Lipschitz continuous). By applying the function *GetInit-Intervals* in Algorithm 4.1, we have the initial splits for all variables $\boldsymbol{x}$, and $\sigma^{\#}(P)(\boldsymbol{m}) = 1$ for any $\boldsymbol{m}$ as the index of density cube. Then $(\sigma^{\#}) =_Q^0 f$. Suppose $\gamma(\sigma_0^{\#}) =_Q^{\mu_0} f_0$ holds before statement S. In inductive steps we prove $\gamma(\sigma_1^{\#}) =_Q^{\mu_1} f_1$ where $\sigma_1^{\#} = [\![S]\!]^{\#}\sigma_0^{\#}$, and $f_1 = ([\![S]\!]\sigma)(\mathcal{L})$ is true density function after the statement.

We apply structural induction on statements:

1. $[\![\text{skip}]\!]^{\#}$: $\sigma_1^{\#}(\boldsymbol{x}) = \sigma_0^{\#}(\boldsymbol{x})$, $f_0(\boldsymbol{x}) = f_1(\boldsymbol{x}) \implies \gamma(\sigma_1^{\#}) =_Q^{\mu_1} f_1$ where $\mu_1 = \mu_0 < \infty$.

2. $[\![x = E]\!]^{\#}$: $\sigma_1^{\#}(P) = \sigma_0^{\#}(P)$ since this assign is deterministic and does not change the density cube, so $f_0 = f_1$. Because $\gamma(\sigma_0^{\#}) =_Q^{\mu_0} f_0$, $\gamma(\sigma_1^{\#}) =_Q^{\mu_1} f_1$.

3. $[\![S_1; S_2]\!]^{\#}$: Let σ_0 be the concrete state with variable $\boldsymbol{x}$ before the statement, and let σ_1 be the concrete state after the statement, namely $\sigma_0(\mathcal{L}) = f_0(\boldsymbol{x})$ and $\sigma_1(\mathcal{L}) = f_1(\boldsymbol{x})$. Then, since $\gamma(\sigma_0^{\#}) =_Q^{\mu_0} f_0$, by inductive hypothesis, $([\![S_1]\!]^{\#}\sigma_0^{\#})(P)(\boldsymbol{m}) = f_1(\underline{\boldsymbol{x}}^{(\boldsymbol{m})}) = ([\![S_1]\!]\sigma_0)(\mathcal{L})(\underline{\boldsymbol{x}}^{(\boldsymbol{m})})$, and f_1 is μ_1-Lipschitz continuous. Let $\sigma_1^{\#} = [\![S_1]\!]^{\#}\sigma_0^{\#}$, then $\gamma(\sigma_1^{\#}) =_Q^{\mu_1} f_1$. Apply inductive hypothesis again, $([\![S_2]\!]^{\#}\sigma_1^{\#})(P)(\boldsymbol{m}) = f_2(\underline{\boldsymbol{x}}^{(\boldsymbol{m})}) = ([\![S_2]\!]\sigma_1)(\mathcal{L})(\underline{\boldsymbol{x}}^{(\boldsymbol{m})})$ and f_2 is μ_2-Lipschitz continuous. So $\gamma(\sigma_2^{\#}) =_Q^{\mu_2} f_2$.

4. $[\![\text{factor}(E)]\!]^{\#}$: In concrete semantics, let $([\![E]\!]\sigma_0)$ be the result of evaluating E. The

true density function is derived as $f_1(\boldsymbol{x}) = f_0(\boldsymbol{x}) \cdot ((\llbracket E \rrbracket \sigma_0)(\boldsymbol{x}))$. In symbolic semantics, $\llbracket E \rrbracket^{\#} \sigma_0^{\#}$ is the interval cube where $(\llbracket E \rrbracket^{\#} \sigma_0^{\#})(\boldsymbol{m}) = [(\llbracket E \rrbracket \sigma_0)(\underline{\boldsymbol{x}}^{(\boldsymbol{m})}), (\llbracket E \rrbracket \sigma_0)(\overline{\boldsymbol{x}}^{(\boldsymbol{m})})]$. Then $\sigma_1^{\#}(P)(\boldsymbol{m}) = \sigma_0^{\#}(P)(\boldsymbol{m}) \cdot ((\llbracket E \rrbracket \sigma_0)(\underline{\boldsymbol{x}}^{(\boldsymbol{m})})) \implies \sigma_1^{\#}(P)(\boldsymbol{m}) = f_1(\underline{\boldsymbol{x}}^{(\boldsymbol{m})})$. To show f_1 is μ_1-Lipschitz continuous, $\mid f_1(\boldsymbol{x}_1) - f_1(\boldsymbol{x}_2) \mid \leq \mu_0 \|\boldsymbol{x}_2 - \boldsymbol{x}_1\| \mid f_1(\boldsymbol{x}_1) \mid + \mid f_0(\boldsymbol{x}_1) - f_0(\boldsymbol{x}_2) \|\, f_0(\boldsymbol{x}_2) \mid$ $= (\mu_0 \mid f_1(\boldsymbol{x}_1) \mid + \mu_E \mid f_0(\boldsymbol{x}_2) \mid) \, \|\boldsymbol{x}_2 - \boldsymbol{x}_1\| = \mu_1 \|\boldsymbol{x}_2 - \boldsymbol{x}_1\|$, where μ_E is the Lipschitz constant of the expression $\llbracket E \rrbracket$ (see the proof for expressions below). This implies $\gamma(\sigma_1^{\#}) =_Q^{\mu_1} f_1$.

5. $\llbracket \text{observe}(d(E_1, \ldots, E_{n-1}), E_n) \rrbracket^{\#}$: by $\llbracket \text{factor}(E) \rrbracket^{\#}$ rule above.

6. $\llbracket x \sim d(E_1, \ldots, E_n) \rrbracket^{\#}$: First, we evaluate the expression $\llbracket d(E_1, \ldots, E_n).\text{pdf}(x) \rrbracket^{\#} \sigma_0^{\#}$ to get $\boldsymbol{I}_{\boldsymbol{K}}^{\text{dpdf}}$ (see the proof for expressions below). According to the analysis rule in Figure 4.4, $\sigma_1^{\#}(P)(\boldsymbol{m}) = \sigma_0^{\#}(P)(\boldsymbol{m}_0) \cdot \underline{\boldsymbol{I}_{\boldsymbol{K}}^{\text{dpdf}}(\boldsymbol{k})}$. Also, $f_1(\boldsymbol{x}) = f_0(\boldsymbol{x}) \cdot (\llbracket d(E_1, \ldots, E_n).\text{pdf}(x) \rrbracket \sigma_0)(\boldsymbol{x})$. By inductive hypothesis, $\sigma_0^{\#}(P)(\boldsymbol{m}) = f_0(\underline{\boldsymbol{x}}^{(\boldsymbol{m})})$, so $\sigma_1^{\#}(P)(\boldsymbol{m}) = \sigma_0(\underline{\boldsymbol{x}}^{(\boldsymbol{m})}) \cdot \underline{\boldsymbol{I}_{\boldsymbol{K}}^{\text{dpdf}}(\boldsymbol{k})} = f_1(\boldsymbol{x})$, which implies $\gamma(\sigma_1^{\#}) =_Q^{\mu_1} f_1$, where μ_1 is the Lipschitz constant of f_1. To prove f_1 is μ_1-Lipschitz continuous, $\mid f(\boldsymbol{x}_1) - f(\boldsymbol{x}_2) \mid \leq \mid f(\boldsymbol{x}_1) \cdot \text{d_pdf}(\boldsymbol{x}_1) - f(\boldsymbol{x}_2) \cdot \text{d_pdf}(\boldsymbol{x}_2) \mid \leq \mu_0 \cdot \mid \text{d_pdf}(\boldsymbol{x}_1) \mid + \mid \text{d_pdf}(\boldsymbol{x}_1) - \text{d_pdf}(\boldsymbol{x}_2) \|\, f_0(\boldsymbol{x}_2) \mid = \mu_1 < \infty$. Therefore, $\gamma(\sigma_1^{\#}) =_Q^{\mu_1} f_1$.

7. $\llbracket \text{if } (E) \text{ then } S_1 \text{ else } S_2 \rrbracket^{\#} \sigma_0^{\#} = (\llbracket \text{factor}(E); S_1 \rrbracket \sigma_0^{\#}) \sqcup (\llbracket \text{factor}(1 - E); S_2 \rrbracket \sigma_0^{\#})$. After evaluating the statement in the true branch we get $\gamma(\sigma_T^{\#}) =_Q^{\mu_T} f_T$, and after evaluating the false branch we get $\gamma(\sigma_F^{\#}) =_Q^{\mu_F} f_F$. Then $(\sigma_T^{\#} \sqcup \sigma_F^{\#})(P)(\boldsymbol{m}) = \sigma_T^{\#}(P)(\boldsymbol{m}) + \sigma_F^{\#}(P)(\boldsymbol{m})$ $= f_T(\underline{\boldsymbol{x}}^{(\boldsymbol{m})}) + f_F(\underline{\boldsymbol{x}}^{(\boldsymbol{m})})$. Also, $\mid f_1(\boldsymbol{x}_1) - f_1(\boldsymbol{x}_2) \mid = \mid f_T(\boldsymbol{x}_1) + f_F(\boldsymbol{x}_1) - (f_T(\boldsymbol{x}_2) + f_F(\boldsymbol{x}_2)) \mid$ $\leq \mu_T \mid \boldsymbol{x}_1 - \boldsymbol{x}_2 \mid + \mu_F \mid \boldsymbol{x}_1 - \boldsymbol{x}_2 \mid = (\mu_T + \mu_F) \|\boldsymbol{x}_1 - \boldsymbol{x}_2\| = \mu_1 \|\boldsymbol{x}_1 - \boldsymbol{x}_2\|$. Therefore, $\gamma(\sigma_1^{\#}) =_Q^{\mu_1} f_1$.

8. $\llbracket \text{for } (i \text{ in } E_1..E_2)\ S \rrbracket^{\#}$ is reduced to if-then-else and sequencing. Thus the property holds. QED.

The error of AQUA's approximation to the normalizing constant z is also bounded:

Lemma 4.2 (Integration Error)**.** Let $U = \prod_{i=1}^{N} (b_i - a_i)$ be the volume of $\boldsymbol{C}$. For all the probability distributions supported in our language, the error is $|z - \hat{z}| \leq U \mu \max_{\boldsymbol{m}} \|\overline{\boldsymbol{x}}^{(\boldsymbol{m})} - \underline{\boldsymbol{x}}^{(\boldsymbol{m})}\|$. If we use M equal-length intervals for each variable, $|z - \hat{z}| \leq U \mu \frac{1}{M} (\sum_{i=1}^{N} (b_i - a_i)^2)^{\frac{1}{2}}$. Then $|z - \hat{z}| \to 0$ as $M \to \infty$.

Proof of Lemma 4.2. Recall, all posteriors f in our language (Section 4.2) are Lipschitz continuous. We derive the error bound by applying the Lipschitz continuous property of f and the triangle inequality. First, $\hat{z} = \int_{\boldsymbol{C}} \hat{f}(\boldsymbol{x}) d\boldsymbol{x} = \sum_{\boldsymbol{m}}^{\boldsymbol{M}} \left(\prod_i^N (\overline{\boldsymbol{x}}^{(\boldsymbol{m})} - \underline{\boldsymbol{x}}^{(\boldsymbol{m})}) \cdot p^{(\boldsymbol{m})} \right)$. According to Lemma 8, $\hat{f}$ is a quantization of f, meaning $\hat{f}(\boldsymbol{x}) = f(\boldsymbol{x})$ at the points

$\boldsymbol{x} = \underline{\boldsymbol{x}}^{(\boldsymbol{m})}$, while for other $\boldsymbol{x} \in (\underline{\boldsymbol{x}}^{(\boldsymbol{m})}, \overline{\boldsymbol{x}}^{(\boldsymbol{m})})$, $\hat{f}(\boldsymbol{x}) = f(\underline{\boldsymbol{x}}^{(\boldsymbol{m})})$. Then

$$\begin{aligned}
|z - \hat{z}| &= |\int_{\boldsymbol{C}} f(\boldsymbol{x})d\boldsymbol{x} - \int_{\boldsymbol{C}} \hat{f}(\boldsymbol{x})d\boldsymbol{x}| && \text{(Error term)} && (4.12)\\
&\leq \int_{\boldsymbol{C}} |f(\boldsymbol{x}) - \hat{f}(\boldsymbol{x})|d\boldsymbol{x} && \text{(Triangle ineq.)} && (4.13)\\
&\leq U \cdot \max_{\boldsymbol{x}} |f(\boldsymbol{x}) - \hat{f}(\boldsymbol{x})| && (U \text{ is volume of } \boldsymbol{C}) && (4.14)
\end{aligned}$$

Then we prove $\max_{\boldsymbol{x}} |f(\boldsymbol{x}) - \hat{f}(\boldsymbol{x})| \leq \mu \max_{\boldsymbol{m}} \|\overline{\boldsymbol{x}}^{(\boldsymbol{m})} - \underline{\boldsymbol{x}}^{(\boldsymbol{m})}\|$. For each interval box $[\boldsymbol{x}^{(\boldsymbol{m})}]$, $f(\boldsymbol{x}) - \hat{f}(\boldsymbol{x}) = f(\boldsymbol{x}) - f(\underline{\boldsymbol{x}}^{(\boldsymbol{m})})$. Because f is μ-Lipschitz continuous, $|f(\boldsymbol{x}) - f(\underline{\boldsymbol{x}}^{(\boldsymbol{m})})| \leq \mu\|\boldsymbol{x} - \underline{\boldsymbol{x}}^{(\boldsymbol{m})}\| \leq \mu \max_{\boldsymbol{m}} \|\overline{\boldsymbol{x}}^{(\boldsymbol{m})} - \underline{\boldsymbol{x}}^{(\boldsymbol{m})}\|$. QED.

Moreover, the integration error bound above will decrease when we decrease the interval length, or increase the number of intervals. Then at the end of the analysis, we approximate the *posterior probability density function* $p(\boldsymbol{x})$ on $\boldsymbol{C}$ as:

Definition 4.9 (Posterior Probability Density Approximation)**.** Define $\hat{p}(\boldsymbol{x}) = \frac{1}{\hat{z}}\hat{f}(\boldsymbol{x})$ as the approximation of $p(\boldsymbol{x})$.

Now we show the end-to-end error of the analysis. As Theorem 4.1 states, by applying sufficiently many intervals, the random variables following AQUA's posterior estimation in $\boldsymbol{C}$ will *converge in distribution* to the true posterior in $\boldsymbol{C}$. Without loss of generality, suppose we apply at least M equal-length intervals for each variable in its domain $[a_i, b_i]$, i.e. $M = \min\{M_1, M_2, \ldots, M_N\}$. And we refer $\hat{p}_M(\boldsymbol{x})$ as AQUA's approximation of $p(\boldsymbol{x})$ by applying at least M equal-length intervals for each variable.

Theorem 4.1 (Convergence of Posterior Density Approximation)**.** Define $F_{\boldsymbol{C}}(\boldsymbol{x}) = \frac{1}{z}\int_{-\infty}^{\boldsymbol{x}} \mathbf{1}_{[\boldsymbol{u}\in\boldsymbol{C}]} \cdot p(\boldsymbol{u})d\boldsymbol{u}$ as the true cumulative distribution function (CDF) on $\boldsymbol{C}$, where $z = \int_{\boldsymbol{C}} p(\boldsymbol{x})d\boldsymbol{x}$, and $\hat{F}_{\boldsymbol{C},M}(\boldsymbol{x}) = \int_{-\infty}^{\boldsymbol{x}} \hat{p}_M(\boldsymbol{u})d\boldsymbol{u}$ as the approximate CDF. Then

$$\lim_{M\to\infty} \hat{F}_{\boldsymbol{C},M}(\boldsymbol{x}) = F_{\boldsymbol{C}}(\boldsymbol{x}). \tag{4.15}$$

Proof of Theorem 4.1. Recall $\boldsymbol{C} = \bigotimes_{i=1}^{N}[a_i, b_i]$, so $\boldsymbol{a}$ is the lower bound of $\boldsymbol{C}$. Given Lemma 4.1 for the quantization error $|\hat{f}(\boldsymbol{x}) - f(\boldsymbol{x})| \leq \mu \max_{\boldsymbol{m}} \|\overline{\boldsymbol{x}}^{(\boldsymbol{m})} - \underline{\boldsymbol{x}}^{(\boldsymbol{m})}\|$, we know $|\int_{\boldsymbol{a}}^{\boldsymbol{x}} \hat{f}(\boldsymbol{u}) - f(\boldsymbol{u})d\boldsymbol{u}| \leq \int_{\boldsymbol{a}}^{\boldsymbol{x}} |\hat{f}(\boldsymbol{u}) - f(\boldsymbol{u})|d\boldsymbol{u} \leq \theta\mu \max_{\boldsymbol{m}} \|\overline{\boldsymbol{x}}^{(\boldsymbol{m})} - \underline{\boldsymbol{x}}^{(\boldsymbol{m})}\|$, where $\theta = \|\boldsymbol{x} - \boldsymbol{a}\|$. Using M equal-length splits for each variables, we can write $\max_{\boldsymbol{x}} \|\overline{\boldsymbol{x}}^{(\boldsymbol{m})} - \underline{\boldsymbol{x}}^{(\boldsymbol{m})}\| = \sqrt{\sum_{i=1}^{N} \left(\frac{b_i - a_i}{M}\right)^2} = \frac{h}{M}$ where $h = \sqrt{\sum_{i=1}^{N} (b_i - a_i)^2}$ is a constant.

By combining the error bounds in Lemma 4.1 and Lemma 4.2 and applying triangle

inequality (Tri. ineq.), we show the convergence of the end-to-end error:

$$|\hat{F}_{\boldsymbol{C},t}(\boldsymbol{x}) - F_{\boldsymbol{C}}(\boldsymbol{x})| \tag{4.16}$$

$$= |\frac{\int_{-\infty}^{\boldsymbol{x}} \hat{p}(\boldsymbol{u})d\boldsymbol{u}}{\int_{\boldsymbol{C}} \hat{p}(\boldsymbol{x})d\boldsymbol{x}} - \frac{\int_{-\infty}^{\boldsymbol{x}} \mathbf{1}_{[\boldsymbol{u}\in\boldsymbol{C}]}p(\boldsymbol{u})d\boldsymbol{u}}{\int_{\boldsymbol{C}} p(\boldsymbol{x})d\boldsymbol{x}}| \quad \text{(Definition 4.9)} \tag{4.17}$$

$$= |\frac{\int_{\boldsymbol{a}}^{\boldsymbol{x}} \hat{f}(\boldsymbol{u})d\boldsymbol{u}}{\hat{z}} - \frac{\int_{\boldsymbol{a}}^{\boldsymbol{x}} f(\boldsymbol{u})d\boldsymbol{u}}{z}| \quad \text{(Definition 4.7)} \tag{4.18}$$

$$\leq \frac{z|\int_{\boldsymbol{a}}^{\boldsymbol{x}} \hat{f}(\boldsymbol{u}) - f(\boldsymbol{u})d\boldsymbol{u}| + \int_{\boldsymbol{a}}^{\boldsymbol{x}} f(\boldsymbol{u})d\boldsymbol{u}|z-\hat{z}|}{\hat{z}z} \quad \text{(Tri. ineq.)} \tag{4.19}$$

$$\leq \frac{z(\theta\mu h/M) + \int_{\boldsymbol{a}}^{\boldsymbol{x}} f(\boldsymbol{u})d\boldsymbol{u}|z-\hat{z}|}{\hat{z}z} \quad \text{(Lemma 4.1)} \tag{4.20}$$

$$\leq \frac{z(\theta\mu h/M) + \int_{\boldsymbol{a}}^{\boldsymbol{x}} f(\boldsymbol{u})d\boldsymbol{u}(U\mu h/M)}{\hat{z}z} \quad \text{(Lemma 4.2)} \tag{4.21}$$

$$\leq \frac{z\theta\mu h + F_{\boldsymbol{C}}(\boldsymbol{x})U\mu h}{M \cdot \hat{z}z} \tag{4.22}$$

$$\to 0 \qquad \text{as } M \to \infty \tag{4.23}$$

Then θ, μ (Lipschitz constant of f), z (normalizing constant), U (volume of $\boldsymbol{C}$), and $F_{\boldsymbol{C}}(\boldsymbol{x})$ are all constants regarding M, and $\hat{z} \to z > 0$ as $M \to \infty$. Hence $|\ \hat{F}_{\boldsymbol{C},t}(\boldsymbol{x}) - F_{\boldsymbol{C}}(\boldsymbol{x})\ | \to 0$ as $M \to \infty$. QED.

We allow a user to provide a bounded domain $\boldsymbol{C}$, or infer it with automatically with a heuristic (Section 4.4). Although AQUA's formal guarantee is in a bounded domain, it can give runtime warnings when any prior or likelihood has probability greater than a given threshold on the rest of the domain $\mathbb{R}^N - \boldsymbol{C}$. If AQUA does not give any warning, the final error caused by truncating infinite domain into $\boldsymbol{C}$ will be smaller than the threshold.

4.4 AQUA OPTIMIZATIONS

Adaptive Intervals. To find the suitable bounded intervals $\boldsymbol{C} = [C_1, C_2, \ldots, C_N]$ that cover most probability, we design a adaptive algorithm (Algorithm 4.2) to adjust $\boldsymbol{C}$ the based on the result from last run.

Algorithm 4.2 takes as inputs the program, the vector of number of intervals, and two thresholds t_0 and t_{dist} for deciding the interval bounds $\boldsymbol{C}$. Increasing C_i or increasing the number of intervals in C_i will help reduce the approximation error.

The function *GetInitBounds* (Line 2) takes the prior distribution of each x_i as a rough estimate of the distribution to determine an initial interval split. If the domain of the prior distribution is bounded in $[a_i, b_i]$ where $-\infty < a_i < b_i < \infty$, e.g. $\texttt{x}_\texttt{i} \sim \texttt{Uniform(a,b)}$, AQUA

Table 4.2: Program Description and Characteristics

	Description	Distributions	#D	#N
prior_mix	Mixture model[47]	$B\times(N+N)\times T^{10}$	10	1
zeroone	Bayesian neural network[131]	$U^2\times M^{20}$	20	2
tug	Causal cognition model[132]	$U^2\times(N+N)^2\times B^{40}$	40	2
altermu	Model with param symmetry[133]	$N^3\times N^{40}$	40	3
altermu2	Model with param symmetry[133]	$U^2\times N^{40}$	40	2
neural	Bayesian neural network[134]	$U^2\times(B\times M)^{39}$	39	2
normal_mixture	Mixture model with mixing rate[50]	$N^2\times Be\times(B\times(N+N))^{63}$	63	3
mix_asym_prior	Mixture model with scale params[50]	$N^2\times G^2\times(B\times(N+N))^{40}$	40	4
logistic	Logistic regression[50]	$U^2\times(B\times M)^{100}$	100	2
logistic_RW	Reweighted logistic regression[43, 50]	$U^2\times Be^{100}\times(B\times M)^{100}$	100	102
anova	Linear regression [50]	$U^2\times N^{40}$	40	2
anova_RP	Localized linear regression[44, 50]	$U^2\times G^{40}\times N^{40}$	40	42
anova_RW	Reweighted linear regression[43, 50]	$U^2\times Be^{40}\times N^{40}$	40	42
lightspeed	Linear regression[50]	$N\times U\times N^{66}$	66	2
lightspeed_RP	Localized linear regression[44, 50]	$N\times U\times G^{66}\times N^{66}$	66	68
lightspeed_RW	Reweighted linear regression[43, 50]	$N\times U\times Be^{66}\times N^{66}$	66	68
unemployment	Linear regression[50]	$N^2\times U\times N^{40}$	40	3
unemployment_RP	Localized linear regression[44, 50]	$N^2\times U\times G^{40}\times N^{40}$	40	43
unemployment_RW	Reweighted linear regression[43, 50]	$N^2\times U\times Be^{40}\times N^{40}$	40	43
timeseries	Timeseries analysis[50]	$U^3\times N^{39}$	39	3
gammaTransform	Transformed param[22]	G	0	3
GPA	Hybrid continuous & discrete distr.[135]	$B\times(B\times(A+U)+B\times(A+U))$	1	3
radar_query1	Bayesian network in robotics[20]	$B\times(A+B)\times U\times N\times(Tr+Tr)$	2	6
radar_query2	Bayesian network in robotics[20]	$B\times(A+B)\times U^2\times N\times Tr$	1	6

Distributions: A: Atomic, B: Bernoulli, Be: Beta, G: Gamma, M: Softmax, N: Normal, T: Student-T, Tr: Triangular, U: Uniform. '+' represents the mixture of two distributions, and '×' represents the product of the individual density functions in the joint probability density function.

divides $[a_i, b_i]$ into M_i equi-length intervals, each with length $(b_i - a_i)/M_i$, where M_i is given

Algorithm 4.2 Posterior Interval Analysis with Adaptive Interval

```
1:  procedure PosteriorAdaptiveAnalysis(Prog, M, t_0, t_dist)
2:      C ← GetInitBounds(Prog, t_0)                      ▷ C = [C_1, C_2, ..., C_N]
3:      changed ← True
4:      while changed do                                   ▷ Stop if C no longer changes
5:          (p̂, Marginal) ← PosteriorAnalysis(Prog, M, C)
6:          changed ← False
7:          for x_i ∈ SampledVars(Prog) do                  ▷ Adapt each C_i
8:              p̂_i(x_i) ← Marginal[x_i]
9:              if ∃x_i ∈ C_i, p̂_i(x_i) < t_dist then
10:                 a_i ← inf{x_i | p̂_i(x_i) > t_dist}
11:                 b_i ← sup{x_i | p̂_i(x_i) > t_dist}
12:                 C_i ← [a_i, b_i]
13:                 changed ← True
14: return (p̂, Marginal)
```

in $\boldsymbol{M}$ by the user. If the distribution is not bounded, e.g. $\texttt{x}_\texttt{i} \sim \texttt{Normal}(0, 1)$, the user can specify a threshold t_0 for AQUA to infer C_is such that values from the prior being out of C_is has probability smaller than t_0. Otherwise by default we set $t_0 = 4 \cdot 10^{-32}$.

In each iteration, the algorithm applies the analysis on the current $\boldsymbol{C}$ (line 5) and check if we need to adapt $\boldsymbol{C}$. We adapt $\boldsymbol{C}$ when any variable x_i has density value $\hat{p}_i(x_i)$ being almost about 0 – smaller than the user provided threshold t_{dist} (e.g. 10^{-8}) (line 8-12). We shrink C_i to focus on the smallest area with density greater than a given threshold t_{dist}. With the same number of intervals M_i, the smaller C_i will produce thiner intervals and result in more accurate results. Practically, this adaptive algorithm is as accurate but is much more efficient than naively increasing the number of intervals M_i on the whole initial domain C_i. Suppose the program takes A adaptive iterations, and it has N variables and each variable has the same number of intervals M, Algorithm 2 has the time complexity $\mathcal{O}(A \cdot N \cdot M^N)$ and the space complexity $\mathcal{O}(M^N)$. In our experiments, A is usually less than 5.

Improving Inference for Many Local Variables. In this optimization we change the analysis of statements in Section 4.3 to marginalize the local variables as soon as possible. Local variables are those defined and only used in local blocks (e.g. in for-loop and if-then-else from Figure 4.4).

By marginalizing out the local variables, we avoid repeatedly computing the joint density on the unused variables. For example, in a robust model one may naively calculate the joint density via $\hat{f}(x) = \prod_{i=1}^{D} \text{d_pdf}(x, \text{w}_i)$, where w_is are local variables defined in each loop body. This requires keeping a $(D{+}1)$-dimensional density cube to capture all the variables x and w_is. Instead, our optimization divides the above product into calculating the individual $\text{d_pdf}(x, \text{w}_i)$, when w_i leaves its scope, so we do not carry the current w_i to the next iteration. In each iteration we only operate on a 2-dimensional Density Cube for variables x and a single w_i. If out of N variables in the program D are local variables we will have a time complexity $\mathcal{O}(N \cdot M^{N-D})$ for Algorithm 4.1 (while the original is $\mathcal{O}(N \cdot M^N)$).

4.5 METHODOLOGY

We evaluate AQUA on 24 probabilistic programs collected from existing literature. We compare the execution time of AQUA on these programs with other probabilistic programming languages: Stan [8], PSI [20], and SPPL [22]. We implement AQUA in Java using ND4J library for tensor computation, and run all experiments on Intel Xeon 3.6 GHz machine with 6 cores and 32 GB RAM. For numerical stability, we use log probability/density (instead of original probability/density) for Density Cube.

Benchmarks. Table 4.2 presents the benchmarks obtained from the literature. Column **Description** summarizes the task of each program. Column **Distributions** shows the distributions of observable and latent variables. For example, the distributions in program "prior_mix" are one Bernoulli (B), one Mixture of two Normals ($N+N$), and 10 Student-T distributions (T^{10}). All posterior distributions are continuous. Column **#D** shows the number of data observations, **#N** shows the number of random variables in the program. The benchmarks "prior_mix", "normal_mixture", "mix_asym_prior", "GPA", "radar_query1" and "radar_query2" have discrete distributions in the if-then-else conditions, but the result posterior density functions are continuous and thus AQUA's formal guarantee holds. "zeroone" and "tug" have discrete distributions in the intermediate results and posterior densities, where AQUA cannot provide the formal guarantee. However, Section 4.6.1 gives empirical evidence that AQUA's result error is very small.

Comparing Posterior Distributions. The Kolmogorov-Smirnov (KS) statistic measures the distance between two probability distributions. We use the KS statistic for the accuracy evaluation in the analysis. Let F_{truth} and $\hat{F}$ denote the cumulative density functions of the posterior distributions of the variable x from the original input data and the noisy data respectively, the *KS statistic* is defined as $KS = \sup_x |F_{truth}(x) - \hat{F}(x)|$, namely, the maximum difference in the cumulative distribution functions. The KS statistic takes a value between 0 (most close distributions) and 1 (most different distributions). Therefore, smaller KS statistic implies better accuracy.

Experimental Setup. We manually derived the *ground truth* posterior distributions for all the programs. We run AQUA with the adaptive algorithm described in Section 4.4. We use the equal number of $M = \max\{60, \lceil 40000^{(1/N)} \rceil\}$ intervals for each variable, where N is the number of sampled variables, so that the total number intervals $M^N \geq 40000$. Rounding up the total number of intervals to 40000 does not significantly affect time but will guarantee more accurate results. We test Stan on its two major inference algorithms, NUTS (a variant of MCMC) and ADVI (a variant of variational inference). For fair comparison, we allow running VI/NUTS until it reaches the same accuracy level (in KS statistic) as AQUA and report the average time, or until it reaches the maximum iterations (fixed at 400000 for both VI and NUTS). We set the timeout to be 20 minutes for all the inference tools.

Table 4.3: Runtime Comparison for AQUA, Stan, PSI, and SPPL. Stan column shows time needed reach AQUA's accuracy.

Program	AQUA		Stan VI		Stan NUTS		PSI	SPPL
	Time(s)	Error	Time(s)	Error	Time(s)	Error	Time (s)	Time (s)
prior_mix	4.77	0.02	0.53	0.31	5.67	0.19	inte	⊘
zeroone	0.98	0.00	0.44	0.21	630.73	0.21	91.16	⊘
tug	0.83	0.01	1.20	0.25	519.94	0.06	inte	⊘
altermu	1.35	0.00	0.96	0.31	29.46	0.03	inte	⊘
altermu2	0.76	0.00	0.75	0.34	25.98	0.07	inte	⊘
neural	0.85	0.01	0.82	0.03	5.10	0.01	t.o.	⊘
normal_mixture	1.19	0.02	1.02	0.12	25.67	0.04	t.o.	⊘
mix_asym_prior	24.63	0.02	1.04	0.09	16.41	0.03	t.o.	⊘
logistic	0.99	0.02	0.74	0.07	17.31	0.02	t.o.	⊘
logistic_RW	1.87	0.01	15.37	0.09	72.45	0.02	t.o.	⊘
anova	0.90	0.01	0.75	0.07	6.72	0.02	inte	⊘
anova_RP	1.55	0.01	6.89	0.07	77.48	0.02	t.o.	⊘
anova_RW	1.40	0.01	6.93	0.06	24.67	0.02	t.o.	⊘
lightspeed	0.74	0.00	0.71	0.04	3.56	0.00	inte	⊘
lightspeed_RP	1.37	0.01	6.18	0.06	61.37	0.02	t.o.	⊘
lightspeed_RW	1.09	0.02	6.19	0.05	61.37	0.05	t.o.	⊘
unemployment	1.44	0.02	0.64	0.21	5.07	0.01	inte	⊘
unemployment_RP	42.34	0.01	6.78	0.25	12.46	0.01	t.o.	⊘
unemployment_RW	27.41	0.02	7.07	0.23	2.53	0.01	t.o.	⊘
timeseries	1.55	0.01	0.87	0.23	12.66	0.01	inte	⊘
gammaTransform	0.72	0.00	0.62	0.05	3.01	0.01	inte	1.30
GPA	0.46	0.02	⊘	⊘	⊘	⊘	0.12	0.05
radar_query1	0.87	0.01	⊘	⊘	⊘	⊘	inte	⊘
radar_query2	1.82	0.02	⊘	⊘	⊘	⊘	inte	⊘
Avg	5.08	0.01	3.17	0.15	77.12	0.04	⊘	⊘
Median	1.35	0.01	0.99	0.10	20.99	0.02	⊘	⊘

[*time*] : VI or NUTS takes more time than AQUA, or AQUA take more time than VI and NUTS.
[*error*] Has the error (in terms of a KS statistic) larger than 0.01 from the best solution.
"⊘": the PPL cannot work on the program. "t.o.": timeout, "inte": evaluates to unsolved integrals.

4.6 EVALUATION

4.6.1 Runtime/Accuracy Comparison

Table 4.3 presents the runtime and accuracy comparison of AQUA with Stan, PSI, and SPPL. Column **Program** shows the name of the probabilistic program. Columns **Time (s)** show the execution time (in seconds) of each tool, averaged across 5 runs. We report the total time for computing joint density and marginals for all sampled variables. Columns **Error** show the error (KS statistic, Section 4.5) of each tool vs. the ground truth when run

for the same time, averaged across 5 runs.

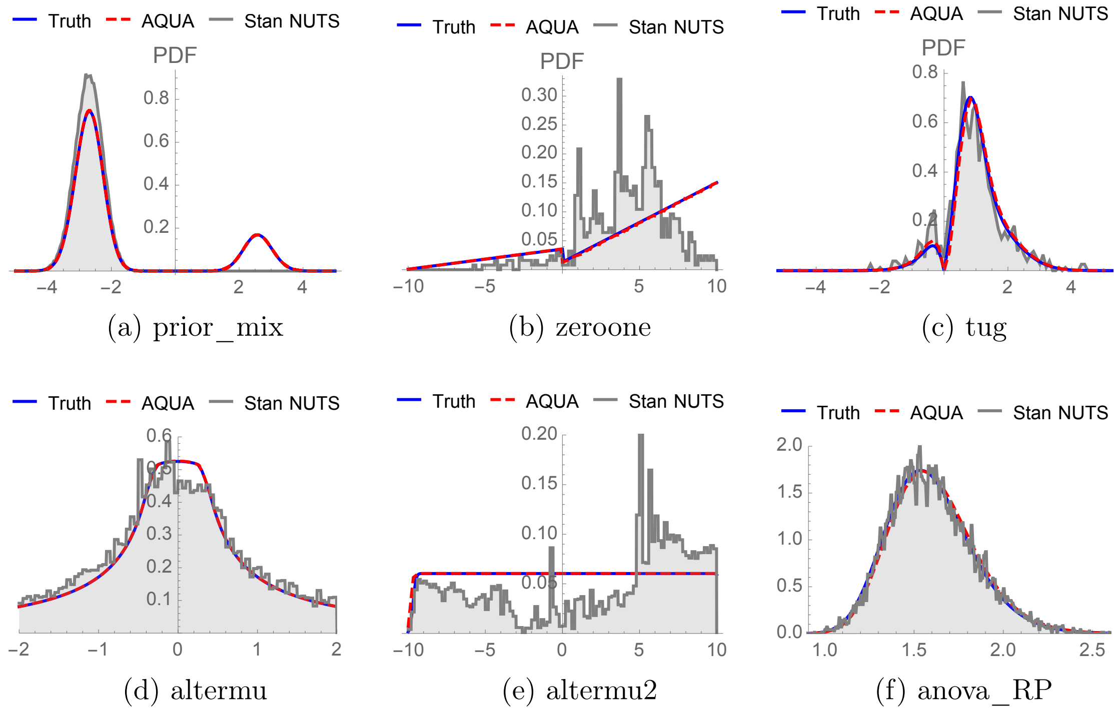
Overall, **AQUA** (Column 2-3) solves the probabilistic programs with average time 5.08s, median time 1.35s. For 20 out of 24 programs, it takes less than two seconds to compute the results. AQUA results in average error 0.01, median error 0.01, and maximum error 0.02. With our optimization on local variables (Section 4.4), we are able to handle the 7 robust programs which have 42-102 variables, which might timeout with a naive approach.

Stan VI (Column 4-5) finishes fast but results in significantly larger error than AQUA or Stan NUTS. The average error from VI is 0.15, minimum error is 0.03 and maximum error is 0.34. For all cases, VI cannot reach the same accuracy level as AQUA. While VI often fits the posterior means correctly but it is not able to capture the joint distribution shape especially when it is non-Gaussian (it is a well known characteristic of VI). **Stan NUTS** (Column 6-7) takes more time than AQUA to reach the same level of accuracy of AQUA, although in theory NUTS will converge to the true distribution with enough iterations. AQUA provides the similar (with difference < 0.01) or even better accuracy (with smaller KS statistic) in all cases for NUTS and NUTS fails to reach the same accuracy level by the maximum number of iterations in 12 cases.

PSI (Column 8) and **SPPL** (Column 9) are not able to give result in many cases. PSI does not finish running within 20 minutes in 11 cases, or evaluates to unsolved integrals in 11 cases, since the exact integration in posterior calculation is often intractable. SPPL does not allow transformed variables in `factor` statements, which is essential to specify the likelihood of the variables given observed data, and thus is inapplicable to most of the programs.

Figure 4.6 presents the posterior densities from six programs where Stan NUTS was not able to reach the same accuracy level of AQUA, within maximum iterations. X-axis shows the value of a variable in the program, Y-axis shows the posterior probability density of the variable. A solid blue line shows the ground truth, a dashed red line shows the density function computed from AQUA, the gray histogram shows the density estimated with samples from Stan NUTS after running for the same time as AQUA. For each program we present the posterior from one variable (the first one in alphabetical order); the posteriors from other variables show a similar pattern.

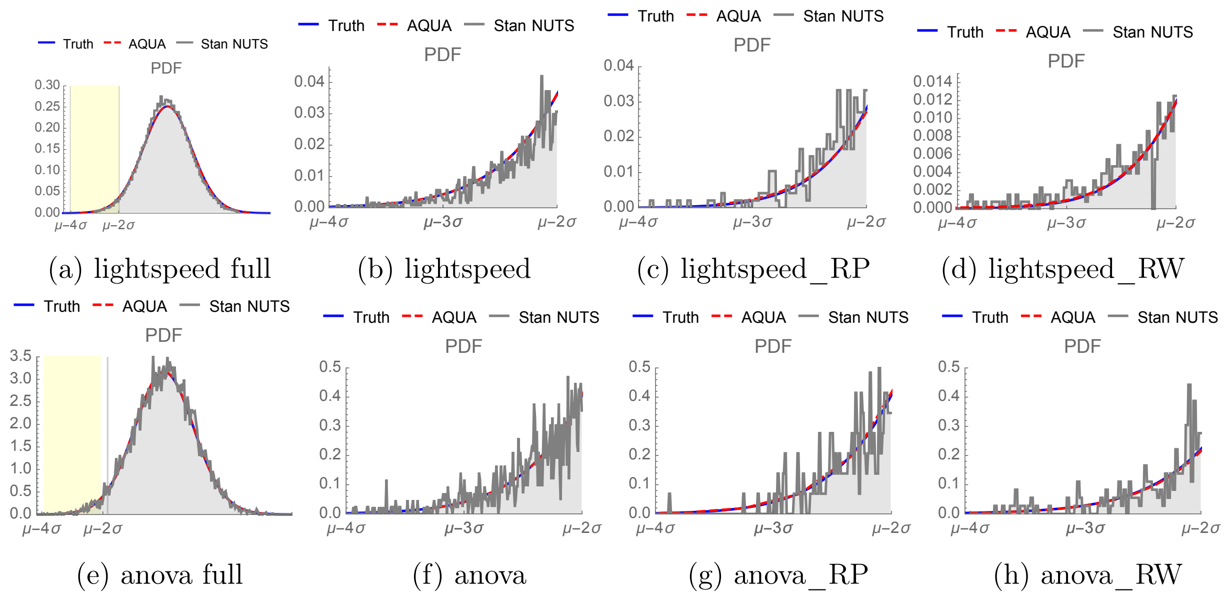
These examples show that AQUA is able to closely track the density of mixture models with large difference in densities (“prior_mix"), non-differentiable distributions (“zeroone" and “tug"), models with variable symmetries (in “altermu" and “altermu2" such symmetries can cause non-identifiability of variables from data), and some robust models with strong correlation between variables that can form complicated posterior geometries (“anova_RP").



(a) prior_mix (b) zeroone (c) tug

(d) altermu (e) altermu2 (f) anova_RP

Figure 4.6: Programs handled by AQUA for which Stan NUTS is imprecise.



(a) lightspeed full (b) lightspeed (c) lightspeed_RP (d) lightspeed_RW

(e) anova full (f) anova (g) anova_RP (h) anova_RW

Figure 4.7: Capturing tails by AQUA and Stan NUTS

4.6.2 Estimating the Posterior Tails

We illustrate AQUA's ability to capture tails on several robust models. The distribution for robust models are often more spread-out than the original model, as they are designed to capture outliers in the data. We consider two different robust models: (1) Reparameterized-Localization (RP) [44], which assumes that each data point is from its distribution with a local variance variable; (2) Reweighting (RW) [43], which down-weights potential outliers in the data. We show the results from AQUA and NUTS running for the same amount of time, together with the ground truth. We omit VI since its accuracy is significantly worse.

Figure 4.7 presents the comparison of AQUA and NUTS. Plots (a),(e) are the full posterior distributions of original distribution. We highlight the left tail $[\mu - 4\sigma, \mu - 2\sigma]$, where μ is the posterior mean of and σ its standard deviation. Plots (b),(f) show the magnified tails from original distribution, plots (c),(g) show the tails from the RP transformation, and (d),(h) show the tails from RW transformation. AQUA is able to capture the tails precisely for both original and robust models, while Stan NUTS is less precise on the robust models (e.g., its KS statistic is 0.05 compared to AQUA's 0.02).

4.7 RELATED WORK

Probabilistic Programming Languages. In recent times, Probabilistic programming languages have gained traction in both academic communities and industry. Most of these languages are thightly coupled with specific algorithms for approximate probabilistic inference. The majoristy of the languages are sampling-based [1, 7, 8, 97, 98, 99, 136, 137], and several recent languages support variational inference [8, 70, 100, 101]. These languages support a rich set of features including general loops and some have support for higher-order inference. These languages are Turing complete and some of them also support advanced features like higher order functions and composability with neural networks. More recently, languages like Edward and Pyro began combining Bayesian reasoning with deep learning [100, 101]. However, they are inherently approximate: sampling-based approaches can reach accurate solution only in the limit, while the variational inference-based may not have theoretical guarantees (except for specific distributions). Recently, there has been interest in expanding the reach of exact or near-exact inference methods. Although these methods are computationally intractable in general, they can solve many practical problems. We next discuss these techniques falling in the domain of symbolic inference and volume computation.

Symbolic Inference. Researchers have proposed several symbolic inference techniques in recent years [19, 20, 21, 22]. Each of these techniques have limitations which AQUA improves

upon. DICE [21] performs fast and exact inference by reducing discrete probabilistic programs to weighted model counting. It supports only programs with discrete distributions.

Hakaru [19] and PSI [20] languages have constructs for both discrete and continuous distributions. They perform exact inference using computer algebra (e.g., Hakaru uses Maple and PSI uses Mathematica). However, they often cannot solve integrals for even modestly complicated probabilistic programs with continuous distributions (as our evaluation has also demonstrated for PSI).

SPPL [22] performs exact inference and supports programs with discrete and continuous distributions. Its inference is enabled by translating programs into sum-product expressions. Nevertheless, it does not allow users to specify the likelihood on transformed variables with continuous distributions, which is a necessity for many real-world models like hierarchical regression or time-series models (such as those used in our evaluation).

QCoral [138] and SYMPAIS [139] combine symbolic execution with sampling to solve the satisfaction probability of constraints. They aim to quantify the probability of a target event by representing the path conditions of the event with symbolic expressions. They apply constraint solver and sampling to compute the probability under the symbolic conditions. However, they only compute the probability of a given event and do not output the whole posterior.

Earlier works [140, 141, 142, 143, 144] perform symbolic inference on graphical models. The distributions they support in the graphical models are limited and are not able to represent many models allowed in probabilistic programs. Shachter et al. [140] works on discrete distributions. Chang et al. [141] allows both discrete and continuous distributions, but restricts continuous distributions to be linear-Gaussian related. Other works [142, 143, 144] use easy-to-integrate approximations to replace the continuous distributions, which include gaussian mixtures, truncated exponentials, and polynomials. These approximations may introduce inevitable error, and these works do not provide any formal guarantee of the error bound. On the contrary, AQUA avoids distorting the distribution by using quantizations of the original distribution. AQUA's quantizations have the guarantee to converge to the true distributions when using sufficiently many splits (Theorem 4.1).

In contrast to these existing approaches, AQUA supports a wide range of probabilistic models with continuous distribution, involving transformed or correlated random variables, and provides scalable, exact (or approximately exact), and interpretable solutions.

Volume Computation. Several works use volume computation methods to make a precise approximation of probabilistic inference [4, 13, 25]. These approaches have constraints on the form of programs they support, regarding conditioning and continuous distributions. None of these systems can support conditioning on continuous variables, and thus we have

not used them in our evaluation.

Sankaranarayanan et al. [25] estimates the probability upper and lower bounds for properties of probabilistic programs. It applies symbolic execution, bounds the path probabilities with hypercubes and does volume bound computation to estimate the probability of the given property. FairSquare [13] verifies the fairness property of probabilistic programs. It generates probabilistic verification conditions and computes the weighted volume described by the conditions. In the volume computation, it presents an approach to discretize some continuous distributions, but it is inflexible. For instance, it approximates Gaussians with only five intervals. These techniques compute only the probability of an event, not the entire posterior. Sweet et al. [4] estimates the probability of information leakage of a query. It ensures a sound over-estimation combining sampling with concolic execution. They support only discrete models.

4.8 CONCLUSION

AQUA is a new inference algorithm which works on general, real-world probabilistic programs with continuous distributions. By using quantization with symbolic inference, AQUA solved all benchmarks in less than 43s (median 1.35s). Our evaluation shows that AQUA is more accurate than approximate algorithms and supports programs that are out of reach of state-of-the-art exact inference tools.

Supplementary information. AQUA is avaliable at `https://github.com/uiuc-arc/AQUA`.

CHAPTER 5: PRECISE ABSTRACT INTERPRETATION OF PROBABILISTIC PROGRAMS WITH INTERVAL DATA UNCERTAINTY

5.1 INTRODUCTION

Probabilistic programs (PP) express complex statistical models as simple programs and automate inference of a posterior distribution using one of many inference algorithms – either approximate [1, 6, 9, 12, 35, 70, 145, 146] or exact [19, 20, 21, 22, 102, 147]. Increasingly, probabilistic programs have been used in applications that make critical decisions, spanning algorithmic fairness [13, 14], computer networks [15, 16, 148], pandemic modeling [3], and security/privacy [4, 5, 17, 18]. In many of these applications, one often requires formal guarantees on the probability distribution, such as whether it satisfies strict assertions on the probabilities of different events [25, 149]. There has been a significant recent progress in verification of probabilistic programs with discrete distributions [15, 21, 148, 150, 151]. However, such approaches typically rely on enumerating and/or abstracting over discrete states.

Obtaining formal guarantees for *continuous* posterior distributions is more challenging than for discrete distributions. This challenge stems from the need to symbolically evaluate both integrals which can be intractable and probability density functions which can be highly nonlinear. While exact inference methods [19, 22, 152] have tried to address this challenge with improved symbolic reasoning, such methods experience difficulties with scaling beyond small probabilistic programs.

As a tractable alternative, Abstract Interpretation [153] is a general approach for soundly overapproximating the semantics of programs, and has also been defined for probabilistic programs, e.g., [23, 24]. Abstract interpretation defines *an abstract domain*, a class of mathematical objects – e.g., intervals (boxes) or polyhedra – that soundly over-approximate the underlying program states (in this case the probability distribution), and *abstract transformers*, functions that define the interpretation of program statements in that abstract domain. While abstract interpretation offers better tractability for reasoning about probabilistic programs, many existing works tailored their abstractions to probabilistic programs *without* Bayesian inference [4, 13, 25]. However, one recent work, GuBPI [26] does support abstract interpretation for Bayesian inference by over-approximating the range of the posterior with *interval bounds*. Nevertheless, one must ensure that any posterior bounds are precise enough for verification tasks, and as we will show in Section 5.10, GuBPI [26] suffers from large imprecision even for small PPs due to the naïve application of interval-based abstraction.

However, improving the precision of abstract interpretation of probabilistic programs poses challenges due to the large number of nonlinear distributions a PP may encode. Hence manually designing precise abstract transformers for each primitive distribution and composite expressions with multiple such distributions, would be exceedingly burdensome on the verification engineer.

In addition to the precision of a probabilistic program analysis, scalability to large datasets is a primary concern for Bayesian inference, since one can have 1000s of data observations, as we will show in our experiments. However existing formal reasoning techniques like GuBPI [26] or PSI [20] suffer limitations that prevent scaling to large datasizes (under 50 when evaluating on the benchmarks in our paper).

Lastly, beyond scalably computing sound and precise bounds for a *single* posterior distribution, it is often advantageous to soundly enclose a *set* of possible posteriors. An important challenge in Bayesian inference is the study of adversarial robustness of Bayesian models to small perturbations in the observed data, as these perturbations can drastically change the inferred posteriors [154, 155]. Precise lower and upper bounds could then be used to formally reason about the set of all possible posteriors that could result from data perturbations. Indeed, in *robust* Bayesian statistics [156], practitioners have developed techniques to obtain interval bounds on a set of possible posterior distributions, which allows one to over-approximate the results of Bayesian inference when the data or parameters have some non-probabilistic (interval-based) perturbation. While adversarial dataset attacks have been studied for machine learning models, e.g., [157], to the best of our knowledge, verifying robustness to this threat model has been significantly less explored in probabilistic programming literature.

Hence to date, there exists no automated program analysis for verifying properties of probabilistic programs that is general enough to support Bayesian inference with continuous distributions *and* data perturbations while simultaneously maintaining high precision and scalability.

Our Work. AURA is a novel abstract interpretation of probabilistic programs that produces sound and precise bounds on the inferred posterior distributions. By evaluating a probabilistic program abstractly, the bounds AURA obtains can be used to verify probabilistic assertions over a program's posterior. Additionally, AURA is the first probabilistic programming system that can efficiently compute precise and sound bounds on an *infinite set of possible posterior distributions* when the observed data is specified as bounds by the user. Lastly, AURA is able to scale to a data size that is an order of magnitude bigger than the size handled in prior work [20, 26].

AURA's key technical contribution comes from reducing abstract interpretation to tractable gradient-based optimization by leveraging a distribution-shape pattern in continuous PPs. Thus AURA constructs sound and optimally precise abstract transformers over the interval abstract domain. Specifically, AURA's transformers are applicable to complex subprograms or even the entire program when their distributions are *pseudoconcave*. Pseudoconcavity is a relaxation of the familiar notion of concavity, and many commonly used continuous distributions (e.g., Gaussian, Uniform, Exponential) and expressions over them satisfy this property. This insight helps our abstract transformers achieve significantly more precision than composing standard interval arithmetic operations for simple subexpressions. Moreover, we show how our abstraction can also be combined and composed with standard interval transformers (Section 5.6.4) to maintain the generality needed to analyze more complicated programs that may not be end-to-end pseudoconcave.

Our abstract transformers first select a partition of the interval of the input parameters. For each partition, they compute the lower and upper bounds on the program's distribution by solving a gradient-based optimization problem. Because of the pseudoconcavity of the distribution we are bounding, the gradient-based search can provably and efficiently find the true optima on each partition, and thus obtain the provably optimal lower and upper bounds for that partition. Finally, the transformer conjoins the bounds for the partitions to produce sound posterior bounds.

In addition to these precise abstract transformers, AURA allows programmers to specify interval bounds on the observed data which AURA will then propagate through the program. These results can then be used to certify bounds on probabilistic queries in order to bound the probability of an event. These bounds hold for *all* possible posterior distribution that could result from data perturbations. Thus AURA uses abstract interpretation to verify properties for infinitely many probabilistic programs simultaneously. AURA also makes integration of the over- and under-approximated densities tractable and efficient during marginalization, distribution normalization, or expectation calculation. Finally, our algorithm can naturally decompose into independent sub-problems that lend themselves to efficient parallel execution on both CPU and GPU.

We evaluated AURA across 19 programs under two scenarios: inference without and with data perturbation. In the first case, AURA outperformed GuBPI [26], a state-of-the-art inference technique with interval-based distribution approximation. On a single-core CPU, AURA solves the problems in under a second, with minimal loss of precision. GuBPI could solve only 9 programs taking 77x more time, and over 2000x less precisely. AURA's GPU version further improves the execution time by over 2x on average over the CPU version (but significantly higher for bigger programs with more data observations). In the scenario with

data perturbation, AURA outperformed interval-based analysis with 12.9x better precision, in 3.1s (geomean). We also demonstrate that AURA's effectiveness in certifying precise bounds extends to analyzing probabilistic predicates, and that AURA successfully scales to thousands of observed datapoints. AURA's scalability results from leveraging parallelism across both the number of partitions (algorithmic-level parallelism) and CPU and GPU cores (implementation-level parallelism).

Contributions

The paper presents the following core technical contributions:

- **Problem Formulation**. We introduce a novel formulation of the problem of certifying bounds on a set of posterior distributions resulting from (bounded) data perturbations. Our formalization is built upon abstract interpretation, which allows us to combine our precise abstract transformers with standard ones whenever the concavity properties hold for only part of the program.

- **Gradient-based Optimization**. We design a novel algorithm for obtaining precise abstract transformers for probabilistic programs that uses gradient-based optimization for optimally solving for precise bounds on the posterior distribution.

- **Soundness.** We show that our abstract interpretation can guarantee soundness for a broad class of probabilistic expressions and programs whose posteriors satisfy concavity or pseudoconcavity at each interval. Additionally, our interval bounds are provably optimal, thus yielding the most precise (interval domain) abstract transformer for a pseudoconcave expression.

- **Implementation.** We present a PyTorch-based implementation of AURA that can be efficiently parallelized on CPU and GPU platforms.

- **Experimental Results**. Our evaluation of AURA across 19 programs under two scenarios – with and without data perturbation – shows that AURA can improve precision an order of magnitude over the state-of-the-art baselines. Additionally, AURA easily scales to larger datasets, producing precise results for benchmarks with an order of magnitude more observations than any existing formal analysis.

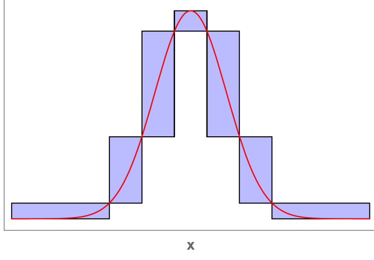


Figure 5.1: Bounding Single Posterior

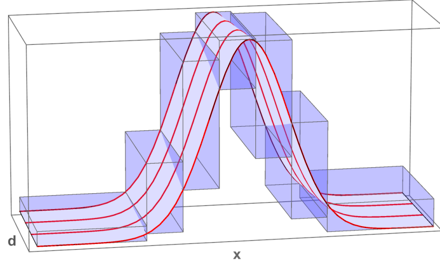


Figure 5.2: Data Perturbation Analysis

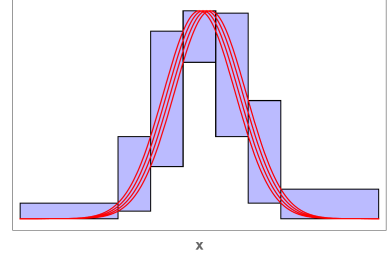


Figure 5.3: Bounding Data Perturb.

5.2 EXAMPLE

Figure 5.4 presents a probabilistic program, P, for estimating the ground braking force. The latent parameter x represents the ground braking force exerted on the vehicle (unit: hundred Newtons), while b represents the observed deceleration (unit: m/s^2) under specific environmental conditions. Although b, the deceleration, can be measured on the wheels, x, the ground braking force, is not directly measurable. However, they are closely related, and often proportional.

In this model, the engineers use the b observation to infer the value of x. They assume that x follows a uniform(0,100) distribution as their prior assumption, meaning that x can take any value between 0 and 100 equally likely. Here the variable b that is assumed to follow a normal distribution is observed to be 6.29 (m/s^2).

```
1 x ~ uniform(0, 100)
2 b ~ normal(0.1*x+1, 1)
3 observe(b, 6.29)
```

Figure 5.4: Example Program

Concrete Semantics. The unnormalized concrete semantics for the Fig. 5.4 program are denoted as $[\![P]\!](x, d)$ and define the unnormalized posterior:

$$[\![P]\!](x, d) = \frac{1}{100} \cdot \frac{1}{\sqrt{2\pi}} e^{-\frac{1}{2}(d-(0.1\times x+1))^2}, \quad x \in [0, 100], \; d = 6.29 \tag{5.1}$$

In Bayesian inference, the unnormalized posterior, as represented by $[\![P]\!](x, d)$, is a product of the likelihood of the observed data and the prior belief about the distribution of the parameters. Unlike traditional likelihoods, $[\![P]\!](x, d)$ is a function of both latent parameter x and data observation d. For a fixed d, Figure 5.1 illustrates this function with a red curve, where the x-axis represents the values of x and the y-axis represents the value of the unnormalized posterior. The unnormalized posterior, however, does not provide direct probability since its integral over all possible parameter values may not equal one. We

thus need to compute a *normalized* posterior, by calculating a bound on the normalizing constant, which ensures the unnormalized posterior is properly scaled.

AURA Abstract Semantics for Fixed Data. To compute a certified bound on the normalizing constant, we first abstractly interpret the range of the unnormalized posterior. Intuitively, we will use interval bounds to obtain lower and upper Riemann sums to bound the integration, hence we use the interval domain. For latent variable x in this program, which takes values between 0 and 100, AURA partitions the range [0,100], denoted as $x^\sharp$, into consecutive sub-intervals $x_i^\sharp$, such that $x^\sharp = \bigcup_i x_i^\sharp$. Examples of $x_i^\sharp$ intervals splits could be $x_1^\sharp = [0, 20], ..., x_n^\sharp = [80, 100]$. In Figure 5.1, the width of the blue boxes corresponds to each $x_i^\sharp$, and the height denotes the interval range of the unnormalized probability density score for that split. To compute the interval over-approximation of the unnormalized probability density score in Eq. 5.1, AURA applies to each interval partition $x_i^\sharp$, an abstract transformer defined for the *entire* program's expression $[\![P]\!]^\sharp(x_i^\sharp, d)$. This approach allows AURA to find the tightest interval bound through optimization. As illustrated in Figure 5.1, the heights of the blue boxes tightly enclose the red curve.

AURA Abstract Semantics for Data Perturbations. In the previous setting d was a fixed constant, however in a practical scenario, an engineer might notice that sensor defects occur during the use of such a braking system, leading to potential inaccuracies in the collected deceleration data. These inaccuracies could result in changes of up to $\varepsilon = 0.05$ in the data. With this consideration, the model now has an additional dimension of uncertainty. The unnormalized posterior $[\![P]\!](x, d)$, as a function of both x and d (which is no longer constant), is depicted in Figure 5.2 as a white surface.

A core contribution of AURA is that we can ultimately obtain a sound enclosure of the normalized posterior that also accounts for all potential sensor inaccuracies. To do so, AURA must also abstract the range of possible data observations to $d^\sharp = [d, d+\varepsilon]$, which we call *the data perturbation interval*. Hence AURA can then abstractly interpret $[\![P]\!]^\sharp(x^\sharp, d^\sharp)$. Although this example illustrates ε being added exclusively to the right of d, AURA is capable of supporting intervals that extend in both directions from any given concrete value. To find the lower and upper bounds, AURA must solve a 2D optimization problem where each 2D region is the Cartesian product of of $x_i^\sharp$ and $d^\sharp$. The resulting lower and upper bounds are depicted by the height of the blue cuboids in the figure. Hence Figure 5.2 illustrates AURA's interval over-approximation for data perturbations as 3D blue cuboids, which surrounds the actual white surface. For illustration, on the same plot, we have also shown four sampled density functions represented as four red curves, each resulting from evaluating $[\![P]\!](x, d)$ with a different d value. Hence, by abstracting d as an interval, AURA is able to determine bounds for an infinitely large number of such red lines that exist along

the white surface. AURA computes these bounds by aggregating the results from each 2D optimization subproblem. Thus AURA obtains the bounds on $[\![P]\!](x, d)$ for all possible d values within $d^\sharp$.

For visual intuition, we have projected Figure 5.2 onto 2D which is shown in Figure 5.3 where the various red lines represent posteriors obtainable from different concrete d values. One can notice that the heights of these boxes differ from those in Figure 5.1, reflecting the impact of data perturbation on the density score.

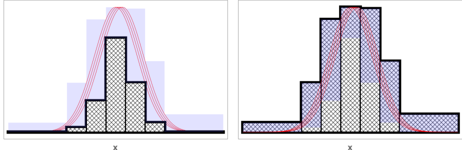


Figure 5.5: Lower and Upper Riemann Sums to bound the Normalizing Constant Integral

Abstract Integration. In Bayesian inference, the normalized posterior distribution can be deterministically derived from the unnormalized posterior by integrating the unnormalized posterior and then dividing the unnormalized posterior by this normalizing constant. Hence the normalized concrete semantics can be defined as $[\![P]\!]_n(x, d) = \frac{[\![P]\!](x,d)}{\int_x [\![P]\!](x,d)dx}$. In the general case, these integrals are not computable, hence we settle for an interval enclosure around the normalizing constant.

To obtain upper and lower bounds on the normalizing constant, AURA bounds the corresponding integral. The integral bounds can be computed using the previous interval bounds on the unnormalized posterior to give lower and upper Riemann sums. Figure 5.5 uses shaded areas to depict the lower and upper Riemann sums based on AURA's bounding boxes. Intuitively, computing the volumes (or Lebesgue measure in the multidimensional case) of the shaded areas under the respective lower and upper bounds gives the lower and upper bounds on the integral. AURA can then perform interval division to get sound intervals for the *normalized* posterior.

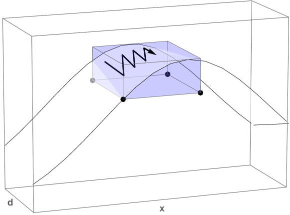


Figure 5.6: Gradient Optimization and Corner Checking on a Partition

Optimization for Tight Posterior Bounds. While we have defined AURA's abstraction and the normalization constant bound in terms of precise interval bounds for each $\mathrm{x}_i^\sharp$, up to this point we have not shown how AURA computes these bounds. In reality, AURA employs gradient optimization techniques to determine the upper bounds and examines the corners of each partition to find the lower bounds. This procedure helps AURA obtain far better precision than naive interval arithmetic. Figure 5.6 illustrates this process on an example interval partition. The wiggling black arrow in the figure demonstrates how AURA uses gradient ascent to locate the maximum of the white surface restricted to this partition. Given that the function $[\![P]\!](x, d)$, represented by the white surface, is concave (or more generally pseudo-concave), gradient ascent is *guaranteed* to find

the maximum, which is then taken as the upper bound for this interval partition. Further leveraging the concave or pseudo-concave property, the minimum value must occur at one of the corners of the partition, depicted as black dots in the figure. This approach guarantees that AURA finds the tightest sound bounds for an interval partition.

Example Results. We now extend the example to use 100 observed data points, as shown in Fig. 5.7. AURA's formalism allows perturbations of multiple data points in parallel. We will assume that multiple points can be subject to sensor defects, potentially increasing its observed value by up to 0.05. AURA can automatically identify the most sensitive points, based on the maximum gradient of the posterior distribution. In this example, we analyze five points that are simultaneously perturbed. AURA can prove sound posterior bounds within just 0.115 seconds when using 200 partitions.

AURA's bounds are presented as blue boxes in Figure 5.8. The red dots in this figure represent a sampled histogram of x, based on different concrete data points d within the specified interval $d^\sharp$, showing that AURA's bounds are precise in this case. Additionally, the orange boxes illustrate posterior bounds obtained by a naive interval analysis and are much less precise than AURA at enclosing the sampled histograms. This trend is because interval analysis applies over-approximate abstract transformers compositionally, which accumulates imprecision. In contrast, AURA's abstraction is able to improve precision by abstracting the entire unnormalized posterior at once.

```
1 x ~ uniform(0, 100)
2 for i in 1..100:
3   observe(
4         normal(0.1*x+1, 1),
5         data[i] )
```

Figure 5.7: Example with Loop

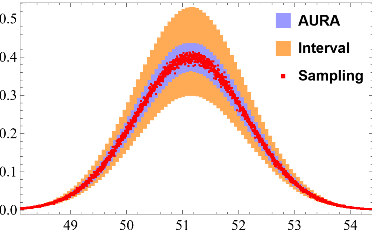


Figure 5.8: Example Result

Bounds on Probabilities of Queries. The certified bounds on the normalized posterior can then be used for various queries. For instance, engineers may need to guarantee that the braking force x (in units of hundred Newtons) under the experimental environment should not fall below a certain safety threshold, say 50 hundred Newtons. Hence AURA's posterior bounds can be used to bound the posterior's probability of the query $Q \triangleq x < 50$. Similar to the computation of the normalizing constant, AURA finds sound bounds on this probability by performing abstract integration with Riemann sums, over the area (or Lebesgue measure for high-dimensional case) where Q is satisfied.

In this example, AURA provides a probability range of [0.11, 0.14] for the braking force falling below a critical threshold. This range is twice as precise as the [0.09, 0.17] obtained from interval analysis. The more precise the probability bounds, the lower the uncertainty around the system's ability to maintain braking force above the safety margin, resulting

in a more predictable braking system. Hence, AURA can prove precise bounds on the posterior probability of queries that hold for an infinite set of possible posterior distributions. Generally, the narrower bounds indicate a smaller effect from data perturbation, namely a better *robustness*. In this work, we define *robustness* as the model's ability to deliver reliable results regardless of data perturbations.

Additionally, calculating this precise probability range with AURA does not add a significant amount of time to the computation of posterior bounds. For simplicity, this example is presented with a single parameter and 100 data points to illustrate AURA's internal steps. In Section 5.10, we evaluate AURA on benchmarks that scale to more parameters and up to 5000 data points. For this example, while PSI can process results but requires more than 6 minutes and manual selection of data perturbation, AURA excels by computing results in just 0.115 seconds and using gradient information for automatic data point selection.

5.3 PRELIMINARIES

5.3.1 Language

We present our probabilistic programming language in Fig. 5.9. Our syntax is conceptually similar to the syntax of Stan [91] and Stan-like languages [158, 159], because we specify the prior distributions over the latent variables in the model M and then we specify the observations used for inference in the data block D. The programmer can also specify models where the parameters of one distribution are an arithmetic expression E of other latent parameters e.g., in linear regressions. The priors in M are required to be continuous with compact support. The observed distributions in D can be either continuous or discrete. If the observed distribution is discrete (e.g., Beta-Bernoulli models), we will not consider data perturbations, thus in those cases, AURA will only compute bounds on a single posterior. If the observed distribution is continuous, AURA supports computing bounds over an infinite set of possible posteriors since the observed value d_i can be specified as an interval.

$$
\begin{aligned}
P &::= M \mid M; D \\
M &::= \mathtt{x}_i \sim \mathit{Dist} \mid M; M \mid \mathtt{if}\ \mathtt{flip}(p)\ M_1\ \mathtt{else}\ M_2 \\
&\mid\ \mathtt{for}\ \mathtt{j=}n_1\ \mathtt{to}\ n_2\ \mathtt{do}\ M \mid \mathtt{let}\ \mathtt{x}_i = E\ \mathtt{in}\ M \\
D &::= \mathit{observe}(\mathit{Dist}, d_i) \mid D; D \mid \mathtt{for}\ \mathtt{j=}n_1\ \mathtt{to}\ n_2\ \mathtt{do}\ D \\
E &::= \mathtt{x}_j \mid E + E \mid E - E \mid E * E \mid E/E \mid c \in \mathbb{R} \\
\mathit{Dist} &::= \mathit{dist}(E_1, \ldots, E_N),\ \mathit{dist} \in \{\mathit{uniform}, \mathit{bernoulli}, ...\}
\end{aligned}
$$

Figure 5.9: AURA Language

In the $\mathtt{flip}(p)$ primitive, p is a fixed constant between 0 and 1. The language is first-order, however one can still encode a broad class of popular probabilistic models such as Linear

and Logistic Regressions, Time-Series Models, various Hierarchical Bayesian models, as well as others.

5.3.2 Concavity and Convexity

A core component of AURA's abstraction will rely upon concavity and concavity-like properties of posterior distributions, hence we now define the necessary mathematical preliminaries.

Definition 5.1. Convex Set. A set $\mathcal{X} \subseteq \mathbb{R}^d$ is convex if for all $\alpha \in [0,1]$ and any $x_1, x_2 \in \mathcal{X}$, then $\alpha x_1 + (1-\alpha)x_2 \in \mathcal{X}$

Definition 5.2. Concavity. A function f is concave over some convex domain $\mathcal{X}$ if for any $\alpha \in [0,1]$ and any $x_1, x_2 \in \mathcal{X}$ the following holds:

$$f(\alpha x_1 + (1-\alpha)x_2) \geq \alpha f(x_1) + (1-\alpha)f(x_2) \tag{5.2}$$

The **Concave** functions are closed under summation, which will be important for AURA's analysis. However many of the probability distributions AURA analyzes are *not* concave. We will see that AURA's analysis still supports weaker notions of concavity, which we describe next.

Definition 5.3. Quasiconcavity. A function f is quasiconcave over some convex domain $\mathcal{X}$ if for any $\alpha \in [0,1]$ and any $x_1, x_2 \in \mathcal{X}$ the following holds:

$$f(\alpha x_1 + (1-\alpha)x_2) \geq \min(f(x_1), f(x_2)) \tag{5.3}$$

Definition 5.4. Log-Concavity. A function f is logarithmically-concave over some convex domain $\mathcal{X}$ if for any $\alpha \in [0,1]$ and any $x_1, x_2 \in \mathcal{X}$ the following holds:

$$f(\alpha x_1 + (1-\alpha)x_2) \geq f(x_1)^{\alpha} \cdot f(x_2)^{1-\alpha} \tag{5.4}$$

For a strictly positive function f (e.g., a probability density), the following implication holds:

$$\textbf{Log-Concave } f \implies \textbf{Concave } \log(f) \tag{5.5}$$

Many named probability densities (Appendix B.1; Table B.1) are **Log-concave** functions and the **Log-concave** functions are closed under multiplication. Additionally, Quasiconcavity is useful for defining Pseudo-concavity.

Definition 5.5. Pseudoconcavity. A function $f(x)$ is pseudoconcave if and only if $f(x)$ is quasiconcave and for any x^*, $\nabla f(x^*) = 0 \implies x^* = \arg\max f(x)$.

Pseudoconcavity is similar to **Quasiconcavity**, however any stationary point of a pseudoconcave function is *necessarily* an optimum value. Furthermore, any Quasiconcave function whose gradient is never zero is automatically Pseudoconcave. Figure 5.10 shows example functions. One may notice the flat plateau region of the quasiconcave function consists of stationary points which are *not* globally optimal. Hence we will later see in Section 5.6 why pseudoconcavity instead of quasiconcavity is essential for ensuring a gradient ascent procedure does not become stuck in local extrema.

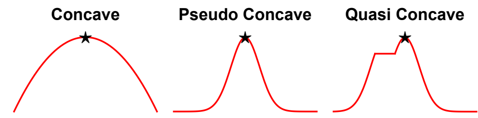


Figure 5.10: Illustration of Different Notions of Concavity

5.3.3 Abstract Interpretation

In abstract interpretation [153], one first defines a concrete semantics of programs and then a corresponding abstract semantics to soundly over-approximate those concrete semantics.

Interval Domain

AURA's analysis will leverage the interval domain. In the interval domain each variable is abstracted by an interval $[a, b] \in \mathbb{IR}$ where a is the lower bound and b is the upper bound and $a \leq b$. In our setting an interval can contain $\pm\infty$ as a lower or upper bound. The set of all m-dimensional intervals will be denoted as $\mathbb{IR}^m$, and a given element of this set will be denoted as $x^\sharp \in \mathbb{IR}^m$ where $x^\sharp[i] = [a_i, b_i]$. The concretization $\gamma : \mathbb{IR}^m \to \mathcal{P}(\mathbb{R}^m)$ of a multidimensional interval is just the set of all points contained in that interval hence:

$$\gamma(x^\sharp) = \{(x_1, ..., x_m) : \forall i \in \{1, ..., m\}, a_i \leq x_i \leq b_i \text{ where } [a_i, b_i] = x^\sharp[i]\} \tag{5.6}$$

Abstract Transformers

While the abstract domain (in our case the Interval domain) describes how sets of program variables are represented, it does not tell us *how to compute* the representation. The computation of an abstract element comes from the abstract transformers. For the interval abstract domain, the most basic abstract transformers are the standard interval arithmetic operations

$$
\begin{aligned}
&[\![P]\!](x,d) = [\![M;D]\!](x,d) && [\![M;D]\!](x,d) = [\![M]\!](x)\cdot[\![D]\!](x,d)\\
&[\![\texttt{if flip}(p)\ P_1\ \texttt{else}\ P_2]\!](x,d) = && [\![M;M]\!](x) = [\![M]\!](x)\cdot[\![M]\!](x)\\
&\qquad\qquad p[\![P_1]\!](x,d) + (1-p)[\![P_2]\!](x,d) && [\![D;D]\!](x,d) = [\![D]\!](x,d)\cdot[\![D]\!](x,d)\\
&[\![observe(Dist,d_i)]\!](x,d) = [\![Dist]\!](x,d)\circ d[i] && [\![\mathtt{x}_i \sim Dist]\!](x,d) = [\![Dist]\!](x,d)\circ x[i]\\
&[\![E+E]\!](x,d) = [\![E]\!](x,d) + [\![E]\!](x,d) && [\![E-E]\!](x,d) = [\![E]\!](x,d) - [\![E]\!](x,d)\\
&[\![E*E]\!](x,d) = [\![E]\!](x,d) * [\![E]\!](x,d) && [\![E/E]\!](x,d) = [\![E]\!](x,d)/[\![E]\!](x,d)\\
&[\![\mathtt{x}_j]\!](x,d) = x[j] && [\![cE]\!](x,d) = c\cdot[\![E]\!](x,d)\\
&[\![dist(E_1,\ldots,E_N)]\!](x,d) = p_{dist}\,(u;[\![E_1]\!](x,d),\ldots,[\![E_N]\!](x,d))\,,\ dist\in\{uniform,\ bernoulli,\ \ldots\}
\end{aligned}
$$

Figure 5.11: Key Rules of Unnormalized Concrete Semantics.

(we denote them as $+^\sharp$ and $\cdot^\sharp$), which can easily be composed [160], however we will later see how AURA is able to obtain precise abstract transformers by solving optimization problems.

5.4 CONCRETE SEMANTICS

We now describe and rigorously formalize the concrete semantics of probabilistic programs. Intuitively, the semantic meaning of a probabilistic program is the normalized posterior distribution, which corresponds to the unnormalized likelihood defined by the statements in the probabilistic program divided by a normalizing constant. We will later see that by fittingly setting up our concrete semantics, the abstract semantics can likewise be easily formalized.

Preliminary Transformation. To simplify the formalism description, we do three source-to-source transformations. The first is moving conditionals upward hence the production for P becomes $P ::= M \mid M;D \mid \texttt{if flip}(p)\ P_1\ \texttt{else}\ P_2$. To move conditionals upward, this transformation includes the code before and after the conditional into the branches. The second transformation is unrolling the `for` loops. Hence the productions for M becomes $M ::= \mathtt{x}_i \sim Dist \mid M;M$ and for D becomes $D ::= observe(Dist,d_i) \mid D;D$ since loops can be unrolled to sequencing: $M;M$ and $D;D$. The third transformation is to replace all occurrences of (fresh) variables introduced by the `let` bindings with their original expressions (using capture-avoiding substitution), so that all the expressions and subexpressions will only contain variables corresponding to sampled distributions, $\mathtt{x}_k$, as the input x only includes those variables corresponding to distributions.

Unnormalized Concrete Semantics. We first formalize the concrete semantics of PPs in terms of their *unnormalized* likelihood. Because likelihoods are just density functions which map observations to scores, we formalize this mapping using a score-based functional semantics.

In the score-based semantics, the interpretation $[\![\cdot]\!]$ of a probabilistic program P is a func-

tion that scores the likelihood of a given trace $x \in \mathbb{R}^m$ (where m is the number of latent parameters) and given data observations $d \in \mathbb{R}^n$. Hence $[\![\cdot]\!]$ is a function of both x and d. Specifically the semantic signature for interpreting a program P is $[\![P]\!] : \mathbb{R}^m \times \mathbb{R}^n \to \mathbb{R}$, and $[\![P]\!](x, d) : \mathbb{R}$. The full semantics are given in Fig. 5.11. For the distribution rule $[\![dist(E_1, \ldots, E_N)]\!](x, d)$, we take the probability density function (PDF) of the particular distribution, denoted as p_{dist}. Intuitively, each statement multiplies the total (unnormalized) probability score by the corresponding probability density function of either the latent parameter or the observed data sample. Hence why in the sequencing rules for models $[\![M; M]\!]$ and observations $[\![D; D]\!]$, we multiply the respective densities of each. Thus $[\![P]\!](x, d)$ computes an (unnormalized) product of density functions when the parameters are given by x and the data observations are given by d. For branches, we take the linear combination of the density functions.

Unnormalized Log-Likelihood Semantics

For reasons of numerical stability it is often helpful to work with the logarithm of the likelihood instead of the original likelihood itself. Thus having defined the Unnormalized Concrete Semantics (which encode a likelihood), we can now define the *log-likelihood* semantics by simply taking a logarithm. In particular:

$$[\![P]\!]_{\log}(x, d) = \log([\![P]\!](x, d)) \tag{5.7}$$

Similarly, we can easily convert back to the original semantics: $[\![P]\!](x, d) = \exp([\![P]\!]_{\log}(x, d))$.

Normalized Concrete Semantics

We can now formalize the notion of a *normalized* probabilistic program. The normalized Concrete semantics are denoted by $[\![\cdot]\!]_n$ and are given by:

$$[\![P]\!]_n(x, d) = [\![P]\!](x, d) \, / \int_x [\![P]\!](x, d) dx, \tag{5.8}$$

where $\int_x [\![P]\!](x, d) dx$ is the normalizing constant. In the general case, the normalizing constant is not tractable to compute, hence the normalized concrete semantics are in general not tractable. However our goal will be to simply over-approximate these (intractable) concrete semantics with AURA's tractable abstract semantics.

5.5 ABSTRACT SEMANTICS FOR DATA PERTURBATION

We now define an abstract semantics for over-approximating entire sets of posterior distributions. Intuitively, the set of posteriors we abstractly interpret are the set of *all* posteriors obtainable when the data d could be perturbed by an adversary. However if one wishes to only bound a *single* posterior, they can still use our semantics, the data interval will just be a degenerate interval with identical lower and upper bounds. A key benefit of abstract interpretation is that it provides a general framework to tackle these problems. We will later see in Section 5.6, how this generality also allows us to compose our precise abstract transformers with standard interval abstract transformers, even when the whole posterior does not satisfy the pseudoconcavity condition.

5.5.1 Unnormalized Abstract Semantics for Data Perturbation

In the data perturbation setting, the observed dataset will be given by some interval, $d^\sharp \in \mathbb{IR}^n$. Thus for a probabilistic program P we have the following signature $[\![P]\!]^\sharp(x^\sharp, d^\sharp) : \mathbb{IR}^m \times \mathbb{IR}^n \to \mathbb{IR}$. This problem setting corresponds to the case where an adversary could perturb any data observation by some bounded amount. In essence, we prove guaranteed bounds on all possible posteriors obtained after an adversarial attack on the data. Hence AURA can analyze and verify properties for an infinite number of probabilistic programs, a task which has not been studied in any prior work.

Optimization

The core idea of AURA is to compute the interval bounds, which are an element of $\mathbb{IR}$, by directly solving minimization and maximization problems. Hence instead of computing lower and upper bounds via interval arithmetic, AURA reduces abstract interpretation to continuous optimization. This optimization formulation is defined as:

$$[\![P]\!]^\sharp(x^\sharp, d^\sharp) = [l, u], \quad \text{where} \quad l = \min_{d \in \gamma(d^\sharp)} \min_{x \in \gamma(x^\sharp)} [\![P]\!](x, d), \quad \text{and} \quad u = \max_{d \in \gamma(d^\sharp)} \max_{x \in \gamma(x^\sharp)} [\![P]\!](x, d). \tag{5.9}$$

One can think of this formulation as defining an abstract transformer tailored for the *entire* program's (or subprogram's) expression instead of defining abstract transformers for individual primitive operations (as interval arithmetic does). Having an abstract transformer defined at this higher level of granularity allows AURA to improve precision greatly over interval arithmetic – and as we will show in Section 5.6, when the unnormalized likelihood

$[\![P]\!](x, d)$ has a pseudo-concave structure, we can solve this optimization problem tractably.

However one can always use any $[l^*, u^*]$ where $l^* \leq l$ and $u \leq u^*$ as sound bounds. Hence as we will see in Section 5.6.4, even for programs which *lack* the necessary pseudoconcavity properties one can always fallback to interval arithmetic to abstractly interpret the semantics of Fig. 5.11. This insight gives us the flexibility to analyze a (pseudo-concave) subprogram within P using AURA's precise optimization-based approach, while using interval arithmetic to bound other sub-expressions which may not be pseudoconcave and then compose the results. Section 5.6.4 presents an example of how we can compose the optimization-based abstraction with regular interval arithmetic. We can now state the following soundness result (the proofs in Appendix 5.7):

Theorem 5.1. The Unnormalized Abstract Semantics for data perturbation over-approximate the Unnormalized Abstract Semantics for fixed data observations. Equivalently for arbitrary program P, dataset $d^\sharp \in \mathbb{IR}^n$, and interval $x^\sharp \in \mathbb{IR}^m$, we have

$$\{[\![P]\!](x, d) : x \in \gamma(x^\sharp), d \in \gamma(d^\sharp)\} \subseteq \gamma([\![P]\!]^\sharp(x^\sharp, d^\sharp)) \tag{5.10}$$

Abstract Unnormalized Log-likelihood Semantics for Data Perturbations

We can take logarithmic transformations of the abstract semantics, even in the case of data perturbations to define $[\![P]\!]^\sharp_{\log}(x^\sharp, d^\sharp) = [l', u']$, where the bounds are defined as $l' = \min_{d \in \gamma(d^\sharp)} \min_{x \in \gamma(x^\sharp)} [\![P]\!]_{\log}(x, d)$, and $u' = \max_{d \in \gamma(d^\sharp)} \max_{x \in \gamma(x^\sharp)} [\![P]\!]_{\log}(x, d)$. So, $[\![P]\!]^\sharp(x^\sharp, d^\sharp)$ $= \exp([\![P]\!]^\sharp_{\log}(x^\sharp, d^\sharp)) = [\exp(l'), \exp(u')]$.

5.5.2 Normalized Abstract Semantics for Data Perturbation

One of the most challenging parts of computing the abstract normalized semantics is performing the abstract integration $\int^\sharp$. The key intuition is that the interval bounds of $[\![P]\!]^\sharp(x^\sharp, d^\sharp)$ form lower and upper Riemann sums, which can be used to bound the value of the integral, and thus bound the numeric range of the integrating constant.

Definition 5.6 (Abstract Integral with Data Perturbation)**.** We let each $x_i^\sharp$ represent a multi-dimensional interval in $\mathbb{IR}^m$, such that $\mathbb{R}^m = \cup_{i=1}^k \gamma(x_i^\sharp)$. Thus each $x_i^\sharp$ is a subset of the posterior distribution's support. Each $x_i^\sharp$ can be denoted as a Cartesian product (denoted as $\otimes$) of the intervals as $x_i^\sharp = \otimes_{j=1}^m [x_{l_{ij}}, x_{u_{ij}}]$. Here m represents the dimension of the latent variables, and for each dimension $j \in \{1, ..., m\}$, the interval $[x_{l_{ij}}, x_{u_{ij}}] \in \mathbb{IR}$ corresponds to that dimension. The volume (or Lebesgue measure in the multidimensional case) of each

$x_i^\sharp$ in the partition is given by $\text{Vol}(x_i^\sharp) = \prod_{j=1}^{k}(x_{u_{ij}} - x_{l_{ij}})$. The abstract integral with data perturbation is:

$$\int^\sharp [\![P]\!]^\sharp(x^\sharp, d^\sharp)\, dx = \left[\sum_{i=1}^{n} l_i \cdot \text{Vol}(x_i^\sharp), \sum_{i=1}^{n} u_i \cdot \text{Vol}(x_i^\sharp)\right] \tag{5.11}$$

where $\text{Vol}(x_i^\sharp) = \prod_{j=1}^{m}(x_{u_{ij}} - x_{l_{ij}})$ and $[\![P]\!]^\sharp(x_i^\sharp, d^\sharp) = [l_i, u_i]$ are the bounds defined in Section 5.5.1. In distributions with compact support, there will be finitely many non-zero terms in the summation. We will later see how this same idea can be used to abstractly integrate $[\![P]\!]_n^\sharp$ to formally bound probabilities of (measurable) events using the posterior bounds computed by $[\![P]\!]_n^\sharp$.

Lemma 5.1 (Soundness of abstract integration for data perturbations)**.** Similarly, given Theorem 5.1 and Definition 5.6, it follows that:

$$\int [\![P]\!](x, d)\, dx \in \gamma\left(\int^\sharp [\![P]\!]^\sharp(x^\sharp, d^\sharp)\, dx\right). \tag{5.12}$$

Theorem 5.2. The Normalized Abstract Semantics for data perturbation over-approximates the Normalized Concrete Semantics for sets of data observations. More formally, for a program P, dataset $d^\sharp \in \mathbb{IR}^n$, and interval $x^\sharp \in \mathbb{IR}^m$, we have:

$$\{[\![P]\!]_n(x, d) : x \in \gamma(x^\sharp), d \in \gamma(d^\sharp)\} \subseteq \gamma([\![P]\!]_n^\sharp(x^\sharp, d^\sharp)). \tag{5.13}$$

While one might think to just run AURA on concrete data points l_d and u_d where $[l_d, u_d] = d^\sharp$, to compute $[\![P]\!]_n^\sharp(x^\sharp, l_d)$ and $[\![P]\!]_n^\sharp(x^\sharp, u_d)$ and then take the union instead of computing $[\![P]\!](x^\sharp, d^\sharp)$, that strategy would be unsound since the posterior bounds are not monotonic with respect to the data variable d. Hence AURA's analysis is necessary when considering data perturbations.

5.6 AURA OPTIMIZATION AND VERIFICATION ALGORITHM

Having defined the abstract semantics $[\![P]\!]^\sharp(x^\sharp, d^\sharp)$ in terms of interval bounds where the lower and upper bounds come from solutions to optimization problems, we now describe how AURA can precisely solve these optimization problems. However, we first need to define the key mathematical properties that the probabilistic programs must satisfy.

5.6.1 Pseudoconcave Probabilistic Programs

A probabilistic program P is *pseudoconcave* if the unnormalized density function defined by $[\![P]\!](x,d)$ is a pseudoconcave function of both x and d (a condition satisfied by many common distributions; see Appendix B.1). Hence, in light of the definitions in Section 5.3.2, we have the following implications:

$$[\![P]\!](x,d) \textbf{ Log-Concave} \implies [\![P]\!]_{\log}(x,d) \textbf{ Concave} \implies [\![P]\!]_{\log}(x,d) \textbf{ Pseudoconcave} \tag{5.14}$$

$$[\![P]\!](x,d) \textbf{ Pseudoconcave} \implies [\![P]\!]_{\log}(x,d) \textbf{ Pseudoconcave} \tag{5.15}$$

We choose **Pseudoconcavity**, because to the best of our knowledge it is the weakest condition that still ensures the lower and upper bounds from gradient-based optimizations on unnormalized posteriors are still sound. A useful property of pseudoconcave functions, and thus pseudoconcave probabilistic programs, is that their derivatives exist everywhere except for a measure zero set. Hence we have that:

Lemma 5.2. A pseudoconcave $[\![P]\!]_{\log}(x,d)$ is almost everywhere differentiable.

Since the derivatives exist (almost) everywhere, we will be able to compute them and subsequently use them for Gradient Ascent to solve the optimization problems of Section 5.5.1.

5.6.2 Computing Lower Bounds with AURA

The first step in computing the abstract semantics $[\![P]\!]^\sharp(x^\sharp, d^\sharp)$ needed to soundly bound posteriors involves computing the interval's lower bound. In the case of data perturbations one must compute $l = \min_{d\in\gamma(d^\sharp)} \min_{x\in\gamma(x^\sharp)} [\![P]\!](x,d)$. However, because of the pseudoconcavity requirements on $[\![P]\!](x,d)$ this minimization problem becomes tractable. In particular we only have to check the corner points of $x^\sharp$ and $d^\sharp$, denoted as $Corners$. Furthermore, for numerical stability, we will work with logarithms, hence we can solve the optimization problem by computing:

$$l = \exp\left(\min_{d\in Corners(d^\sharp)} \min_{x\in Corners(x^\sharp)} [\![P]\!]_{\log}(x,d) \right) \tag{5.16}$$

Theorem 5.3. (Soundness) The lower bounds l computed above in Eq. 5.16 are sound when the log-likelihood $[\![P]\!]_{\log}(x,d)$ is pseudoconcave.

Proof. Any pseudoconcave function is also quasiconcave, and quasiconcave functions over compact convex sets can be minimized by checking corner points [161]. QED.

Indeed, a key benefit of using the Interval abstract domain instead of the Polyhedral or Zonotope domains is that checking extremal points of intervals is more straightforward and efficient compared to checking extremal points of polyhedra or zonotopes. Furthermore, this lower bound is not just sound, but it is **optimal**, meaning it is impossible to have a tighter lower bound for the function on that interval. We state this result below:

Corollary 5.1. (Optimality) the lower bound computed in Eq. 5.16 is the most precise bound possible.

Proof. (sketch) Because $[\![P]\!]_{\log}(x, d)$ is continuous and the interval $d^\sharp \times x^\sharp$ is compact, the minimum (not just an infimum) will be attained on that interval. Additionally, since exp is monotonically, increasing, the minimizer of $[\![P]\!]_{\log}(x, d)$ will also be the minimizer of $\exp([\![P]\!]_{\log}(x, d))$ QED.

5.6.3 Computing Upper Bounds with AURA

Similarly, in order to obtain sound enclosures, AURA also needs to compute the upper bound by solving the following optimization problem $u = \max_{d \in \gamma(d^\sharp)} \max_{x \in \gamma(x^\sharp)} [\![P]\!](x, d)$. Our key technical insight is that this maximization problem can be solved directly by performing *Projected Gradient Ascent* on the log likelihood, $[\![P]\!]_{\log}(x, d)$. Further, since $x^\sharp$ and $d^\sharp$ define (multi-dimensional) intervals, they are convex sets, hence the constraints of this optimization problem are convex.

Definition 5.7. Projected Gradient Ascent. Given a differentiable function $f(x) : \mathcal{X} \subset \mathbb{R}^m \to \mathbb{R}$, one iteratively computes:

$$x_{n+1} = \Pi_{\mathcal{X}}\big(x_n + \eta \nabla_x f(x_n)\big) \tag{5.17}$$

with learning rate $\eta \in \mathbb{R}_{>0}$ until convergence where $\|x_{n+1} - x_n\| \leq \epsilon$. Here $\Pi_{\mathcal{X}}$ is the projection operator that takes a x_{n+1} that may lie outside $\mathcal{X}$, and returns the closest point inside $\mathcal{X}$. If the constraints are intervals: $\mathcal{X} = \otimes_{i=1}^m [l_i, u_i] \subseteq \mathbb{IR}^m$, the projection is $\Pi_{\mathcal{X}}(x) = \otimes_{i=1}^m \Pi_{\mathcal{X}}(x[i])$ where:

$$\Pi_{\mathcal{X}}(x[i]) = \begin{cases} l_i & l_i > x[i] \\ x[i] & x[i] \in [l_i, u_i] \\ u_i & u_i < x[i] \end{cases} \tag{5.18}$$

AURA Gradient Optimization AURA will run the following Gradient Ascent computations for the function $[\![P]\!]_{\log}(x, d) : \gamma(x^\sharp) \times \gamma(d^\sharp) \to \mathbb{R}_{\geq 0}$:

$$(x_{n+1}, d_{n+1}) = \Pi_{x^\sharp;d^\sharp}\big((x_n, d_n) + \eta \nabla [\![P]\!]_{\log}(x_n, d_n)\big) \tag{5.19}$$

until $x_{n+1} = x_n$ and $d_{n+1} = d_n$. The learning rate must satisfy $\eta \leq \frac{1}{\|\nabla [\![P]\!]_{\log}(x_n, d_n)\|}$ to ensure convergence. We further discuss the selection of the learning rate in Section 5.8.2. This optimization problem is constrained because the latent parameters x come from distributions with compact support (e.g., uniform). A key benefit of using the interval domain is that the projection function $\Pi_{x^\sharp;d^\sharp}$ in Eq. 5.19 reduces to the (efficiently computable) projection in Eq. 5.18 since $x^\sharp \times d^\sharp$ is just a multi-dimensional interval. Upon computing the $x_{n+1} = x_n$ and $d_{n+1} = d_n$ that the Projected Gradient Ascent converges to, we exponentiate the result to compute:

$$u = \exp([\![P]\!]_{\log}(x_{n+1}, d_{n+1})) \tag{5.20}$$

Theorem 5.4. (Soundness) The upper bounds u computed in Eq. 5.20 are sound when the log-likelihood $[\![P]\!]_{\log}(x, d)$ is pseudoconcave.

Proof. (sketch) By Lemma 5.2, gradient ascent is well-defined for pseudoconcave $[\![P]\!]_{\log}(x, d)$. Furthermore, projected gradient ascent/descent is guaranteed to converge to the *true* maxima (instead of a local one) for pseudoconcave/pseudoconvex functions [162, 163, 164]. Lastly, when projected gradient ascent applied to a pseudoconcave function finds a fixed point $x_{n+1} = x_n$, $d_{n+1} = d_n$, such a fixed point is guaranteed to the be the true optima ([164] Theorem 4.2) QED.

Corollary 5.2. The upper bound computed in Eq. 5.20 is the most precise upper bound possible.

Proof. (sketch) Since $[\![P]\!]_{\log}(x, d)$ is continuous and the interval $d^\sharp \times x^\sharp$ is compact, the maximum (not just a supremum) will be attained on that interval. Additionally, since exp is monotonically, increasing, the maximizer of $[\![P]\!]_{\log}(x, d)$ will also be the maximizer of $\exp([\![P]\!]_{\log}(x, d))$ QED.

AURA runs the gradient ascent until a fixed point ($x_{n+1} = x_n$) is found. Alternatively, it can run the gradient ascent for fewer iterations (before convergence to a fixed point), and add an error bound to the result to account for the distance to the true optimum value. For instance, prior works [164, 165] have guaranteed that for pseudo-concave function $f(x) : \mathcal{X} \subset \mathbb{R}^m \to \mathbb{R}$, after T iterations of projected gradient ascent, the error between

the true maximum and the current obtained values will not exceed $E = \sqrt{\kappa^2 \|x_0 - x^*\|^2 / T}$, where x_0 represents the starting point, x^* denotes the true maximum, and κ denotes the local Lipschitz constant over $\mathcal{X}$ that can be easily computed by AURA. Hence, even without a sufficient number of iterations to reach the fixed point, when adding tiny bound E, AURA can still give bounds which are sound. This strategy is also applicable if one wishes to account for numerical roundoff error.

Lastly, because of the optimality of the lower and upper bounds, AURA achieves the most precise abstract transformer of pseudoconcave functions for the interval domain.

5.6.4 Beyond Pseudoconcavity: Compositionality and Interval Abstract Domain

Supporting Branch Statements and Mixture Models. While pseudoconcave likelihoods lead to precise and tractable gradient-based abstract interpretation, the question arises of how to support posteriors that are *not* pseudoconcave. This scenario is encountered in popular mixture distributions that result from the branching primitive, `if flip`(p) P_1 `else` P_2, in our language.

We will show that AURA still supports mixture distributions that contain such branches which can cause the likelihoods to no longer be pseudoconcave. The key technical insight is that even if the entire posterior is multi-modal and thus not pseudoconcave, each *component* when viewed in isolation could be pseudoconcave. Thus by applying AURA's abstract interpreter, $[\![\cdot]\!]^\sharp$, to each component (which *will* be pseudoconcave) and then combining the results with standard interval arithmetic, we can still obtain sound posterior bounds. Indeed, we recall from Fig. 5.11 that:

$$[\![\texttt{if flip}(p)\ P_1\ \texttt{else}\ P_2]\!](x, d) = p \cdot [\![P_1]\!](x, d) + (1 - p) \cdot [\![P_2]\!](x, d) \tag{5.21}$$

Thus for this primitive, we will define the unnormalized *abstract* semantics as:

$$[\![\texttt{if flip}(p)\ P_1\ \texttt{else}\ P_2]\!]^\sharp(x^\sharp, d^\sharp) = p \cdot^\sharp [\![P_1]\!]^\sharp(x^\sharp, d^\sharp) +^\sharp (1 - p) \cdot^\sharp [\![P_2]\!]^\sharp(x^\sharp, d^\sharp) \tag{5.22}$$

Hence by applying the abstract interpreter to P_1 and P_2, each of which *will* be pseudoconcave, the computation of $[\![P_1]\!]^\sharp(x^\sharp, d^\sharp)$ and $[\![P_2]\!]^\sharp(x^\sharp, d^\sharp)$ can use AURA's gradient based optimization to obtain tight bounds. Given the soundness of interval addition $+^\sharp$ and multiplication $\cdot^\sharp$, we obtain:

Lemma 5.3. When P_1 and P_2 are pseudoconcave, based on Theorem 5.1, the preservation of soundness by interval addition and interval multiplication, AURA's abstraction for a

branching program is sound:

$$\{[\![\texttt{if flip}(p)\ P_1\ \texttt{else}\ P_2]\!](x,d) : x \in \gamma(x^\sharp), d \in \gamma(d^\sharp)\} \subseteq \gamma([\![\texttt{if flip}(p)\ P_1\ \texttt{else}\ P_2]\!]^\sharp(x^\sharp, d^\sharp)) \tag{5.23}$$

Once we have these sound bounds on the *unnormalized* posterior, we can pass them as input to the abstract integration, and thus bound the *normalized* posterior for programs with branches.

Lemma 5.4. Under data perturbation, the soundness Theorem 5.2, still holds for a program P that consists of a mixture of pseudoconcave branches .

$$\{[\![\texttt{if flip}(p)\ P_1\ \texttt{else}\ P_2]\!]_n(x,d) : x \in \gamma(x^\sharp), d \in \gamma(d^\sharp)\} \subseteq \gamma([\![\texttt{if flip}(p)\ P_1\ \texttt{else}\ P_2]\!]_n^\sharp(x^\sharp, d^\sharp)) \tag{5.24}$$

Pseudoconcave Subexpressions . Beyond using interval arithmetic for the addition and multiplication operations in Eq. 5.22, *since AURA uses the interval domain, we can fallback to interval arithmetic for* any *of the subexpressions in Fig. 5.11.* Indeed, the baseline in Section 5.10.2 uses a standard interval abstract interpretation for all expressions. Thus, even if the program P lacks a pseudoconcave posterior, we can compute optimized bounds on the largest subexpressions which *are* pseudoconcave, and then use standard interval arithmetic for the rest. Hence Sections 5.6.2 and 5.6.3 provide sound interval domain abstract transformers for *any* pseudoconcave function, including subexpressions within P. We state this property in Theorem 5.5:

Theorem 5.5. Let M be a prior subexpression in program P. If $[\![M]\!](x,d)$ is a pseudoconcave function and if the lower bounds are computed as in Section 5.6.2 and upper bounds computed as in Section 5.6.3, then the following bounds are sound

$$[\![M]\!]^\sharp(x^\sharp, d^\sharp) = [l,u] \quad \textit{where} \quad l = \min_{x\in\gamma(x^\sharp)} \min_{d\in\gamma(d^\sharp)} [\![M]\!](x,d) \qquad u = \max_{x\in\gamma(x^\sharp)} \max_{d\in\gamma(d^\sharp)} [\![M]\!](x,d) \tag{5.25}$$

Similarly let D be an observation subexpression in program P. If $[\![D]\!](x,d)$ is pseudoconcave, then the bounds computed as in Sections 5.6.2 and 5.6.3 are sound:

$$[\![D]\!]^\sharp(x^\sharp, d^\sharp) = [l,u] \quad \textit{where} \quad l = \min_{x\in\gamma(x^\sharp)} \min_{d\in\gamma(d^\sharp)} [\![D]\!](x,d) \qquad u = \max_{x\in\gamma(x^\sharp)} \max_{d\in\gamma(d^\sharp)} [\![D]\!](x,d) \tag{5.26}$$

The soundness of the lower bounds for these subexpressions (instead of full programs) follows identically to Theorem 5.3 and for upper bounds identically to Theorem 5.4. We

can now define $[\![M]\!]^\sharp_{best}$ and $[\![D]\!]^\sharp_{best}$ as abstract transformers that use the precise optimized bounds of Theorem 5.5 for the entire subexpression M or D if it is pseudoconcave, otherwise they will default to recursively evaluating the subexpression in interval arithmetic $\mathbb{IR}$ for subexpressions which are not pseudoconcave.

$$[\![M]\!]^\sharp_{best}(x^\sharp, d^\sharp) = \begin{cases} [\![M]\!]^\sharp_{\mathbb{IR}} & [\![M]\!](x,d) \text{ is not pseudoconcave} \\ [\![M]\!]^\sharp & [\![M]\!](x,d) \text{ is pseudoconcave} \end{cases} \tag{5.27}$$

where $[\![M]\!]^\sharp_{\mathbb{IR}}$ evaluates $[\![M]\!]$ using standard interval arithmetic.

$$[\![D]\!]^\sharp_{best}(x^\sharp, d^\sharp) = \begin{cases} [\![D]\!]^\sharp_{\mathbb{IR}} & [\![D]\!](x,d) \text{ is not pseudoconcave} \\ [\![D]\!]^\sharp & [\![D]\!](x,d) \text{ is pseudoconcave} \end{cases} \tag{5.28}$$

where similarly, $[\![D]\!]^\sharp_{\mathbb{IR}}$ evaluates $[\![D]\!]$ using standard interval arithmetic.

In light of this definition, we can now reformalize AURA's abstract interpreter (for unnormalized semantics) to use the optimized bounds for pseudo-concave sub-expressions, even when the full program's expression $[\![P]\!](x,d)$ is not pseudo-concave. The interval results of the sub-expressions can then be combined using interval arithmetic, particularly interval multiplication $\cdot^\sharp$. This reformalization is shown in Fig. 5.12.

$$[\![P]\!]^\sharp(x,d) = [\![M;D]\!]^\sharp(x^\sharp, d^\sharp) \tag{5.29}$$
$$[\![M;D]\!]^\sharp(x^\sharp, d^\sharp) = [\![M]\!]^\sharp_{best}(x^\sharp) \cdot^\sharp [\![D]\!]^\sharp_{best}(x^\sharp, d^\sharp) \tag{5.30}$$
$$[\![D;D]\!]^\sharp(x^\sharp, d^\sharp) = [\![D]\!]^\sharp_{best}(x^\sharp, d^\sharp) \cdot^\sharp [\![D]\!]^\sharp_{best}(x^\sharp, d^\sharp) \tag{5.31}$$
$$[\![M;M]\!]^\sharp(x^\sharp) = [\![M]\!]^\sharp_{best}(x^\sharp) \cdot^\sharp [\![M]\!]^\sharp_{best}(x^\sharp) \tag{5.32}$$

Figure 5.12: Reformalization of AURA's unnormalized abstract semantics for programs which are not completely pseudoconcave

Thus AURA supports a compositional analysis even when the full program P is not pseudoconcave. Indeed, a key benefit of our abstract interpretation-based approach is that it allows us to compose our precise, optimized bounds (for subexpressions that are pseudoconcave), with standard interval arithmetic (for subexpressions that are not pseudoconcave).

Theorem 5.6. (Soundness) The unnormalized posterior bounds computed in Fig. 5.12 are sound.

Algorithm 5.1 AURA Core Algorithm

Input: Probabilistic Program P, abstract dataset $d^\sharp$, partition of P's support $\bigcup_{i=1}^{m} x_i^\sharp$, query Q (optional)
Output: Posterior Bounds $[\![P]\!]_n^\sharp(x, d^\sharp)$, $[Q_l, Q_u]$ where $Pr(Q) \in [Q_l, Q_u]$ (only if query provided)

1: **for** $x_i^\sharp$ in splits **do**
2: $\quad l_i, u_i = [\![P]\!]^\sharp(x_i^\sharp, d^\sharp)$ $\triangleright$ Unnormalized posterior bounds computed in Section 5.6.2-5.6.4 (data parallel)
3: $[c_l, c_u] = \int^\sharp [\![P]\!]^\sharp(x^\sharp, d^\sharp)dx$ $\triangleright$ Normalizing Constant bound computed in Def. 5.6 (reduction)
4: **for** $x_i^\sharp$ in splits **do**
5: $\quad [\![P]\!]_n^\sharp(x_i^\sharp, d^\sharp) = [l_i, u_i] \div^\sharp [c_l, c_u]$ $\triangleright$ Normalization (data parallel)
6: **if** Query **then**
7: $\quad [Q_l, Q_u] = \int^\sharp \mathbf{1}_Q \cdot [\![P]\!]_n^\sharp(x^\sharp, d^\sharp)dx$ $\triangleright$ Abstract integration of posterior from Def. 5.35 (reduction)

This soundness guarantee follows from the fact that compositions of sound abstract transformers are still sound [160].

Example 5.1. *Let $P \triangleq M; D$, as in Fig. 5.9 where M and D are arbitrary subexpressions. To bound $[\![P]\!](x, d)$ we bound $[\![M; D]\!](x, d)$. However when $[\![M]\!](x, d)$ is pseudoconcave but $[\![D]\!](x, d)$ is not, we compute $[\![M]\!]^\sharp(x^\sharp, d^\sharp) \cdot^\sharp [\![D]\!]_{\mathbb{IR}}(x^\sharp, d^\sharp)$ where $[\![M]\!]^\sharp(x^\sharp, d^\sharp)$ uses the bounds of Theorem* 5.5 *and $[\![D]\!]_{\mathbb{IR}}(x^\sharp, d^\sharp)$ is the standard interval arithmetic abstraction and $\cdot^\sharp$ is interval multiplication.*

5.6.5 AURA Verification Algorithm

Equations 5.16 and 5.20 provide a strategy to solve the optimization problems of Section 5.5.1, thus giving a way to compute $[\![P]\!]^\sharp(x^\sharp, d^\sharp)$. Further, Section 5.6.4 shows how to support programs with branches. Hence we now give the full algorithm for how AURA abstractly interprets entire posterior distributions of probabilistic programs. The entire procedure is shown in Algorithm 5.1.

Partitioning (Splits). The core intuition is that we will actually partition the support of P's probability distribution into disjoint interval "*splits*". If $x^\sharp$ represents the interval containing the entire (compact) support of P's (unnormalized) likelihood, then we will take partitions $x_i^\sharp$ such that $x^\sharp = \bigcup_i x_i^\sharp$. We use an equal-area splitting strategy but support other strategies (see Section 5.6.6 and 5.8.3). For each split $x_i^\sharp$, AURA computes $[\![P]\!]^\sharp(x_i^\sharp, d^\sharp)$ in lines 1-2. A core implementation insight is that each interval split $x_i^\sharp$ can be analyzed in parallel for the computation $[\![P]\!]^\sharp(x_i^\sharp, d^\sharp)$.

Normalization. To obtain bounds on the normalized posterior $[\![P]\!]_n^\sharp$ using the bounds on the unnormalized posterior $[\![P]\!]^\sharp$, AURA must perform the abstract integration to bound the normalizing constant using the strategy in Def. 5.6. The partitions that we used in

the previous step can be used for lower and upper Riemann sums (line 3) as mentioned in Section 5.5.2. Upon computing the normalizing constant bound $[c_l, c_u]$, AURA then performs interval division to normalize the unnormalized posterior bound of each split (lines 4-5).

Certified Bounds on Probabilistic Queries. AURA can then use the normalized posterior bounds $[\![P]\!]^\sharp_n$ to certify bounds on the posterior probability of different queries. Under the data perturbation model, the bounds computed by $[\![P]\!]^\sharp_n$ enclose not just a single posterior distribution (like in [26]) but rather an *infinite* number of posteriors. Hence whenever AURA computes bounds on the probabilities of queries, these bounds also hold for an infinite set of possible posteriors.

Definition 5.8. Queries. A query Q is a logical formula over the variables of P given by the following grammar:

$$Q ::= \mathtt{x}_j \geq c \mid \mathtt{x}_j \leq c \mid Q \wedge Q \mid Q \vee Q \tag{5.33}$$

The queries define measurable events, hence we can define the posterior distribution's probability of a particular query Q. This probability is defined as:

$$Pr_{x \sim [\![P]\!]_n(\cdot, d)}(Q) = \int_x \mathbf{1}_Q \cdot [\![P]\!]_n(x, d)\, dx, \tag{5.34}$$

where $\mathbf{1}_Q$ is the binary indicator function for the event Q. However as in Section 5.4, this integral is not computable in the general case, hence AURA will *over-approximate* this probability. The over-approximation will be computed using an abstract integration similar to Def. 5.6. The key difference is that we will use the *normalized* posterior interval bounds, $[\![P]\!]^\sharp_n$, instead of the unnormalized bounds $[\![P]\!]^\sharp$ that Def. 5.6 uses. The new abstract integration is given as:

$$\int^\sharp \mathbf{1}_Q \cdot [\![P]\!]^\sharp_n(x, d^\sharp) dx = \sum_i [\![P]\!]^\sharp_n(x^\sharp_i, d^\sharp) \cdot Vol(x^\sharp_i \cap \{x : Q\}) = [l_Q, u_Q] \tag{5.35}$$

The summation ($\sum$) is interval addition and the $Vol(\cdot)$ function computes the Lebesgue measure of the input set. Since each $x^\sharp_i$ is an interval, and the set $\{x : Q\}$ is a union or intersection of finitely many intervals, the result of $x^\sharp_i \cap \{x : Q\}$ is itself a union or intersection of finitely many intervals and thus its Lebesgue measure can be computed easily. Hence Eq. 5.35 ultimately computes an interval that encloses the true integral. We can now state the soundness result (proof in Section 5.7):

Theorem 5.7. (Soundness) For probabilistic program P, dataset $d \in \gamma(d^\sharp)$ and Query Q

we have:

$$Pr_{x\sim[\![P]\!]_n(\cdot,d)}(Q) \in \int^{\sharp} \mathbf{1}_Q \cdot [\![P]\!]_n^{\sharp}(x, d^{\sharp})dx \tag{5.36}$$

Parallelization. All algorithm steps are parallelizable on CPU or GPU: the two for-loops are data-parallel, while abstract integration and computing bounds on queries are reductions.

5.6.6 Implementation

We implemented AURA to strike a balance between precision, efficiency and scalability. A key strategy involves leveraging GPUs for parallelizing interval splits and processing vectorized data. AURA is implemented using PyTorch (v1.9.1), supporting both GPU and CPU backends. For determining the splits as inputs for Algorithm 5.1, we use a custom precision enhancing heuristic. Rather than making each split $x_i^{\sharp}$ be the same width, AURA calibrates the splits based on estimations of the posterior's curvature, using a pre-analysis with a small number of splits; detailed in Section 5.8.3.

AURA supports a wide range of known distributions, including *normal*, *uniform*, *gamma*, *exponential*, *bernoulli*, *logistic*, *laplace*, *beta*, *bernoulli_logit*, and *bernoulli_probit*. In addition, for infinite support distributions (e.g. `normal`) AURA uses truncated versions in the priors to ensure compact support (as also done by GuBPI [26]), which we denote with a subscript t. The observed distributions need not be truncated. While AURA's implementation assumes ideal real arithmetic (as is common in ML verification, e.g., [26, 166, 167]), our evaluation shows that numerical imprecision resulting from floating-point is small. Furthermore, our implementation could be directly extended to soundly account for floating-point roundoff error by using existing techniques [168] or by using arbitrary precision numerical libraries, e.g., NVIDIA XMP [169] for GPUs.

To scale AURA for large datasets common in modern applications, which may exceed the memory of a single GPU, we crafted software tiling and sharding to distribute computations across multiple GPUs. We devised a method for integration of local posterior tiles, eliminating the need to store and communicate all posterior tiles between devices.

5.7 SOUNDNESS PROOFS

We first give a Soundness Proof for Theorem 5.1.

Proof. Since $[\![P]\!]^{\sharp}(x^{\sharp}, d^{\sharp}) = [l, u]$ is a 1D interval in $\mathbb{IR}$, we have to show that

$$l \leq \inf\{[\![P]\!](x, d) : x \in \gamma(x^{\sharp}), d \in \gamma(d^{\sharp})\} \tag{5.37}$$

$$u \geq \sup\{[\![P]\!](x,d) : x \in \gamma(x^\sharp), d \in \gamma(d^\sharp)\} \tag{5.38}$$

However by definition:

$$l = \min_{d \in \gamma(d^\sharp)} \min_{x \in \gamma(x^\sharp)} [\![P]\!](x,d) \tag{5.39}$$

and

$$\min_{d \in \gamma(d^\sharp)} \min_{x \in \gamma(x^\sharp)} [\![P]\!](x,d) = \inf\{[\![P]\!](x,d) : x \in \gamma(x^\sharp), d \in \gamma(d^\sharp)\} \tag{5.40}$$

Similarly

$$u = \max_{d \in \gamma(d^\sharp)} \max_{x \in \gamma(x^\sharp)} [\![P]\!](x,d) \tag{5.41}$$

and

$$\max_{d \in \gamma(d^\sharp)} \max_{x \in \gamma(x^\sharp)} [\![P]\!](x,d) = \sup\{[\![P]\!](x,d) : x \in \gamma(x^\sharp), d \in \gamma(d^\sharp)\} \tag{5.42}$$

Hence soundness follows (by construction) QED.

We now give a Soundness Proof for Lemma 5.1

Proof. To prove

$$\int [\![P]\!](x,d)\,dx \in \gamma\left(\int^\sharp [\![P]\!]^\sharp(x^\sharp, d^\sharp)\,dx\right). \tag{5.43}$$

We need to show that

$$\sum_{i=1}^{n} l_i \cdot \mathrm{Vol}(x_i^\sharp) \leq \int [\![P]\!](x,d)dx \leq \sum_{i=1}^{n} u_i \cdot \mathrm{Vol}(x_i^\sharp) \tag{5.44}$$

However the left hand side is really just a lower Riemann sum which is always less than the true integral, and likewise the right hand side is just an upper Riemann sum which is always larger than the true integral. QED.

We now provide a proof of Theorem 5.2

Proof. Since the unnormalized bounds are sound from Theorem 5.1, the bounds on the normalizing constant are sound from Lemma 5.1 and interval division is sound, the normalized bounds must be sound too. QED.

We now provide a proof of Theorem 5.7

Proof.

$$\int^{\sharp} \mathbf{1}_Q \cdot [\![P]\!]^{\sharp}_n(x, d^{\sharp})dx = \sum_i [\![P]\!]^{\sharp}_n(x^{\sharp}_i, d^{\sharp}) \cdot Vol(x^{\sharp}_i \cap \{x : Q\}) \tag{5.45}$$

$$= \sum_i [l_i, u_i] \cdot Vol(x^{\sharp}_i \cap \{x : Q\}) \tag{5.46}$$

$$= \sum_i [l_i \cdot Vol(x^{\sharp}_i \cap \{x : Q\}), u_i \cdot Vol(x^{\sharp}_i \cap \{x : Q\})] \tag{5.47}$$

$$= [\sum_i l_i \cdot Vol(x^{\sharp}_i \cap \{x : Q\}), \sum_i u_i \cdot Vol(x^{\sharp}_i \cap \{x : Q\})] \tag{5.48}$$

Where the lower and upper bounds are just lower and upper Riemann sums, hence they enclose $\int \mathbf{1}_Q \cdot [\![P]\!]_n(x, d)dx$ which is exactly just $Pr_{x\sim[\![P]\!]_n(\cdot,d)}$. QED.

5.8 IMPLEMENTATION AND OPTIMIZATIONS

We now describe the implementation of AURA and program optimizations used to ensure efficiency and scalability.

5.8.1 GPU Parallelization

A core implementation insight is that each interval split $x^{\sharp}_i$ can be analyzed in parallel for the computation $[\![P]\!]^{\sharp}(x^{\sharp}_i, d^{\sharp})$. Intuitively this means that both `for` loops of Algorithm 5.1 can be parallelized, as well as the abstract integration (i.e., summation) that are standard reductions. For parallelizing these four stages, AURA uses PyTorch with GPU and CPU backends.

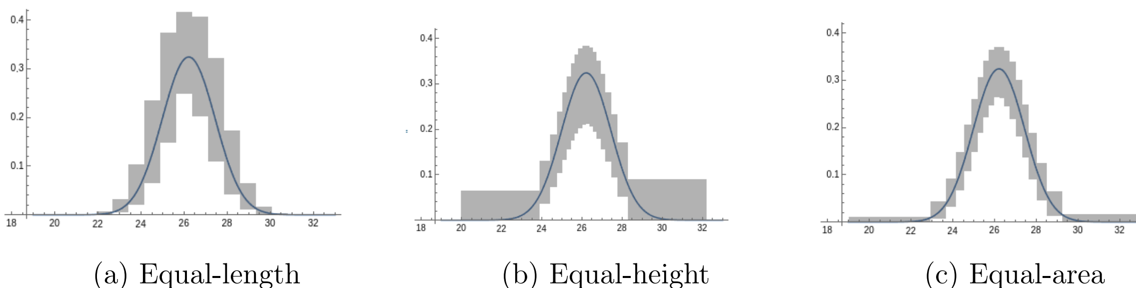
5.8.2 Efficient Lipschitz Analysis for Learning Rate Selection

A notable implementation detail of AURA's gradient based optimization is choosing an appropriate learning rate η. This choice also has theoretical relevance since as mentioned in Section 5.6.3, the magnitude of η affects the convergence guarantees of gradient descent. PyTorch's Automatic Differentiation allows us to compute the Lipschitz constant (LC) of the unnormalized posterior $[\![P]\!]$, which can be used to determine η. The LC is bounded by the largest gradient norm (for the local region of the split), and for known distributions, this maximum gradient will occur at the boundaries. Furthermore, using the rules from [32],

we aggregate the LCs of individual distributions and estimate the one for the unnormalized posterior.

5.8.3 Precision-Enhancing Splitting

As mentioned, AURA performs partitions the parameter space into splits $x_i^\sharp$, which can be analyzed in parallel as described in Section 5.8.1. However one question we ask is what is the *best* strategy for this splitting. The most obvious strategy is to perform the splitting such that each split $x_i^\sharp$ has the same width, however as we will see, this does not necessarily lead to the best precision. Indeed, while different splitting heuristics do not affect the soundness, they *do* affect the precision.



(a) Equal-length (b) Equal-height (c) Equal-area

Figure 5.13: Example of the Analysis Results by Different Splitting Strategies

As an illustration of this phenomenon, Figure 5.13 shows the resulting bounds obtained from three different heuristics each using 20 splits on a simple regression model (lightspeed). We show the ground truth with a blue line and the bounds found by AURA with gray boxes in order to highlight the differences in precision between the three heuristics. Figure 5.13a shows the result from the equal-length strategy , which divides the variable interval in equal-length sub-intervals. Due to its simplicity, this strategy is widely used, e.g. by the baseline GuBPI. However, as we observed from the plot, while the bounds at the tails are reasonably precise, the bounds around the mode (middle part) of the curve are very loose and imprecise.

Another strategy we tried is the equal-height splits (Figure 5.13b), which aims to divide the intervals such that the resulting bounds have the same height. Without knowing the true bounds beforehand, AURA could run a separate analysis (like AQUA [61]) once with a small number of splits (e.g. 60 splits with equal-length) to estimate the shape of the curve, and then use those results to decide the splits with approximately equal height. Nevertheless, this strategy is imprecise for bounding the tails of the distribution.

Therefore, we designed the third strategy to strike an ideal balance between precisely bounding the tails and precisely bounding the mode. In Figure 5.13c, we generate splits that results in similar the area for each bounding box. We estimate the shape of the curve,

and use the results to generate splits with approximately equal area. We use this equal-area strategy as the default in AURA analysis, however AURA also supports the other two strategies and allows customized splits.

5.9 METHODOLOGY

Benchmarks

We evaluated AURA, alongside with GuBPI and PSI on 19 benchmarks from existing literature with diverse program structure and distributions. We show the details in Table 5.1. For each program, we manually verified its pseudo-concavity. We present the details of checking in Appendix B.1. One can also automatically check for pseudo-concavity, e.g., using [170].

Table 5.1: Benchmark program details. Symbols used: $\mathcal{B}$: Bernoulli, $\mathcal{B}_{\log}$: Bernoulli-Logit, $\mathcal{B}_{pro}$: Bernoulli-Probit, $\mathcal{U}$: Real Uniform, $\mathcal{N}$: Normal, β: Beta, $\mathcal{L}$: Laplace, $\mathcal{S}$: Logistic. Operators: +: mix of distributions, ×: product of densities. E.g., human_height's **Prior** is $\mathcal{B} \times (\mathcal{N}_t + \mathcal{N}_t)$: a mixture of truncated normal distributions chosen from a Bernoulli distribution; likelihoods (**Lik**) are $\mathcal{N}^3$: three observations from normal distribution. In **PC** column, $\checkmark_{PC}$ shows Pseudoconcavity, $\checkmark_{LC}$ shows Log-Concavity, $\checkmark_{mix}$ for a mixture of Pseudoconcave functions.

Program	Prior	Lik	Description	PC
exponential	Exponential_t	N/A	An Exponential distribution	$\checkmark_{LC}$
gamma	Gamma_t	N/A	A Gamma distribution	$\checkmark_{LC}$
gaussian	$\mathcal{N}_t$	N/A	A Normal distribution	$\checkmark_{LC}$
coinBias	β	$\mathcal{B}^5$	Bias of coin using Beta-Bernoulli model [26]	$\checkmark_{LC}$
human_height	$\mathcal{B} \times (\mathcal{N}_t + \mathcal{N}_t)$	$\mathcal{N}^3$	Learning height with mixture prior [171]	$\checkmark_{mix}$
clinicalTrial	$\mathcal{B} \times (\beta + \beta)$	$\mathcal{B}_{\log}^{10}$	Logistic regression with mixed prior [172]	$\checkmark_{mix}$
altermu2	$\mathcal{U}^2$	$\mathcal{N}^{40}$	Model with param symmetry [61]	$\checkmark_{LC}$
personality	$\mathcal{S}_t$	${\mathcal{B}_{\log}}^{1000}$	Logistic regression for cheating study [173]	$\checkmark_{LC}$
reg_logistic	$\mathcal{N}_t$	$\mathcal{S}^{919}$	Linear regression with logistic likelihood [174]	$\checkmark_{LC}$
privacy	$\mathcal{N}_t^2$	$\mathcal{N}$	Regression estimates age from group mean [129]	$\checkmark_{LC}$
logistic	$\mathcal{U}^2$	$\mathcal{B}_{\log}^{100}$	Logistic regression [50]	$\checkmark_{LC}$
lightspeed	$\mathcal{U}^2$	$\mathcal{N}^{40}$	Linear regression [50]	$\checkmark_{PC}$
anova_radon_n	$\mathcal{U}^2$	$\mathcal{N}^{40}$	Hierarchical linear regression, non-predictive [50]	$\checkmark_{PC}$
IQStan	$\mathcal{U}^3$	$\mathcal{N}^3 \times \mathcal{N}^3$	Regression on two datasets with shared variance [175]	$\checkmark_{PC}$
reg_laplace	$\mathcal{U}^2 \times {\mathcal{L}_t}^2$	$\mathcal{N}^{919}$	Linear regression with Laplace priors [176]	$\checkmark_{LC}$
prior_mix	$\mathcal{B} \times (\mathcal{N}_t + \mathcal{N}_t)$	$\mathcal{N}^{10}$	Model with mixture prior [61]	$\checkmark_{mix}$
wells_probit	$\mathcal{U}^2$	${\mathcal{B}_{pro}}^{500}$	Logistic model w. probit activation [50]	$\checkmark_{LC}$
timeseries	$\mathcal{U}^3$	$\mathcal{N}^{99}$	Timeseries model [50]	$\checkmark_{LC}$
unemployment	$\mathcal{U}^3$	$\mathcal{N}^{40}$	Linear Regression [50]	$\checkmark_{PC}$

Baselines

For certifying bounds on posterior distributions (without data perturbation), we compare AURA with two baselines: GuBPI [26], the start-of-the art tool for obtaining sound bounds on single posteriors and PSI [20], which leverages symbolic analysis to determine the exact posterior. For both baselines we use their most recent versions. For GuBPI we report the most precise result and their computation times, based on a grid search across all configurable GuBPI parameters[1]. We run GuBPI and AURA with the same number of splits.

For the study of adversarial data perturbation, these two baselines cannot be used. GuBPI's implementation cannot support interval specifications for data, and GuBPI also suffers numerical instability on scalar inputs (Section 5.10.1). PSI's implementation cannot solve the integrals for most benchmarks and data sizes considered in this work. We instead implemented pure interval analysis within AURA, akin to GuBPI's interval abstraction, but we carefully enhanced numerical stability[2].

Precision Metrics

We define the lower p_l and upper p_u bound functions for marginal posterior of the parameter x: $p_l(x) = l$ and $p_u(x) = u$ where $[l, u] = [\![P]\!]^\sharp_n(x^\sharp_i, d^\sharp)$ for $x \in \gamma(x^\sharp_i)$, given AURA results. We define two precision metrics (more details in Appendix B.2):

• *Total Variation Distance (TVD):* TVD measures the discrepancy between the lower and upper bounds of a distribution. It is calculated as $\text{TVD}_x = \frac{1}{2}\int |p_l(x) - p_u(x)|\,dx$. When a program has multiple parameters, TVD is averaged across all the parameters: $\text{TVD} = \frac{1}{M}\sum_{j=1}^{M} \text{TVD}_{x_j}$.

• *Absolute Difference on Parameter Means (ADM):* ADM measures the maximum absolute difference in expected values between any distribution within the bounds and the true distribution. Namely, $ADM_x = \max_{p' \in \mathcal{P}} |\mathbb{E}_{p'}(x) - \mathbb{E}_{p_{truth}}(x)|$, where $\mathcal{P} = \{p' : \forall x, p_l(x) \leq p'(x) \leq p_u(x)\}$ and $\mathbb{E}_{p_{truth}}(x)$ is obtained from Stan's NUTS sampling. We average ADM across all parameters. The TVD and ADM for bounds obtained from GuBPI and the interval analysis are analogous.

[1] The parameters include method ("boxes","linear"), scoring precision (0.001-0.1), variable precision (0.01-1), the depth of the symbolic exploration (10-1K) and splits in the "boxes" method (200-800K). We omit the configurations under which GuBPI implementation is unsound due to disconnected bounding boxes on continuous curves (Appendix B.4 presents an example).

[2] Evaluating this interval analysis version on the benchmarks in Section 5.10.1 achieves much higher precision than GuBPI, but is still much less precise compared to AURA.

Setup for Adversarial Data Perturbation Analysis

We also use AURA to find bounds on posteriors obtainable after data perturbations (see Section 5.5). Perturbations involve 1-5 key data points per benchmark, identified by gradient magnitude $\frac{\partial}{\partial d}[\![P]\!](x, d)$. For datasets with <100 points, perturbations are capped at five points or 5% of the dataset. Perturbation intervals are computed by modifying the original data by $[0, 0.01\sigma]$ if $sign(\frac{\partial}{\partial d}[\![P]\!](x, d))$ is positive or $[-0.01\sigma, 0]$ if the sign is negative, where σ is the dataset's standard deviation. We adapt the adversarial data perturbation method from the Fast Gradient Signed Method (FGSM) [177], a prevalent method in machine learning, which is to add a small perturbation towards the direction increasing the loss (cf. decreasing likelihood). We focus on benchmarks with continuous data distributions. Both AURA and the baseline are enhanced with GPU acceleration and equal-area splits for efficiency. We use Total Variation Distance (TVD) as the metric.

Experimental Setup

We run AURA and all the other tools on a AMD 4.2 GHz machine with 32 cores with NVIDIA RTX A5000 GPUs. For a fair comparison with GuBPI and PSI which only utilize a single core, we present the AURA timing on a single core as well.

5.10 EVALUATION

5.10.1 AURA Precision and Execution Time for Bounding a Single Posterior

Table 5.2 presents the results of AURA and two other baseline tools, GuBPI and PSI. We run AURA on both a single core CPU and a GPU, and all the other tools on a single core CPU. Columns 2-5 present the *precision* of the bounds of AURA and GuBPI with Total Variation Distance (**TVD**) and Absolute Difference on Parameter Means (**ADM**). Columns 6-9 (**Time (s)**) show the run times. The last row (**Geomean**) displays the geometric mean for benchmarks applicable to the specific tool.

Precision of Bounds AURA obtains highly precise bounds for all benchmarks on average (geomean) 0.03 in TVD and 0.14 in ADM (lower is better). AURA outperforms GuBPI in precision across all benchmarks: only 3 GuBPI instances have lower error than AURA's worst error in either metric. These results highlight the difference between AURA's gradient-based method and GuBPI's interval abstraction for programs with non-trivial number of data points (10-1000). PSI is guaranteed to produce exact results, but it is only able to

Table 5.2: AURA Execution Time and Precision of AURA compared to baselines (using 200 quantization splits). We run AURA and other tools on a single core CPU. For PSI, we denote timeout (>90 min) as "t.o.", unevaluated integrals as "inte". We compute the geometric mean of AURA's speedup over the baselines (as $\times^*$) only on the benchmarks that work for the baseline (on the single-core CPU).

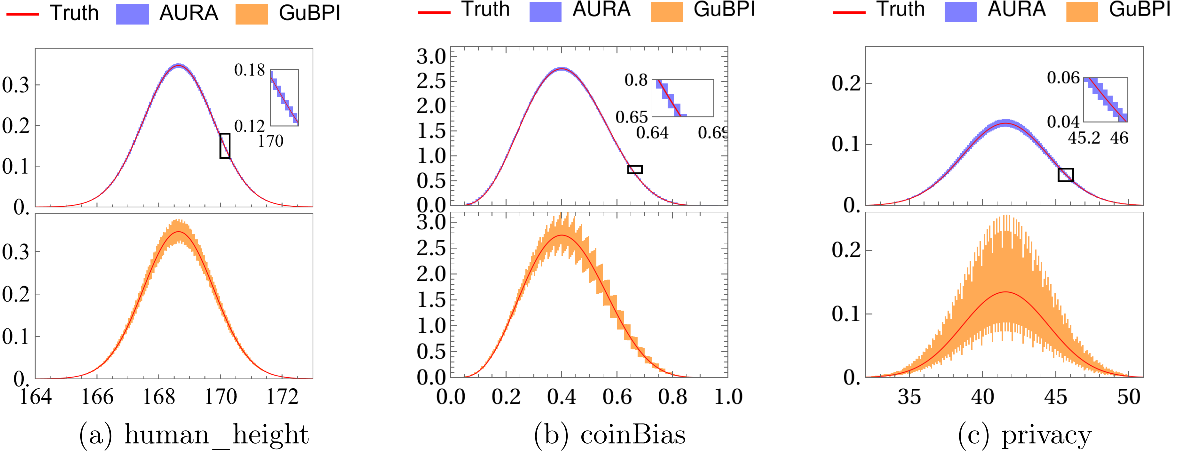
Program	**TVD**		**ADM**		**Time (s)**			
	AURA	GuBPI	AURA	GuBPI	AURA (GPU)	AURA (CPU)	GuBPI	PSI
exponential	0.02	-	0.05	-	0.04	0.01	-	0.02
gamma	0.02	-	0.03	-	0.10	0.03	-	0.03
gaussian	0.02	0.02	0.03	0.04	0.03	0.01	0.10 (9.8×)	0.02
coinBias	0.01	0.06	0.01	0.06	0.04	0.01	170.93 ($1.3{\times}10^4\times$)	0.15
human_height	0.02	0.04	5.14	13.97	0.06	0.04	1.33 (29.7×)	0.13
clinicalTrial	0.02	∞	0.02	∞	0.10	0.03	-	1.53
altermu2	0.11	16.33	0.20	88.01	0.04	0.20	132.84 (668.0×)	12.17
personality	0.02	-	0.00	-	0.04	0.05	-	-
reg_logistic	0.02	-	0.04	-	2.47	0.94	-	-
privacy	0.03	0.38	2.57	31.40	0.26	0.23	18.69 (81.0×)	inte
logistic	0.03	-	0.09	-	0.13	2.77	-	t.o.
lightspeed	0.03	$5.0{\times}10^5$	1.28	$2.7{\times}10^7$	0.07	0.76	93.40 (122.6×)	inte
anova_radon_n	0.03	$1.1{\times}10^8$	0.06	$1.2{\times}10^8$	1.66	1.55	93.98 (60.5×)	inte
IQStan	0.04	$3.3{\times}10^5$	5.43	$7.0{\times}10^7$	3.85	183.48	71.17 (0.4×)	inte
reg_laplace	0.04	-	0.08	-	4.05	5.02	-	t.o.
prior_mix	0.04	$1.0{\times}10^6$	1.10	$2.5{\times}10^6$	0.16	0.07	12.40 (178.6×)	inte
wells_probit	0.06	-	0.05	-	0.36	29.67	-	-
timeseries	0.13	∞	0.21	∞	33.83	873.96	-	inte
unemployment	0.16	∞	0.33	∞	28.90	833.43	-	inte
Geomean	0.03	267.15	0.14	3334.35	0.32	0.66	19.56 ($77.9\times^*$)	0.17 ($6.6\times^*$)

compute results for seven simple benchmarks (all of them with under 40 data points and simple posterior distribution expressions).

Execution Time Using a GPU, AURA solved 13 out of 19 benchmarks within 1 second, and all the benchmarks within one minute. On a single core CPU, AURA solves 12 out of 19 benchmarks within 1s, and is faster on 10 benchmarks than AURA on GPU. This speed difference is because these programs are too small to take advantage of the GPU: the GPU version of AURA is 25-50x faster than the CPU version for the four largest programs.

On the programs GuBPI can solve, AURA is much faster than GuBPI, with geometric mean 125.7× on GPU, and 77.9× on the single core CPU.

Analysis Examples Figure 5.14 presents the posterior density bounds derived by AURA and GuBPI for three benchmarks. The x-axis shows the parameter values; the y-axis shows the posterior probabilistic density. We show the ground truth with a red line and bounded



(a) human_height (b) coinBias (c) privacy

Figure 5.14: Visualization of AURA and GuBPI Bounds

area derived from AURA and GuBPI with blue and orange rectangles, respectively. The ground truth is derived manually and symbolically with the aid of Mathematica's numerical integration. From the plots, AURA is significantly more precise than GuBPI. AURA's bounds closely follow the ground truth line (see the enlarged area). This precision is mainly because AURA uses optimization to find the tightest abstractions. AURA also improves numerical stability by using the log density, avoiding the over/underflow issues that cause GuBPI to report extremely large numbers (TVD $> 10^5$) in five benchmarks. Unlike GuBPI, which uniformly splits intervals, AURA creates nearly equal-area bounding boxes, adjusting splits based on density function shape (details in Section 5.8.3).

5.10.2 AURA Precision for Bounding a set of Posteriors under Data Perturbation

Precision of Bounds We use AURA to find bounds for a set of posteriors when subjected to data perturbation, as outlined in Section 5.9. Table 5.3 presents the precision (in TVD) alongside the run time for both AURA and a baseline interval analysis. For a fair comparison of our optimization-based and interval abstractions, we run both AURA and the interval analysis using a GPU. On average (geo-mean), AURA achieves a precision 12.9$\times$ better than that of the interval analysis. We observed across all benchmarks that the input perturbation causing the maximum posterior error (TVD) almost never occurs at the extremes of the input perturbation interval, indicating that the sound bounds cannot be simulated just from data values at perturbation extremes.

Execution Time AURA's run time averages at 3.14s (geometric mean). The increase in run time when compared to AURA analysing a single posterior without perturbation is due

Table 5.3: AURA and Interval Analysis Results for Data Perturbation

Program	TVD		Time (s)	
	AURA	Interval	AURA (GPU)	Interval (GPU)
human_height	0.04	0.09 (2.3×)	0.25	0.02
reg_logistic	0.05	8.13 (175.8×)	3.49	1.39
lightspeed	0.07	0.56 (8.2×)	0.19	0.04
anova_radon_n	0.07	0.57 (8.2×)	1.52	0.05
reg_laplace	0.07	6.28 (87.6×)	10.44	1.18
prior_mix	0.07	0.23 (3.2×)	0.27	0.03
IQStan	0.07	0.20 (2.7×)	15.67	0.14
timeseries	0.18	1.04 (5.7×)	335.16	5.37
unemployment	0.23	4.10 (18.1×)	131.76	0.82
altermu2	0.28	16.43 (58.2×)	0.19	0.06

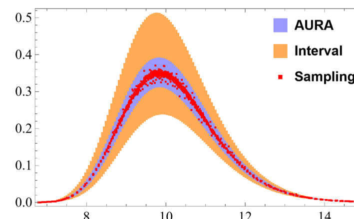


Figure 5.15: lightspeed (param: σ)

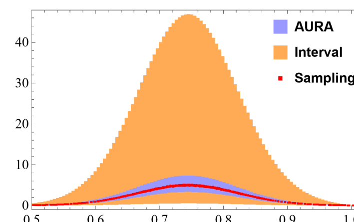


Figure 5.16: unemployment (param:β_1)

to the additional complexity of high-dimensional optimization across both the parameters and the data dimensions subjected to perturbation. Compared to the interval analysis, AURA has an additional cost of iteratively evaluating the unnormalized posterior during gradient ascent.

Analysis Examples Figures 5.15 and 5.16 illustrate the bounds computed by AURA for two example models under data perturbation, compared with the results from the interval analysis and reference distributions generated via Stan sampling. To generate the reference distributions, we simulated at least three concrete perturbations for each perturbed data point and used Stan's NUTS to collect 400,000 samples for each concrete perturbation. The plots show the parameter value on the x-axis against the posterior probability density on the y-axis, with red dots representing the sampled reference distributions and blue and orange rectangles representing the bounds computed by AURA and interval analysis, respectively. Consistent with the findings reported in Table 5.3, AURA shows significantly tighter bounds. This precision stems from AURA's optimization-based abstraction, which is designed to find the narrowest bounds before normalization.

5.10.3 Queries under Data Perturbation

We demonstrate the use of AURA in evaluating the posterior probability of specific events when the input dataset is subject to perturbations. Intuitively, one provides a query, as described in Def. 5.8, and AURA then computes sound bounds on the posterior probability

of the event defined by the query, where the probability bounds hold for *any* posterior that could result from the data perturbation. Table 5.4 shows the example query results on the two programs shown in Figure 5.15 and 5.16. The columns "AURA" and "Interval" show the bounds on the query probability determined by AURA and the interval analysis, respectively. The "Imp." column shows how much smaller AURA's bounds are compared to those of interval analysis. Across all the programs feasible for data perturbation (with the full table in Appendix B.3), AURA's bounds on the queries are much more precise, being on average (geometric mean) 5.35x narrower than the bounds from interval arithmetic. The time for each query is close to the time for computing the posterior bounds under perturbation.

Table 5.4: Examples of Queries (Full Table in Appendix B.3)

lightspeed			
Query	AURA	Interval	Improve.
$20 \leq \beta_0 < 40$	$[0.88, 1.00]$	$[0.47, 1.00]$	4.4×
$\beta_0 < 30 \wedge \sigma < 10$	$[0.42, 0.55]$	$[0.21, 1.00]$	6.1×
unemployment			
Query	AURA	Interval	Improve.
$\sigma < 1.2 \wedge \beta_0 < 3 \wedge \beta_1 < 0.7$	$[0.19, 0.41]$	$[0.03, 1.00]$	4.5×
$0.95 < \beta_1 < 1$	$[0.00, 0.01]$	$[0.00, 0.04]$	10.4×

5.10.4 Scaling to Larger Datasets

Figure 5.17 presents the impact of data size on AURA results. We increase the data size for all 15 applicable programs from 200 to 5000 (simulated from the same distribution as the original data), without applying data perturbation. We run AURA on a GPU with 200 splits. The left y-axis shows AURA's execution time in blue; the right y-axis shows precision in orange, both representing the geometric mean across benchmarks. AURA maintains nearly constant precision across varying data sizes, while GuBPI fails to scale and gives imprecise bounds (TVD $> 10^5$) already at around 10 data points. Also, AURA's time increases linearly with data size, and the number of data points AURA can compute with depends primarily on the size of the GPU memory. Beyond 2000 data points, the memory of a single A5000 GPU is insufficient for a few benchmarks, and thus we distributed the computation across two GPUs for data sizes ranging from 2000-5000 for all benchmarks. Spliting to two GPUs introduced a small shift in the execution time around 2000 data points, but execution time increase remained consistently linear both before and after this split, which also demonstrates AURA's scalability to leverage multiple GPUs. In all cases the impact on the precision is minimal.

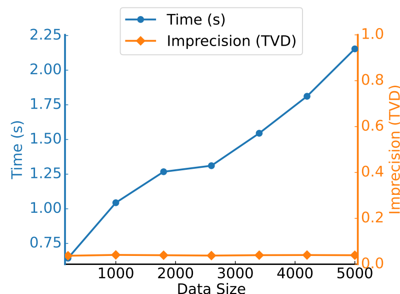


Figure 5.17: Varying #Data

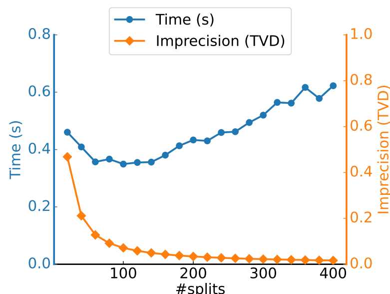


Figure 5.18: Varying #Splits

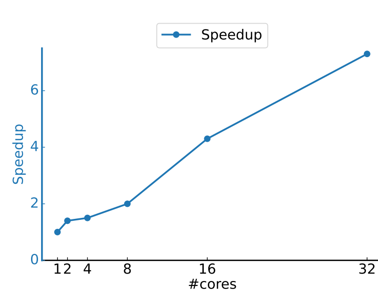


Figure 5.19: Varying #CPUs

5.10.5 Ablation Studies

Varying Quantization Splits Figure 5.18 presents the geometric mean precision and performance trade-off, as a function of the number of quantization splits (#splits). AURA's error decreases exponentially, while the computation time initially decreases before starting to increase. More splits help the optimization converge in fewer steps since the interval regions that gradient ascent explores are smaller, albeit at the cost of increased computation per step from having more intervals. For our benchmarks, 200 splits gives a reasonable balance between run time and precision.

Scaling to Multiple CPUs Fig. 5.19 shows the average speedup when running AURA on different number of cores using original data. The speedup of our PyTorch-based implementation (with no additional performance-enhancing optimizations) is approximately a linear function of the number of cores. AURA is the first tool for PP inference with guaranteed bounds to run in parallel.

Numerical data formats For each benchmark, we calculated the posterior bounds using both FP32 and FP64 precision. The overall numerical discrepancy between FP32 and FP64 was generally less than 10^{-6}. Specifically, the geometric mean of differences in the posterior bounds across all intervals and benchmarks was $4.7 \cdot 10^{-7}$. The geomean execution time overhead when using FP64 compared to FP32 on a GPU is only 1.1x. Thus, our analysis confirms that using FP32 precision gives accurate outcomes across all benchmarks.

5.11 RELATED WORK

Exact Probabilistic Programming Systems Despite recent progress, exact inference systems are limited for continuous distributions: e.g., many support only discrete models (DICE [21]), only handle Sum-Product networks [22] or cannot solve many complicated integrals (PSI [20], Hakaru [19]). AQUA [178] implements quantized inference, but its result

is not sound. In contrast, AURA computes bounds on the posterior distribution that are provably sound.

Interval-Based Abstractions for Probabilistic Inference Closest in spirit to our work is GuBPI [26] which computes interval bounds on a *single* posterior distribution. However, GuBPI is less precise than AURA and cannot scale to the datasizes AURA supports. Additionally, unlike AURA, GuBPI does not consider the data perturbation setting. While Cousot and Monerau [23] studied probabilistic abstract interpretation, their language did not consider Bayesian inference and lacks `observe` statements, a limitation also present in other work [13, 25] that focused on computing interval bounds of query probabilities.

Probability boxes (p-boxes) [156], while designed for bounding sets of cumulative density functions (CDFs), are not viable for probabilistic programming due to inherent limitations in computing directly on CDFs. Specifically, there is no method to combine CDFs or probability masses to derive a posterior, which is crucial in probabilistic programming and Bayesian inference that fundamentally requires computing the product of prior and likelihood PDFs. Similarly, Dempster-Shafer structures [156], which assign probability to intervals (akin to assigning probability "masses" to discrete outcomes), lack a mechanism to compute the posterior by combining these probabilities.

Dataset Perturbations of Bayesian Inference A key motivation underlying the need for provable abstractions of probabilistic inference is to obtain formal guarantees on inference results when the Bayesian model is perturbed. Prior work has shown that Bayesian inference is particularly sensitive to small perturbations in the observed data [31, 154, 155, 179, 180, 181]. Despite this need, to the best of our knowledge, the only existing probabilistic programming work that studies questions about data perturbations is PSense [31]. However, since PSense is built on top of PSI, it consequently struggles with intractable integrals for most continuous programs and cannot analyze those from our experiments. Further, PSense is limited to considering only one data point perturbation at a time. In contrast, AURA can find sound bounds when multiple data perturbations are present.

Optimization-Based Abstract Interpretation The idea of performing abstract interpretation of numerical computations by solving optimization problems has been studied before. One of the earliest works [182] showed how to reduce abstract interpretation to convex optimization in certain restricted cases. More recently, PRIMA [183] leverages optimization to abstractly interpret non-linear operations in the setting of DNN verification,

and Pasado [166] uses optimization to find precise abstract transformers for automatic differentiation primitives. However, AURA is the first to apply optimization-based abstract interpretation to probabilistic programs.

5.12 CONCLUSION

We presented AURA, an abstract interpretation of probabilistic programs that is also the first work to provide certified bounds on posterior distributions under data perturbations. By designing custom, precise and scalable abstract transformers for probabilistic programming using optimization, AURA represents a first step towards making provably robust probabilistic programming a reality. Further, by considering the data perturbation setting for the first time, AURA provides the first work on robustness of probabilistic programs to data attacks. We anticipate that AURA also opens the door to obtaining certified robustness for general probabilistic models (like Normalizing Flows and Probabilistic Circuits), and even abstracting other (non-probabilistic) pseudo-concave functions.

CHAPTER 6: CONCLUSIONS AND FUTURE WORK

6.1 CONCLUSIONS

Probabilistic programming has revolutionized Bayesian modeling by decoupling model development from inference. It simplified statistical analysis, thus making it widely adopted in applications like autonomous vehicle testing, pandemic prediction, and security modeling.

This dissertation has addressed the critical issues of trustworthiness in the domain of probabilistic programming, illustrating both its growing applications and inherent challenges. By taking a systematic approach that spans the entire probabilistic programming computation stack, this work not only identifies the trustworthiness issues but also offers innovative solutions to quantify and mitigate these issues. By introducing ASTRA and SixthSense, the dissertation provides essential tools for evaluating the robustness of probabilistic programming and debugging convergence issues.

Further enriching this field, the dissertation introduces a new quantized inference algorithm, AQUA, which allows more accurate and salable probabilistic inference compared to existing sampling-based and exact inference algorithms. We further expand probabilistic reasoning by introducing AURA, which provides a novel perspective by providing precise, guaranteed bounds on posterior distributions amidst data perturbations. Thus it paves the way for future research in probabilistic programming and its applications in ensuring its trustworthiness in addressing complex, real-world problems.

6.2 FUTURE WORK

6.2.1 Program Synthesis for Enhancing the Robustness of Probabilistic Programs

To improve the robustness of probabilistic programs, we propose a more concrete approach to synthesizing the robustness-improving transformations. Building on top of Chapter 2. where we evaluated various transformations by executing all of them, one can further address the challenge of selecting the most effective transformation without the necessity of trial runs. Therefore, we can integrate robustness analysis directly into a program synthesis framework specifically designed to optimize probabilistic programs for robustness. The analysis should evaluate potential transformations and combinations based on their metrics. Then, using these evaluations, the system could employ a decision-making algorithm to select and apply the most promising transformation. Moreover, a recent work AquaSense [184]

utilizes AQUA analysis for conducting sensitivity analysis of probabilistic programs. Unlike general robustness analyses, AquaSense uniquely concentrates on the impact of prior selections on program outcomes. Since there can be complicated interaction between prior and likelihood choices, the envisioned synthesis framework can be further enhanced by integrating insights from the sensitivity analysis.

6.2.2 Robustness of Probabilistic Programs through the lens of Adversarial Attacks

AURA is the first work that utilizes the FGSM (Fast Gradient Sign Method) attack, a concept borrowed from the machine learning domain, to test probabilistic programs. The study of model robustness through adversarial attacks is well-established within the machine learning community buy remains relatively under-explored for probabilistic programs. Inspired by AURA, there is potential to develop more complicated or potent adversarial attacks tailored for probabilistic programs. Such attacks could offer a method to examine robustness by observing the model's responds to various threat models. Moreover, the adversarial attack can also be integrated into the synthesis framework discussed earlier.

6.2.3 Optimization-based Bounds for Other Machine Learning Models

AURA opens up a new direction by employing optimization for bounding the posteriors of probabilistic programs. A key insight from this approach is the guarantee of optimization convergence when the posterior is pseudo-concave. This property of pseudo-concavity extends beyond just probabilistic programs. Many machine learning models rely on probabilistic choices and implement pseudo-concave functions. For example, applications that utilize probabilistic programs for computing complex loss functions, or Bayesian neural networks, which bear similarities to probabilistic program, could also benefit from this optimization method. Thus, the same AURA framework may help assess and enhance the robustness of a wider range of machine learning models.

6.2.4 Advanced Hardware Acceleration of Robustness Analyses

While deep neural networks have benefited a lot from hardware acceleration such as GPU utilization, probabilistic programs have not been extensively explored from a hardware acceleration perspective, despite their computationally intensive nature. AQUA and AURA demonstrates the feasibility of splitting the analysis on a large domain into smaller, manageable intervals and conducting these analyses in parallel on a GPU/CPU, showcasing a

promising path for more advanced optimizations. For instance, leveraging multiple GPUs could further enhance the performance of the robustness analyses; more efficient caching strategies could optimize the reuse of some computations; and reordering or regrouping optimizations within the model could also lead to more efficient memory access. Notably, the order of some statements in a probabilistic program can be altered (e.g. the prior declaration of independent parameters), due to its more declarative than imperative semantics. Additionally, compiler optimizations such as loop interchanging or conditional reordering could also be applied, opening up many new possibilities.

REFERENCES

[1] N. Goodman, V. Mansinghka, D. M. Roy, K. Bonawitz, and J. B. Tenenbaum, "Church: a language for generative models," *arXiv preprint arXiv:1206.3255*, 2012.

[2] D. J. Fremont, T. Dreossi, S. Ghosh, X. Yue, A. L. Sangiovanni-Vincentelli, and S. A. Seshia, "Scenic: a language for scenario specification and scene generation," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2019, pp. 63–78.

[3] H. Bherwani, S. Anjum, S. Kumar, S. Gautam, A. Gupta, H. Kumbhare, A. Anshul, and R. Kumar, "Understanding covid-19 transmission through bayesian probabilistic modeling and gis-based voronoi approach: a policy perspective," *Environment, Development and Sustainability*, vol. 23, no. 4, pp. 5846–5864, 2021.

[4] I. Sweet, J. M. C. Trilla, C. Scherrer, M. Hicks, and S. Magill, "What's the over/under? probabilistic bounds on information leakage," in *International Conference on Principles of Security and Trust*, ser. POST. Springer, Cham, 2018.

[5] M. Kučera, P. Tsankov, T. Gehr, M. Guarnieri, and M. Vechev, "Synthesis of probabilistic privacy enforcement," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 391–408.

[6] R. M. Neal, "Mcmc using hamiltonian dynamics," *Handbook of markov chain monte carlo*, vol. 2, no. 11, p. 2, 2011.

[7] A. V. Nori, C.-K. Hur, S. K. Rajamani, and S. Samuel, "R2: An efficient mcmc sampler for probabilistic programs," in *AAAI*, 2014.

[8] B. Carpenter, A. Gelman, M. Hoffman, D. Lee, B. Goodrich, M. Betancourt, M. A. Brubaker, J. Guo, P. Li, and A. Riddell, "Stan: A probabilistic programming language," *JSTATSOFT*, vol. 20, no. 2, 2016.

[9] M. D. Hoffman and A. Gelman, "The no-u-turn sampler: adaptively setting path lengths in hamiltonian monte carlo." *Journal of Machine Learning Research*, vol. 15, no. 1, pp. 1593–1623, 2014.

[10] M. Betancourt and M. Girolami, "Hamiltonian monte carlo for hierarchical models," *Current trends in Bayesian methodology with applications*, vol. 79, p. 30, 2015.

[11] M. I. Jordan, Z. Ghahramani, T. S. Jaakkola, and L. K. Saul, "An introduction to variational methods for graphical models," *Machine learning*, vol. 37, no. 2, pp. 183–233, 1999.

[12] D. M. Blei, A. Kucukelbir, and J. D. McAuliffe, "Variational inference: A review for statisticians," *Journal of the American statistical Association*, vol. 112, no. 518, pp. 859–877, 2017.

[13] A. Albarghouthi, L. D'Antoni, S. Drews, and A. Nori, "Fairsquare: probabilistic verification of program fairness," *Proceedings of the ACM on Programming Languages*, vol. 1, no. OOPSLA, p. 80, 2017.

[14] O. Bastani, X. Zhang, and A. Solar-Lezama, "Probabilistic verification of fairness properties via concentration," *Proceedings of the ACM on Programming Languages*, vol. 3, no. OOPSLA, pp. 1–27, 2019.

[15] N. Foster, D. Kozen, K. Mamouras, M. Reitblatt, and A. Silva, "Probabilistic netkat," in *Programming Languages and Systems: 25th European Symposium on Programming, ESOP 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2–8, 2016, Proceedings 25*. Springer, 2016, pp. 282–309.

[16] T. Gehr, S. Misailovic, P. Tsankov, L. Vanbever, P. Wiesmann, and M. Vechev, "Bayonet: probabilistic inference for networks," in *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 2018, pp. 586–602.

[17] H. Darir, G. E. Dullerud, and N. Borisov, "Probflow: Using probabilistic programming in anonymous communication networks." in *NDSS*, 2023.

[18] P. Mardziel, S. Magill, M. Hicks, and M. Srivatsa, "Dynamic enforcement of knowledge-based security policies using probabilistic abstract interpretation." *JCS*, vol. 21, no. 4, pp. 463–532, 2013.

[19] P. Narayanan, J. Carette, W. Romano, C.-c. Shan, and R. Zinkov, *Probabilistic Inference by Program Transformation in Hakaru (System Description)*, ser. FLOPS, 2016.

[20] T. Gehr, S. Misailovic, and M. Vechev, "PSI: Exact symbolic inference for probabilistic programs," in *International Conference on Computer Aided Verification*. Springer, 2016, pp. 62–83.

[21] S. Holtzen, G. Van den Broeck, and T. Millstein, "Scaling exact inference for discrete probabilistic programs," *Proceedings of the ACM on Programming Languages*, vol. 4, no. OOPSLA, pp. 1–31, 2020.

[22] F. A. Saad, M. C. Rinard, and V. K. Mansinghka, "SPPL: a probabilistic programming system with exact and scalable symbolic inference," *PLDI*, 2021.

[23] P. Cousot and M. Monerau, "Probabilistic abstract interpretation," in *Programming Languages and Systems*. Springer, 2012, pp. 169–193.

[24] D. Wang, J. Hoffmann, and T. Reps, "Pmaf: an algebraic framework for static analysis of probabilistic programs," in *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 2018, pp. 513–528.

[25] S. Sankaranarayanan, A. Chakarov, and S. Gulwani, "Static analysis for probabilistic programs: inferring whole program properties from finitely many paths," *ACM SIGPLAN Notices*, vol. 48, no. 6, pp. 447–458, 2013.

[26] R. Beutner, C.-H. L. Ong, and F. Zaiser, "Guaranteed bounds for posterior inference in universal probabilistic programming," in *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, 2022, pp. 536–551.

[27] M. Boreale and L. Collodi, "Guaranteed inference for probabilistic programs: A parallelisable, small-step operational approach," in *International Conference on Verification, Model Checking, and Abstract Interpretation*. Springer, 2023, pp. 141–162.

[28] G. Barthe, B. Grégoire, J. Hsu, and P.-Y. Strub, "Coupling proofs are probabilistic product programs," *ACM SIGPLAN Notices*, vol. 52, no. 1, pp. 161–174, 2017.

[29] M. Gorinova, D. Moore, and M. Hoffman, "Automatic reparameterisation of probabilistic programs," in *International Conference on Machine Learning*. PMLR, 2020, pp. 3648–3657.

[30] M. I. Gorinova, A. D. Gordon, and C. Sutton, "Probabilistic programming with densities in slicstan: efficient, flexible, and deterministic," *Proceedings of the ACM on Programming Languages*, vol. 3, no. POPL, pp. 1–30, 2019.

[31] Z. Huang, Z. Wang, and S. Misailovic, "Psense: Automatic sensitivity analysis for probabilistic programs," in *16th International Symposium on Automated Technology for Verification and Analysis*, ser. ATVA, 2018.

[32] G. Barthe, T. Espitau, B. Grégoire, J. Hsu, and P.-Y. Strub, "Proving expected sensitivity of probabilistic programs," vol. 2, no. POPL, 2017.

[33] S. Dutta, O. Legunsen, Z. Huang, and S. Misailovic, "Testing probabilistic programming systems," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 2018, pp. 574–586.

[34] S. Dutta, W. Zhang, Z. Huang, and S. 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*. ACM, 2019, pp. 729–739.

[35] E. Bingham, J. P. Chen, M. Jankowiak, F. Obermeyer, N. Pradhan, T. Karaletsos, R. Singh, P. Szerlip, P. Horsfall, and N. D. Goodman, "Pyro: Deep Universal Probabilistic Programming," *Journal of Machine Learning Research*, 2018.

[36] C. Robert and G. Casella, *Monte Carlo statistical methods.* New York: Springer Science & Business Media, 2013.

[37] M. J. Beal, *Variational algorithms for approximate Bayesian inference.* University of London, 2003.

[38] S. J. Taylor and B. Letham, "Forecasting at scale," *The American Statistician*, vol. 72, no. 1, pp. 37–45, 2018.

[39] J. Ai, N. S. Arora, N. Dong, B. Gokkaya, T. Jiang, A. Kubendran, A. Kumar, M. Tingley, and N. Torabi, "Hackppl: a universal probabilistic programming language," in *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, 2019, pp. 20–28.

[40] B. Gokkaya, J. Ai, M. Tingley, Y. Zhang, N. Dong, T. Jiang, A. Kubendran, and A. Kumar, "Bayesian neural networks using hackppl with application to user location state prediction," 2018.

[41] A. D. Gordon, T. A. Henzinger, A. V. Nori, and S. K. Rajamani, "Probabilistic programming," in *FoSE*, 2014.

[42] P. Huber and E. Ronchetti, *Robust statistics.* Wiley, New York, 1981.

[43] Y. Wang, A. Kucukelbir, and D. M. Blei, "Robust probabilistic modeling with bayesian data reweighting," in *Proceedings of the 34th International Conference on Machine Learning - Volume 70*, ser. ICML'17. JMLR.org, 2017. [Online]. Available: http://dl.acm.org/citation.cfm?id=3305890.3306058 pp. 3646–3655.

[44] C. Wang, D. M. Blei et al., "A general method for robust bayesian modeling," *Bayesian Analysis*, vol. 13, no. 4, pp. 1159–1187, 2018.

[45] "Reparameterization," https://mc-stan.org/docs/stan-users-guide/reparameterization.html, 2018, http://mc-stan.org/users/documentation/index.html.

[46] J. O. Berger, E. Moreno, L. R. Pericchi, M. J. Bayarri, J. M. Bernardo, J. A. Cano, J. De la Horra, J. Martín, D. Ríos-Insúa, B. Betrò et al., "An overview of robust bayesian analysis," *Test*, vol. 3, no. 1, pp. 5–124, 1994.

[47] A. Gelman, H. S. Stern, J. B. Carlin, D. B. Dunson, A. Vehtari, and D. B. Rubin, *Bayesian data analysis.* New York: Chapman and Hall/CRC, 2013.

[48] F. A. Saad, M. F. Cusumano-Towner, U. Schaechtle, M. C. Rinard, and V. K. Mansinghka, "Bayesian synthesis of probabilistic programs for automatic data modeling," *Proceedings of the ACM on Programming Languages*, vol. 3, no. POPL, p. 37, 2019.

[49] F. E. Allen, "Control flow analysis," *ACM Sigplan Notices*, vol. 5, no. 7, 1970.

[50] "Stan Example Models," 2018, https://github.com/stan-dev/example-models.

[51] A. Kucukelbir, R. Ranganath, A. Gelman, and D. M. Blei, "Automatic variational inference in stan," in *NIPS*, 2015.

[52] H. Akaike, "A new look at the statistical model identification," *IEEE transactions on automatic control*, vol. 19, no. 6, pp. 716–723, 1974.

[53] T. Ando, *Bayesian model selection and statistical modeling.* CRC Press, 2010.

[54] S. Watanabe and M. Opper, "Asymptotic equivalence of bayes cross validation and widely applicable information criterion in singular learning theory." *Journal of machine learning research*, vol. 11, no. 12, 2010.

[55] M. Stone, "Cross-validatory choice and assessment of statistical predictions," *Journal of the royal statistical society: Series B (Methodological)*, vol. 36, no. 2, pp. 111–133, 1974.

[56] R. E. Kass and A. E. Raftery, "Bayes factors," *Journal of the american statistical association*, vol. 90, no. 430, pp. 773–795, 1995.

[57] R. McElreath, *Statistical rethinking: A Bayesian course with examples in R and Stan.* Chapman and Hall/CRC, 2020.

[58] "Label Switching in Mixture Models," https://mc-stan.org/docs/stan-users-guide/label-switching-problematic.html, 2018, http://mc-stan.org/users/documentation/index.html.

[59] Z. Huang, S. Dutta, and S. Misailovic, "Debugging convergence problems in probabilistic programs via program representation learning with sixthsense," *International Journal on Software Tools for Technology Transfer*, pp. 1–20, 2024.

[60] P.-C. Bürkner, J. Gabry, and A. Vehtari, "Approximate leave-future-out cross-validation for bayesian time series models," *Journal of Statistical Computation and Simulation*, vol. 90, no. 14, pp. 2499–2523, 2020.

[61] Z. Huang, S. Dutta, and S. Misailovic, "Aqua: Automated quantized inference for probabilistic programs," in *International Symposium on Automated Technology for Verification and Analysis.* Springer, 2021, pp. 229–246.

[62] S. Dutta, Z. Huang, and S. Misailovic, "Sixthsense: Debugging convergence problems in probabilistic programs via program representation learning," in *International Conference on Fundamental Approaches to Software Engineering.* Springer, Cham, 2022, pp. 123–144.

[63] F. Futami, I. Sato, and M. Sugiyama, "Variational inference based on robust divergences," *arXiv preprint arXiv:1710.06595*, 2017.

[64] I. L. T. Gbohounme, O. O. Ngesa, and J. Eggoh, "Self-selecting robust logistic regression model," *Int. J. Statist. Probab.*, vol. 6, no. 3, pp. 1–9, 2017.

[65] N. Carlini and D. Wagner, "Towards evaluating the robustness of neural networks," in *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2017, pp. 39–57.

[66] D. Gopinath, G. Katz, C. S. Pasareanu, and C. Barrett, "Deepsafe: A data-driven approach for checking adversarial robustness in neural networks," *arXiv preprint arXiv:1710.00486*, 2017.

[67] S. Gu and L. Rigazio, "Towards deep neural network architectures robust to adversarial examples," *arXiv preprint arXiv:1412.5068*, 2014.

[68] N. Papernot, P. McDaniel, X. Wu, S. Jha, and A. Swami, "Distillation as a defense to adversarial perturbations against deep neural networks," in *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2016, pp. 582–597.

[69] U. Shaham, Y. Yamada, and S. Negahban, "Understanding adversarial training: Increasing local stability of supervised models through robust optimization," *Neurocomputing*, vol. 307, pp. 195–204, 2018.

[70] T. Minka, J. Winn, J. Guiver, S. Webster, Y. Zaykov, B. Yangel, A. Spengler, and J. Bronskill, "Infer.NET 2.5," 2013, microsoft Research Cambridge. http://research.microsoft.com/infernet.

[71] N. K. Tehrani, N. S. Arora, D. Noursi, M. Tingley, N. Torabi, and E. Lippert, "Bean machine: A declarative probabilistic programming language for efficient programmable inference," in *PGM*, 2020.

[72] "Modeling censored time-to-event data using pyro," 2019, https://eng.uber.com/modeling-censored-time-to-event-data-using-pyro/.

[73] S. Flaxman, S. Mishra, A. Gandy, H. J. T. Unwin, T. A. Mellan, H. Coupland, C. Whittaker, H. Zhu, T. Berah, J. W. Eaton et al., "Estimating the effects of non-pharmaceutical interventions on covid-19 in europe," *Nature*, pp. 1–5, 2020.

[74] A. Gelman, "Stan being used to study and fight coronavirus," 2020, Stan Forums. [Online]. Available: https://discourse.mc-stan.org/t/stan-being-used-to-study-and-fight-coronavirus/14296

[75] F. Obermeyer, "Deep probabilistic programming with pyro," 2020, models, Inference, and Algorithms. [Online]. Available: https://www.broadinstitute.org/talks/deep-probabilistic-programming-pyro

[76] R. M. Neal, "An improved acceptance procedure for the hybrid monte carlo algorithm," *Journal of Computational Physics*, vol. 111, no. 1, pp. 194–203, 1994.

[77] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "code2vec: Learning distributed representations of code," *Proceedings of the ACM on Programming Languages*, vol. 3, no. POPL, p. 40, 2019.

[78] U. Alon, S. Brody, O. Levy, and E. Yahav, "code2seq: Generating sequences from structured representations of code," in *International Conference on Learning Representations*, 2019. [Online]. Available: https://openreview.net/forum?id=H1gKYo09tX

[79] V. Raychev, M. Vechev, and A. Krause, "Predicting program properties from big code," in *ACM SIGPLAN Notices*, vol. 50, no. 1. ACM, 2015, pp. 111–124.

[80] S. Iyer, I. Konstas, A. Cheung, and L. Zettlemoyer, "Summarizing source code using a neural attention model," in *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 2016, pp. 2073–2083.

[81] C. Mendis, A. Renda, S. Amarasinghe, and M. Carbin, "Ithemal: Accurate, portable and fast basic block throughput estimation using deep neural networks," in *ICML*, 2019.

[82] E. Bingham and H. Mannila, "Random projection in dimensionality reduction: applications to image and text data," in *Proceedings of the international conference on Knowledge discovery and data mining (KDD)*. ACM, 2001, pp. 245–250.

[83] A. Andoni and P. Indyk, "Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions," *Communications of the ACM*, vol. 51, no. 1, p. 117, 2008.

[84] "Stan. using target += syntax," 2016, https://stackoverflow.com/questions/40289457/stan-using-target-syntax.

[85] M. Datar, N. Immorlica, P. Indyk, and V. S. Mirrokni, "Locality-sensitive hashing scheme based on p-stable distributions," in *Proceedings of the twentieth annual symposium on Computational geometry*. ACM, 2004, pp. 253–262.

[86] "Prior choice recommendations in stan," 2011, https://github.com/stan-dev/stan/wiki/Prior-Choice-Recommendations.

[87] 2018, http://mc-stan.org/users/documentation/index.html.

[88] H. Raiffa and R. Schlaifer, "Applied statistical decision theory," 1961.

[89] R. Sakia, "The box-cox transformation technique: a review," *Journal of the Royal Statistical Society: Series D (The Statistician)*, vol. 41, no. 2, pp. 169–178, 1992.

[90] "Inference case studies in knitr," 2019, https://github.com/betanalpha/knitr_case_studies.

[91] A. Gelman, D. Lee, and J. Guo, "Stan a probabilistic programming language for bayesian inference and optimization," *Journal of Educational and Behavioral Statistics*, 2015.

[92] "Nearpy," 2011, https://github.com/pixelogik/NearPy.

[93] "Tree interpreter package," 2020, https://github.com/andosa/treeinterpreter.

[94] T. Fawcett, "An introduction to roc analysis," *Pattern recognition letters*, vol. 27, no. 8, pp. 861–874, 2006.

[95] J. Davis and M. Goadrich, "The relationship between precision-recall and roc curves," in *Proceedings of the 23rd international conference on Machine learning*. ACM, 2006, pp. 233–240.

[96] C. G. Northcutt, T. Wu, and I. L. Chuang, "Learning with confident examples: Rank pruning for robust classification with noisy labels," in *Proceedings of the Thirty-Third Conference on Uncertainty in Artificial Intelligence*, ser. UAI'17. Sydney, Australia: AUAI Press, 2017. [Online]. Available: http://auai.org/uai2017/proceedings/papers/35.pdf

[97] F. Wood, J. W. van de Meent, and V. Mansinghka, "A new approach to probabilistic programming inference," in *AISTATS*, 2014.

[98] V. Mansinghka, D. Selsam, and Y. Perov, "Venture: a higher-order probabilistic programming platform with programmable inference," *arXiv preprint 1404.0099*, 2014.

[99] N. D. Goodman and A. Stuhlmüller, "The design and implementation of probabilistic programming languages," 2014.

[100] D. Tran, A. Kucukelbir, A. B. Dieng, M. Rudolph, D. Liang, and D. M. Blei, "Edward: A library for probabilistic modeling, inference, and criticism," *arXiv*, 2016.

[101] "Pyro," 2018, http://pyro.ai.

[102] G. Claret, S. K. Rajamani, A. V. Nori, A. D. Gordon, and J. Borgström, "Bayesian inference using data flow analysis," in *FSE*, 2013.

[103] C. Nandi, D. Grossman, A. Sampson, T. Mytkowicz, and K. S. McKinley, "Debugging probabilistic programs," in *MAPL*. ACM, 2017, pp. 18–26.

[104] Z. Huang, S. Dutta, and S. Misailovic, "Astra: Understanding the practical impact of robustness for probabilistic programs," in *Uncertainty in Artificial Intelligence*. PMLR, 2023, pp. 900–910.

[105] S. Dutta, A. Shi, R. Choudhary, Z. Zhang, A. Jain, and S. Misailovic, "Detecting flaky tests in probabilistic and machine learning applications," in *ISSTA*, 2020.

[106] S. Dutta, A. Shi, and S. Misailovic, "Flex: fixing flaky tests in machine learning projects by updating assertion bounds," in *FSE*, 2021.

[107] S. Dutta, A. Arunachalam, and S. Misailovic, "To seed or not to seed? an empirical analysis of usage of seeds for testing in machine learning projects," in *ICST*, 2022.

[108] S. Dutta, J. Selvam, A. Jain, and S. Misailovic, "Tera: Optimizing stochastic regression tests in machine learning projects," in *ISSTA*, 2021.

[109] F. Long and M. Rinard, "Automatic patch generation by learning correct code," in *ACM SIGPLAN Notices*, vol. 51, no. 1. ACM, 2016, pp. 298–312.

[110] M. Allamanis, H. Peng, and C. Sutton, "A convolutional attention network for extreme summarization of source code," in *International Conference on Machine Learning*, 2016, pp. 2091–2100.

[111] K. Wang and Z. Su, "Learning blended, precise semantic program embeddings," *ArXiv, vol. abs/1907.02136*, 2019.

[112] M. Allamanis, H. Jackson-Flux, and M. Brockschmidt, "Self-supervised bug detection and repair," *Advances in Neural Information Processing Systems*, vol. 34, pp. 27 865–27 876, 2021.

[113] M. Pradel and K. Sen, "Deepbugs: A learning approach to name-based bug detection," *Proceedings of the ACM on Programming Languages*, vol. 2, no. OOPSLA, pp. 1–25, 2018.

[114] C. S. Xia and L. Zhang, "Less training, more repairing please: revisiting automated program repair via zero-shot learning," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022, pp. 959–971.

[115] P. Y. Simard, D. Steinkraus, and J. C. Platt, "Best practices for convolutional neural networks applied to visual document analysis." in *Icdar*, vol. 3, no. 2003, 2003.

[116] E. D. Cubuk, B. Zoph, D. Mane, V. Vasudevan, and Q. V. Le, "Autoaugment: Learning augmentation policies from data," *arXiv preprint arXiv:1805.09501*, 2018.

[117] L. Taylor and G. Nitschke, "Improving deep learning using generic data augmentation," *arXiv preprint arXiv:1708.06020*, 2017.

[118] K. Leyton-Brown, H. H. Hoos, F. Hutter, and L. Xu, "Understanding the empirical hardness of np-complete problems," *Communications of the ACM*, vol. 57, no. 5, pp. 98–107, 2014.

[119] E. B. Khalil, P. Le Bodic, L. Song, G. Nemhauser, and B. Dilkina, "Learning to branch in mixed integer programming," in *Thirtieth AAAI Conference on Artificial Intelligence*, 2016.

[120] M. Balunovic, P. Bielik, and M. Vechev, "Learning to solve smt formulas," in *Advances in Neural Information Processing Systems*, 2018, pp. 10 338–10 349.

[121] R. Istrate, F. Scheidegger, G. Mariani, D. Nikolopoulos, C. Bekas, and A. C. I. Malossi, "Tapas: Train-less accuracy predictor for architecture search," *arXiv preprint arXiv:1806.00250*, 2018.

[122] S. Dutta, G. Joshi, S. Ghosh, P. Dube, and P. Nagpurkar, "Slow and stale gradients can win the race: Error-runtime trade-offs in distributed sgd," *arXiv preprint arXiv:1803.01113*, 2018.

[123] B. Deng, J. Yan, and D. Lin, "Peephole: Predicting network performance before training," *arXiv preprint arXiv:1712.03351*, 2017.

[124] Y. Pu, K. Narasimhan, A. Solar-Lezama, and R. Barzilay, "sk_p: a neural program corrector for moocs," in *Companion Proceedings of the 2016 OOPSLA*. ACM, 2016, pp. 39–40.

[125] Y. Li, D. Choi, J. Chung, N. Kushman, J. Schrittwieser, R. Leblond, T. Eccles, J. Keeling, F. Gimeno, A. Dal Lago et al., "Competition-level code generation with alphacode," *Science*, vol. 378, no. 6624, pp. 1092–1097, 2022.

[126] "Chatgpt: Optimizing language models for dialogue," 2022, https://openai.com/blog/chatgpt.

[127] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. d. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman et al., "Evaluating large language models trained on code," *arXiv preprint arXiv:2107.03374*, 2021.

[128] A. Svyatkovskiy, S. K. Deng, S. Fu, and N. Sundaresan, "Intellicode compose: Code generation using transformer," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 1433–1443.

[129] R. Pardo, W. Rafnsson, C. W. Probst, and A. Wąsowski, "Privug: using probabilistic programming for quantifying leakage in privacy risk analysis," in *European Symposium on Research in Computer Security*. Springer, 2021, pp. 417–438.

[130] A. Aguirre, G. Barthe, J. Hsu, B. L. Kaminski, J.-P. Katoen, and C. Matheja, "A pre-expectation calculus for probabilistic sensitivity," ser. POPL, 2021.

[131] P. Bissiri, C. Holmes, and S. Walker, "A general framework for updating belief distributions," *Journal of the Royal Stat. Soc. Series B, Statistical Methodology*, vol. 78, no. 5, p. 1103, 2016.

[132] N. Goodman and J. Tenenbaum, "Probabilistic Models of Cognition," probmods.org.

[133] R. Nishihara, T. Minka, and D. Tarlow, "Detecting parameter symmetries in probabilistic models," *arXiv preprint arXiv:1312.5386*, 2013.

[134] R. M. Neal, *Bayesian learning for neural networks*. Toronto: Springer Science & Business Media, 2012.

[135] J. Laurel and S. Misailovic, "Continualization of probabilistic programs with correction," in *European Symposium on Programming*, ser. ESOP. Springer, 2020, pp. 366–393.

[136] W. R. Gilks, A. Thomas, and D. J. Spiegelhalter, "A language and program for complex bayesian modelling," *The Statistician*, 1994.

[137] A. Pfeffer, "Ibal: a probabilistic rational programming language," in *Proceedings of the 17th international joint conference on Artificial intelligence-Volume 1*. Morgan Kaufmann Publishers Inc., 2001, pp. 733–740.

[138] M. Borges, A. Filieri, M. d'Amorim, C. S. Păsăreanu, and W. Visser, "Compositional solution space quantification for probabilistic software analysis," ser. PLDI, 2014.

[139] Y. Luo, A. Filieri, and Y. Zhou, "Sympais: Symbolic parallel adaptive importance sampling for probabilistic program analysis," *arXiv preprint arXiv:2010.05050*, 2020.

[140] R. D. Shachter, B. D'Ambrosio, and B. Del Favero, "Symbolic probabilistic inference in belief networks." in *AAAI*, vol. 90, 1990, pp. 126–131.

[141] K.-C. Chang and R. Fung, "Symbolic probabilistic inference with both discrete and continuous variables," *Systems, Man and Cybernetics, IEEE Transactions on*, vol. 25, no. 6, pp. 910–916, 1995.

[142] S. Moral, R. Rumí, and A. Salmerón, "Mixtures of truncated exponentials in hybrid bayesian networks," in *Symbolic and Quantitative Approaches to Reasoning with Uncertainty*. Barcelona, Spain: Springer, 2001, pp. 156–167.

[143] P. P. Shenoy and J. C. West, "Inference in hybrid bayesian networks using mixtures of polynomials," *International Journal of Approximate Reasoning*, vol. 52, no. 5, 2011.

[144] S. Sanner and E. Abbasnejad, "Symbolic variable elimination for discrete and continuous graphical models." in *AAAI*, ser. AAAI'12. AAAI Press, 2012, pp. 1954–1960.

[145] M. D. Hoffman, D. M. Blei, C. Wang, and J. Paisley, "Stochastic variational inference," *The Journal of Machine Learning Research*, vol. 14, no. 1, pp. 1303–1347, 2013.

[146] N. Tehrani, N. S. Arora, Y. L. Li, K. D. Shah, D. Noursi, M. Tingley, N. Torabi, E. Lippert, E. Meijer et al., "Bean machine: A declarative probabilistic programming language for efficient programmable inference," in *International Conference on Probabilistic Graphical Models*. PMLR, 2020, pp. 485–496.

[147] J. Zhang and J. Xue, "Incremental precision-preserving symbolic inference for probabilistic programs," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2019, pp. 237–252.

[148] S. Smolka, P. Kumar, D. M. Kahn, N. Foster, J. Hsu, D. Kozen, and A. Silva, "Scalable verification of probabilistic networks," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2019, pp. 190–203.

[149] J. Wang, Y. Sun, H. Fu, K. Chatterjee, and A. K. Goharshady, "Quantitative analysis of assertion violations in probabilistic programs," in *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, 2021.

[150] M. Huang, H. Fu, K. Chatterjee, and A. K. Goharshady, "Modular verification for almost-sure termination of probabilistic programs," *Proceedings of the ACM on Programming Languages*, vol. 3, no. OOPSLA, 2019.

[151] P. Schröer, K. Batz, B. L. Kaminski, J.-P. Katoen, and C. Matheja, "A deductive verification infrastructure for probabilistic programs," *Proceedings of the ACM on Programming Languages*, vol. 7, no. OOPSLA2, pp. 2052–2082, 2023.

[152] T. Gehr, S. Steffen, and M. Vechev, "λpsi: Exact inference for higher-order probabilistic programs," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2020, pp. 883–897.

[153] P. Cousot and R. Cousot, "Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, ser. POPL '77. ACM, 1977, pp. 238–252.

[154] M. Gloeckler, M. Deistler, and J. H. Macke, "Adversarial robustness of amortized bayesian inference," in *Proceedings of the 40th International Conference on Machine Learning*, 2023, pp. 11 493–11 524.

[155] F. Altekrüger, P. Hagemann, and G. Steidl, "Conditional generative models are provably robust: Pointwise guarantees for bayesian inverse problems," *Transactions on Machine Learning Research*, 2023.

[156] S. Ferson, V. KREINOVICK, L. Ginzburg, and F. SENTZ, "Constructing probability boxes and dempster-shafer structures," Sandia National Lab.(SNL-NM), Albuquerque, NM (United States); Sandia, Tech. Rep., 2003.

[157] A. P. Meyer, A. Albarghouthi, and L. D'Antoni, "The dataset multiplicity problem: How unreliable data impacts predictions," in *Proceedings of the 2023 ACM Conference on Fairness, Accountability, and Transparency*, 2023, pp. 193–204.

[158] G. Baudart, J. Burroni, M. Hirzel, L. Mandel, and A. Shinnar, "Compiling stan to generative probabilistic languages and extension to deep probabilistic programming," in *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, 2021, pp. 497–510.

[159] J. Tassarotti and J.-B. Tristan, "Verified density compilation for a probabilistic programming language," *Proceedings of the ACM on Programming Languages*, vol. 7, no. PLDI, 2023.

[160] A. Miné et al., "Tutorial on static inference of numeric invariants by abstract interpretation," *Foundations and Trends® in Programming Languages*, vol. 4, no. 3-4, 2017.

[161] A. Majthay and A. Whinston, "Quasi-concave minimization subject to linear constraints," *Discrete Mathematics*, vol. 9, no. 1, pp. 35–59, 1974.

[162] J. B. Cruz and L. L. Pérez, "Convergence of a projected gradient method variant for quasiconvex objectives," *Nonlinear Analysis: Theory, Methods & Applications*, vol. 73, no. 9, pp. 2917–2922, 2010.

[163] J. E. Higgins and E. Polak, "Minimizing pseudoconvex functions on convex compact sets," *Journal of Optimization Theory and Applications*, vol. 65, no. 1, pp. 1–27, 1990.

[164] J. C. Dunn, "Global and asymptotic convergence rate estimates for a class of projected gradient processes," *SIAM Journal on Control and Optimization*, vol. 19, no. 3, pp. 368–400, 1981.

[165] E. Hazan, K. Levy, and S. Shalev-Shwartz, "Beyond convexity: Stochastic quasi-convex optimization," *Advances in neural information processing systems*, vol. 28, 2015.

[166] J. Laurel, S. B. Qian, G. Singh, and S. Misailovic, "Synthesizing precise static analyzers for automatic differentiation," *Proceedings of the ACM on Programming Languages*, vol. 7, no. OOPSLA2, 2023.

[167] S. Wang, H. Zhang, K. Xu, X. Lin, S. Jana, C.-J. Hsieh, and J. Z. Kolter, "Beta-crown: Efficient bound propagation with per-neuron split constraints for neural network robustness verification," *Advances in Neural Information Processing Systems*, vol. 34, pp. 29 909–29 921, 2021.

[168] A. Miné, "Relational abstract domains for the detection of floating-point run-time errors," in *European Symposium on Programming*. Springer, 2004, pp. 3–17.

[169] "XMP Library," 2016, https://github.com/NVlabs/xmp/tree/master.

[170] M. Hladík, L. V. Kolev, and I. Skalna, "Linear interval parametric approach to testing pseudoconvexity," *Journal of Global Optimization*, vol. 79, pp. 351–368, 2021.

[171] D. Oberski, "Mixture models: Latent profile and latent class analysis," *Modern statistical methods for HCI*, pp. 275–287, 2016.

[172] P. F. Thall, M. Ursino, V. Baudouin, C. Alberti, and S. Zohar, "Bayesian treatment comparison using parametric mixture priors computed from elicited histograms," *Statistical methods in medical research*, vol. 28, no. 2, pp. 404–418, 2019.

[173] D. W. Heck, I. Thielmann, M. Moshagen, and B. E. Hilbig, "Who lies? a large-scale reanalysis linking basic personality traits to unethical decision making," *Judgment and Decision making*, vol. 13, no. 4, pp. 356–371, 2018.

[174] T. Salimans, A. Karpathy, X. Chen, and D. P. Kingma, "Pixelcnn++: Improving the pixelcnn with discretized logistic mixture likelihood and other modifications," *arXiv preprint arXiv:1701.05517*, 2017.

[175] J. Laurel, R. Yang, A. Sehgal, S. Ugare, and S. Misailovic, "Statheros: Compiler for efficient low-precision probabilistic programming," in *2021 58th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2021, pp. 787–792.

[176] P. M. Williams, "Bayesian regularization and pruning using a laplace prior," *Neural computation*, vol. 7, no. 1, pp. 117–143, 1995.

[177] I. J. Goodfellow, J. Shlens, and C. Szegedy, "Explaining and harnessing adversarial examples," *arXiv preprint arXiv:1412.6572*, 2014.

[178] Z. Huang, S. Dutta, and S. Misailovic, "Automated quantized inference for probabilistic programs with aqua," *Innovations in Systems and Software Engineering*, vol. 18, no. 3, pp. 369–384, 2022.

[179] A. C. Blaas, "On the adversarial robustness of bayesian machine learning models," Ph.D. dissertation, University of Oxford, 2021.

[180] M. Wicker, L. Laurenti, A. Patane, Z. Chen, Z. Zhang, and M. Kwiatkowska, "Bayesian inference with certifiable adversarial robustness," in *International Conference on Artificial Intelligence and Statistics*. PMLR, 2021.

[181] Y. Feng, T. G. Rudner, N. Tsilivis, and J. Kempe, "Attacking bayes: Are bayesian neural networks inherently robust?" in *Fifth Symposium on Advances in Approximate Bayesian Inference*, 2023.

[182] T. M. Gawlitza, H. Seidl, A. Adjé, S. Gaubert, and É. Goubault, "Abstract interpretation meets convex optimization," *Journal of Symbolic Computation*, vol. 47, no. 12, pp. 1416–1446, 2012.

[183] M. N. Müller, G. Makarchuk, G. Singh, M. Püschel, and M. Vechev, "Prima: General and precise neural network certification via scalable convex hull approximations," *Proc. ACM Program. Lang.*, vol. 6, no. POPL, jan 2022. [Online]. Available: https://doi.org/10.1145/3498704

[184] Z. Zhou, Z. Huang, and S. Misailovic, "Aquasense: Automated sensitivity analysis of probabilistic programs via quantized inference," in *International Symposium on Automated Technology for Verification and Analysis*. Springer, 2023, pp. 288–301.

[185] M. Bagnoli and T. Bergstrom, "Log-concave probability and its applications," in *Rationality and Equilibrium: A Symposium in Honor of Marcel K. Richter*. Springer, 2006, pp. 217–241.

[186] G. O. Roberts and J. S. Rosenthal, "General state space markov chains and mcmc algorithms," *Probability surveys*, vol. 1, pp. 20–71, 2004.

[187] “Wikipedia: Mean absolute error,” 2023, https://en.wikipedia.org/wiki/Mean_absolute_error.

[188] “GuBPI – An Analyzer for Probabilistic Programs to Compute Guaranteed Bounds on the Posterior,” 2022, https://github.com/gubpi-tool/gubpi.

[189] “PSI Solver,” 2019, https://github.com/eth-sri/psi/tree/e729dd7d68e23a4a75731b4bb800c95111a7a30b.

APPENDIX A: ASTRA ADDITIONAL RESULTS

A.1 BEST MSE IMPROVEMENT FOR DIFFERENT NOISE MODELS

Tables A.1,A.2 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.

Table A.1: 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 A.2: 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)

A.2 CONVERGENCE SCORES AT NOISE LEVELS 2 AND 6

Tables A.3 and A.4 present the convergence scores at noise levels 2 and 6. We observed a similar overall trend in convergence scores across different noise levels.

Table A.3: (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 A.4: (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	–

APPENDIX B: AURA PROOF AND EXPERIMENT DETAILS

B.1 PSEUDO-CONCAVITY OF BENCHMARKS

In this section, we prove the pseudo-concavity of all our benchmarks. Namely, for each benchmark P, the function $[\![P]\!](x, d)$ is pseudo-concave with respect to their parameters and data. Users of AURA can apply the same conclusions or adopt the general proof strategy if their programs are similar to ours.

Our benchmarks are categorized into four distinct classes:

1. Benchmarks named "exponential", "gamma", and "gaussian" correspond to individual distributions. Their pseudo-concavity is given in Lemma B.1, with details in Table B.1.

2. Benchmarks such as "lightspeed", "anova_radon_n", "IQStan", and "unemployment" have a linear regression model structure. The pseudo-concavity of these benchmarks is established by Theorem B.1.

3. Benchmarks including "personality", "reg_logistic", "privacy", "logistic", "reg_laplace", "altermu2","wells_probit", and "timeseries" are log-concave. Their log-concavity is established in Theorem B.2, result from their composition of individual log-concave functions, which is rigorously proven using structural induction.

4. Benchmarks such as "human_height", "clinicalTrial", and "prior_mix" are composed of mixture models. The unnormalized posterior of these models is a summation of pseudo-concave functions. The pseudo-concavity of each component is given by one of the first three points.

Table B.1 shows the details of log-concavity (LC) and pseudo-concavity (PC) of each distribution with respect to their parameters or data.

We first outline several essential lemmas that underpin the subsequent proof and formulation of the theorems:

Lemma B.1. (Log-Concavity and Pseudoconcavity of the Individual Distributions in Table B.1) The Log-Concavity and Pseudoconcavity properties of the individual distributions (normal, uniform, beta, bernoulli, bernoulli-logit, bernoulli-probit, laplace, logistic, gamma, exponential) shown in Table B.1 are well-known facts, as summarised in [185].

Lemma B.2. (Product of Log-Concave Functions) Let $f_i(x_i)$ be a set of functions where each f_i is log-concave, then $g(x_1, ..., x_n) = \prod_i f_i(x_i)$ is LC w.r.t $x_1, \ldots, x_N$.

Table B.1: Log-Concavity and Pseudo-Concavity of Individual Distributions and Likelihoods. $\mathscr{P}$ denotes the power set. For example, $p_{\text{normal}}(x, \mu, \sigma)$ is LC w.r.t μ and x when both of them are variables, or w.r.t μ when x being constant, or w.r.t x with μ being constant.

Distribution	**Log-Concavity (LC) w.r.t**	**Pseudo-Concavity (PC) w.r.t**
$\prod_i p_{\text{normal}}(x_i \vert \mu, \sigma)$	$\mathscr{P}(\{x_1, ..., x_N, \mu\})$	$\mathscr{P}(\{x_1, ..., x_N, \mu, \sigma\})$
$p_{\text{normal}}(x \vert \mu, \sigma)$	$\mathscr{P}(\{x, \mu\})$	$\mathscr{P}(\{x, \mu, \sigma\})$
$p_{\text{uniform}}(x \vert a, b)$	x	x
$p_{\text{beta}}(x \vert \alpha, \beta)$	x if $\alpha \geq 1 \wedge \beta \geq 1$	x if $\alpha \geq 1 \vee \beta \geq 1$
$p_{\text{bernoulli}}(x \vert p)$	p	p
$p_{\text{bernoulli-logit}}(x \vert \theta)$	θ	θ
$p_{\text{bernoulli-probit}}(x \vert \theta)$	θ	θ
$p_{\text{laplace}}(x \vert \mu, b)$	$\mathscr{P}(\{x, \mu\})$	$\mathscr{P}(\{x, \mu, b\})$
$p_{\text{logistic}}(x \vert \mu, s)$	$\mathscr{P}(\{x, \mu\})$	$\mathscr{P}(\{x, \mu, s\})$
$p_{\text{gamma}}(x \vert k, \theta)$	x if $k \geq 1$	x for any k
$p_{\text{exponential}}(x \vert \lambda)$	x	x

Lemma B.3. (Product of Pseudoconcave Functions) Let $f_i(x_i)$ be a set of functions where each f_i is log-concave, then $g(x_1, ..., x_n) = \prod_i f_i(x_i)$ is also pseudoconcave w.r.t $x_1, \ldots, x_N$.

Lemma B.4. (Composition of a Log-Concave function with a linear function) If $f(x) : \mathbb{R}^m \to \mathbb{R}$ is Log-Concave and $A \in \mathbb{R}^{m \times n}$, then $g(y) : \mathbb{R}^n \to \mathbb{R}$ defined by $f(A(y))$ is Log-Concave.

Lemma B.5. (Composition of a Quasiconcave function with a linear function) If $f(x) : \mathbb{R}^m \to \mathbb{R}$ is Quasiconcave and $A \in \mathbb{R}^{m \times n}$, then $g(y) : \mathbb{R}^n \to \mathbb{R}$ defined by $f(A(y))$ is Quasiconcave.

Lemma B.6. (Composition of a Pseudoconvex function with a monotonic function) Let $f(x) : \mathbb{R}^m \to \mathbb{R}$ be Pseudoconvex, then for any non-decreasing function $g : \mathbb{R} \to \mathbb{R}$, then $g(f(x))$ is Pseudoconvex. Similarly is f is pseudoconcave, then $g(f(x))$ is pseudoconcave

Lemma B.7. Let $f(x) : \mathbb{R}^m \to \mathbb{R}$ be Quasiconvex, then if $\nabla f \neq 0$, then f is Pseudoconvex. Likewise if f is quasiconcave and $\nabla f \neq 0$, then f is Pseudoconcave.

Lemma B.8. Let $\alpha : \mathbb{R}^2 \to \mathbb{R}$ be defined as $\alpha(\sigma, c) = sqrt(-2\sigma^2(ln(\sigma^n(\sqrt{2\pi})^n)))$. α is concave with respect to σ for $\sigma > 0$, $n \in \mathbb{N}_+$, and $0 < c < \frac{1}{(\sqrt{2\pi\sigma})^n}$.

Proof. The second derivative of $\alpha(\sigma, c)$ is

$$\alpha''(\sigma, c) = \frac{-n\sigma^2(n - 2ln(c \cdot (\sqrt{2\pi}\sigma)^n))}{2\sqrt{2}(-\sigma^2 \cdot ln(c \cdot (\sqrt{2\pi}\sigma)^n))^{\frac{3}{2}}} \tag{B.1}$$

But the numerator is strictly negative while the denominator is strictly positive, hence

$$\alpha''(\sigma, c) < 0 \tag{B.2}$$

QED.

Lemma B.9. If $f(x) : \mathbb{R}^m \to \mathbb{R}$ has convex upper contour sets for all levels, then f is quasiconcave.

Lemma B.10. (Quasiconcavity of Composed Multi-variant Normal Likelihood) For simplicity, denote

$$f_{normal}(\mu, \sigma, y_1, ..., y_n) = \prod_i p_{\text{normal}}(y_i|\mu, \sigma) = \prod_{i=1}^{n} \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\frac{(y_i-\mu)^2}{\sigma^2}} \tag{B.3}$$

Then $f_{normal}(\mu, \sigma, y_1, ..., y_n)$ is Quasiconcave on $[l_\sigma, u_\sigma] \times [l_{y_1}, u_{y_1}] \times ...[l_{y_n}, u_{y_n}]$ where $l_\sigma > 0$.

Proof. We first define

$$h(\sigma, y_1, ..., y_n) = \prod_{i=1}^{n} \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\frac{(y_i)^2}{\sigma^2}} \tag{B.4}$$

and we prove it is quasiconcave. We algebraically convert the product of the gaussian pdfs (one for each observed data point) into a single exponential function:

$$\prod_{i=1}^{n} \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\frac{(y_i)^2}{\sigma^2}} = \frac{1}{\sigma^n(\sqrt{2\pi})^n} e^{-\frac{1}{2}} \frac{\sum_i(y_i^2)}{\sigma^2} \tag{B.5}$$

Define the upper contour set at level c as

$$S_c = \{(\sigma, y_1, ..., y_n) | \frac{1}{\sigma^n(\sqrt{2\pi})^n} e^{-\frac{1}{2}} \frac{\sum_i(y_i^2)}{\sigma^2} \geq c\} \tag{B.6}$$

We do the following algebraic rearrangements:

$$\frac{1}{\sigma^n(\sqrt{2\pi})^n} e^{-\frac{1}{2}} \frac{\sum_i(y_i^2)}{\sigma^2} \geq c \tag{B.7}$$

$$\Rightarrow e^{-\frac{1}{2}} \frac{\sum_i(y_i^2)}{\sigma^2} \geq c \cdot \sigma^n(\sqrt{2\pi})^n \tag{B.8}$$

$$\Rightarrow -\frac{1}{2} \frac{\sum_i(y_i^2)}{\sigma^2} \geq ln(c\sigma^n(\sqrt{2\pi})^n) \tag{B.9}$$

$$\Rightarrow \sum_i (y_i^2) \leq -2\sigma^2(ln(c\sigma^n(\sqrt{2\pi})^n)) \tag{B.10}$$

$$\Rightarrow sqrt(\sum_i (y_i^2)) \leq sqrt(-2\sigma^2(ln(c\sigma^n(\sqrt{2\pi})^n))) \tag{B.11}$$

$$\Rightarrow \|y\|_2 \leq sqrt(-2\sigma^2(ln(c\sigma^n(\sqrt{2\pi})^n))) \tag{B.12}$$

Hence $S_c = \{(\sigma, y_1, ..., y_n) \ : \ \|y\|_2 \leq sqrt(-2\sigma^2(ln(c\sigma^n(\sqrt{2\pi})^n)))\}$.

We will actually use the following shorthand notation

$$\alpha(\sigma, c) = sqrt(-2\sigma^2(ln(c\sigma^n(\sqrt{2\pi})^n)))\} \tag{B.13}$$

Hence $S_c = \{(\sigma, y_1, ..., y_n) \ : \ \|y\|_2 \leq \alpha(\sigma, c)\}$.

We now prove the convexity of S_c for $0 < c < \frac{1}{(\sqrt{2\pi}\sigma)^n}$. If $c \leq 0$, then any $(\sigma, y_1, ..., y_n)$ satisfies the constraint since $\frac{1}{\sigma^n(\sqrt{2\pi})^n} e^{-\frac{1}{2}\frac{\sum_i (y_i^2)}{\sigma^2}} > 0$, and the set of *all* $(\sigma, y_1, ..., y_n)$ is a convex set. Likewise if $c \geq \frac{1}{(\sqrt{2\pi}\sigma)^n}$, then S_c is either a singleton or empty, both of which are convex sets.

Let $v, w \in S_c$, where by notational convenience $v = (v_1, ..., v_n, \sigma_1)$ and $w = (w_1, ..., w_n, \sigma_2)$. We will prove that their convex combination $\lambda v + (1-\lambda)w \in S_c$ for any $\lambda \in [0, 1]$.

Since $v \in S_c$ we know that $\|(v_1, ..., v_n)\|_2 \leq \alpha(\sigma_1, c)$, and furthermore $\lambda\|(v_1, ..., v_n)\|_2 \leq \lambda \cdot \alpha(\sigma_1, c)$. Similarly since $w \in S_c$, we know that $\|(w_1, ..., w_n)\|_2 \leq \alpha(\sigma_2, c)$ and likewise $(1-\lambda)\|(w_1, ..., w_n)\|_2 < (1-\lambda)\alpha(\sigma_2, c)$. Hence

$$\lambda\|(v_1, ..., v_n)\|_2 + (1-\lambda)\|(w_1, ..., w_n)\|_2 \leq \lambda\alpha(\sigma_1, c) + (1-\lambda)\alpha(\sigma_2, c) \tag{B.14}$$

Since $\lambda, (1-\lambda) \geq 0$

$$\|\lambda(v_1, ..., v_n)\|_2 + \|(1-\lambda)(w_1, ..., w_n)\|_2 \leq \lambda\alpha(\sigma_1, c) + (1-\lambda)\alpha(\sigma_2, c) \tag{B.15}$$

By triangle inequality (since $\|\cdot\|_2$ is a norm)

$$\|\lambda(v_1, ..., v_n) + (1-\lambda)(w_1, ..., w_n)\|_2 \leq \|\lambda(v_1, ..., v_n)\|_2 + \|(1-\lambda)(w_1, ..., w_n)\|_2 \tag{B.16}$$

Hence

$$\|\lambda(v_1, ..., v_n) + (1-\lambda)(w_1, ..., w_n)\|_2 \leq \lambda\alpha(\sigma_1, c) + (1-\lambda)\alpha(\sigma_2, c) \tag{B.17}$$

By the concavity of $\alpha(\sigma, c)$ (Lemma B.8) we know by Jensen's inequality that

$$\lambda\alpha(\sigma_1, c) + (1-\lambda)\alpha(\sigma_2, c) \leq \alpha(\lambda\sigma_1 + (1-\lambda)\sigma_2, c) \tag{B.18}$$

Hence

$$\|\lambda(v_1, ..., v_n) + (1-\lambda)(w_1, ..., w_n)\|_2 \leq \lambda\alpha(\sigma_1, c) + (1-\lambda)\alpha(\sigma_2, c) \leq \alpha(\lambda\sigma_1 + (1-\lambda)\sigma_2, c) \tag{B.19}$$

Or just

$$\|\lambda(v_1, ..., v_n) + (1-\lambda)(w_1, ..., w_n)\|_2 \leq \alpha(\lambda\sigma_1 + (1-\lambda)\sigma_2, c) \tag{B.20}$$

This implies the point $(\lambda v_1 + (1-\lambda)w_1, ..., \lambda v_n + (1-\lambda)w_n, \lambda\sigma_1 + (1-\lambda)\sigma_2)$ is in S_c, hence the upper contour sets S_c are convex, thus by Lemma B.9 we have quasiconcavity of h.

Furthermore, since f_{normal} is $h(A(x_1, ..., x_n, \mu, \sigma)$ where $A(x_1, ..., x_n, \mu, \sigma)$ is the linear function defined as

$$A(x_1, ..., x_n, \mu, \sigma) = (x_1 - \mu, ..., x_n - \mu, \sigma) \tag{B.21}$$

And since h is already proved to be quasiconcave and quasiconcave functions are closed under composition with linear functions (Lemma B.5), then f_{normal} is quasiconcave. QED.

Lemma B.11. $f_{normal}(\mu, \sigma, y_1, ..., y_n)$ is **Pseudoconcave**.

Proof. By Lemma B.10, we know that f_{normal} is at least quasiconcave, but since we have that $\frac{\partial}{\partial\sigma} f_{normal}(\mu, \sigma, y_1, ..., y_n) \neq 0$, then $\nabla f_{normal}(\mu, \sigma, y_1, ..., y_n)$ is never zero, thus by Lemma B.7 it is actually pseudoconcave (though not fully concave) QED.

Lemma B.12. $\log(f_{normal}(\mu, \sigma, y_1, ..., y_n))$ is **Pseudoconcave**.

Proof. Since log is a monotonic, non-decreasing function, the composition of log with a pseudoconcave function (such as f_{normal}) is still pseudoconcave by lemma B.6 QED.

Theorem B.1 (Pseudoconcavity of Linear Regression Programs)**.** The posterior distribution of Bayesian Linear Regression with the general pattern shows in Figure B.1 is Pseudoconcave.

Proof. All of our linear regression benchmarks have the code form of Fig. B.1 and thus have uniform priors over all parameters (slope, intercept, σ), hence the expression for $[\![P]\!](x, d)$ will be

$$[\![P]\!](x, d) = \frac{1}{u_m - l_m}\frac{1}{u_b - l_b}\frac{1}{u_s - l_s} f_{normal}(b, s, d_1 - mv_1, ..., d_n - mv_n) \tag{B.22}$$

where $x = (b, s)$ and $d = (d_1, ..., d_n)$. Since all v_i are constants, this is just the composition of a linear transformation with f_{normal} (which is already pseudoconcave) hence $[\![P]\!](x, d)$ is pseudoconcave and thus so is $[\![P]\!]_{\log}(x, d)$. QED.

```
1 m ~ uniform(l_m, u_m)
2 b ~ uniform(l_b, u_b)
3 s ~ uniform(l_s, u_s)
4 y1 ~ normal(m*v1+b, s)
5 ...
6 yn ~ normal(m*vn+b, s)
7 observe(y1, d1)
8 ...
9 observe(yn, dn)
```

Figure B.1: Linear Regression Code

Theorem B.2 (Log-Concavity in Branch-free Programs with Log-Concave Distributions)**.** For a program P written in our language (as shown in Figure 5.9), if P does not contain branching, and assuming that each individual distribution in the program is Log-Concave as classified in Table B.1, then P is also Log-Concave.

Proof. We prove Theorem B.2 using structural induction on the concrete semantics rules (Figure 5.11). The rules can be classified into three categories: statements, arithmetic expressions, and distribution expressions. The proof has three parts:

1. For arithmetic expressions $[\![E+E]\!]$, $[\![E-E]\!]$, $[\![cE]\!]$, $[\![\mathrm{x}_j]\!]$: if all operands are linear, and since these expressions are linear with respect to their operands, the resulting expression will also be linear. In structural induction, the base cases are a singleton parameter or a data point, both of which are directly linear functions w.r.t parameters or data themselves. Then, for each of these rules, by assuming their operand sub-expressions evaluate to linear functions, the $+$, $-$, and constant factor result in linear functions.

2. For the distribution expression $[\![\mathit{dist}(E_1, \ldots, E_N)]\!](x,d) = p_{\mathit{dist}}(u; [\![E_1]\!](x,d), \ldots, [\![E_N]\!]$ $(x,d))$: by the assumption of this lemma, p_{dist} is Log-Concave w.r.t its parameters/-data. Then by Lemma B.4, and the conclusion on arithmetic expressions that E_i must be linear functions, we have $[\![\mathit{dist}(E_1, \ldots, E_N)]\!](x,d)$ being Log-Concave.

3. For statements $[\![M;M]\!]$, $[\![D;D]\!]$, $[\![M;D]\!]$, $[\![\mathrm{observe}(\mathrm{Dist}, d_i)]\!](x,d)$ or $[\![\mathrm{x}_i \sim \mathrm{Dist}]\!](x)$, we prove they result in Log-Concave functions given that the sub-statements give Log-Concave functions. Again by structural induction:

    - The base cases are the single statements for likelihood or prior: $[\![\mathrm{observe}(\mathrm{Dist}, d_i)]\!]$ $(x,d) = [\![\mathrm{Dist}]\!](x,d) \circ d[i]$ or $[\![\mathrm{x}_i \sim \mathrm{Dist}]\!](x) = [\![\mathrm{Dist}]\!](x,d) \circ x[i]$. Both statements evaluate to the distribution expression $[\![\mathit{dist}(E_1, \ldots, E_N)]\!](x,d)$, which is Log-Concave as shown above.

- Then, the inductive step utilizes the two compositional properties of Log-Concave functions outlined in Lemma B.2. For sequencing statements $[\![M; M]\!]$, $[\![D; D]\!]$, and $[\![M; D]\!]$, if a preceding density is Log-Concave, and it is multiplied with a subsequent prior/likelihood function that is also Log-Concave (by inductive hypothesis), the resulting function remains Log-Concave (by Lemma B.2). Furthermore, for expressions (Lemma B.4).

This confirms that the Log-Concavity is preserved under all the statement rules in Figure 5.11, except for the branching rule. Therefore, $[\![P]\!](x, d)$ is Log-Concave. QED.

Lemma B.13 (Pseudo-Concavity of Log Unnormalized Posteriors)**.** Given a set of distributions where their corresponding prior or likelihood PDFs are either LC or PC, the log unnormalized posterior $[\![P]\!]_{\log}(x, d)$, which is the sum of log PDFs of these distributions, is pseudo-concave.

Proof. Let $[\![P]\!]_{\log}(x, d) = \sum_i \log f_i(x, d)$ where each $f_i(x, d)$ is the PDF of a distribution that is either LC or PC. By Lemma B.3, the sum $\sum_i \log f_i(x, d)$ retains the property of being LC or PC. Then $[\![P]\!]_{\log}(x, d)$ is pseudo-concave. QED.

B.2 EXPERIMENTAL SETUP DETAILS

B.2.1 Precision Metrics

We define two precision metrics for the analysis of program P. From the analysis, we first establish a lower bound function p_l and an upper bound function p_u for any value of the latent parameter x: For a fixed dataset:$p_l(x) = l$ and $p_u(x) = u$ are obtained from the analysis as $[l, u] = [\![P]\!]^\sharp_n(x^\sharp_i, d)$ for x within any $\gamma(x^\sharp_i)$. For perturbed datasets:$p_l(x) = l$ and $p_u(x) = u$ are defined as $[l, u] = [\![P]\!]^\sharp_n(x^\sharp_i, d^\sharp)$ for x within any $\gamma(x^\sharp_i)$.

Total Variation Distance (TVD). TVD [186] is a widely-used metric that intuitively measures the *area* between two distribution density functions. For two univariate probability density functions, p and q for a continuous random variable $x \in \mathbb{R}$, $\text{TVD}_{pq} = \frac{1}{2}\int |p(x) - q(x)|\, dx$. To measure the precision of the bounds on posterior distributions, we define the TVD for the bounds as:

$$\text{TVD}_x = \frac{1}{2}\int |p_l(x) - p_u(x)|\, dx. \tag{B.23}$$

The TVD between the lower and upper bounds also represents the maximum TVD for any two probability density functions confined within these bounds. If the program contains multiple parameters, the overall TVD is reported as: $\text{TVD} = \frac{1}{M}\sum_{j=1}^{M} \text{TVD}_{x_j}$, averaged across the parameters $x_1, x_2, \ldots, x_M$, where each TVD_{x_j} is computed based on the marginal density function of each x_j after computing the bounds on their joint density function.

Absolute Difference on Parameter Means (ADM). ADM [44, 187] measures the absolute difference between the expected values of parameters within two distributions. For two probabilistic density functions p and q on a random variable $x \in \mathbb{R}$, ADM is $ADM_{pq} = |\mathbb{E}_p(x) - \mathbb{E}_q(x)|$, where $\mathbb{E}_p(x) = \int x \cdot p(x)dx$ and similarly for q. We use ADM to quantify the precision of the bounds. Formally, given the bounds $p_l(x)$ and $p_u(x)$ on the posterior density function, we consider all the posterior density functions between these bounds, denoted as $\mathcal{P} = \{p' : \forall x. p_l(x) \leq p'(x) \leq p_u(x)\}$. The ADM then becomes the maximal absolute difference in the expectations between any function in $\mathcal{P}$ and the true expectation:

$$ADM_x = \max_{P \in \mathcal{P}} |\mathbb{E}_p(x) - \mathbb{E}_{p_{truth}}(x)|\,. \tag{B.24}$$

To get $\mathbb{E}_{p_{truth}}(x)$, we use Stan's NUTS sampling to obtain 400,000 samples and take the sample mean as the true mean. We report the program's $ADM = \frac{1}{M}\sum_{i}^{M} ADM_{x_j}$, averaged across all parameters.

The TVD and ADM for bounds obtained from GuBPI and the interval analysis are analogous.

B.2.2 Baseline Setup Details

We use the most recent version of GuBPI [188] and report the most precise solutions and their computation times, based on a grid search across all configurable GuBPI parameters and we run GuBPI with the same number of splits as AURA. The parameters include method ("boxes","linear"), scoring precision (0.001-0.1), variable precision (0.01-1), the depth of the symbolic exploration (10-1000) and splits in the "boxes" method (200-800000). We omit configurations under which GuBPI implementation is not sound due to disconnected bounding boxes on continuous curves (Appendix F presents an example). Since GuBPI does not work with infinite support, we use the precision enhancing splitting strategy to compute the same bounded interval for GuBPI and AURA. The time for this step is negligible (<0.01s) and is included in AURA's run time but not GuBPI's. We exclude the one-time cost to initialize the GPU from AURA's run time. For PSI, we use the most recent version [189] with default configurations. We run AURA's abstract interpretation until the gradient ascent converges, which we define as the point at which the density value is repeated within the machine epsilon.

B.3 QUERY UNDER DATA PERTURBATION FOR ALL FEASIBLE BENCHMARKS

We demonstrate the practical application of AURA in evaluating the posterior probability of specific events when the input dataset is subject to perturbations as described in Section 5.9. Intuitively, one provides a query, as described in Def. 5.8, and AURA then computes sound bounds on the posterior probability of the event defined by the query, where the probability bounds hold for *any* possible posterior that could result from the data perturbation.

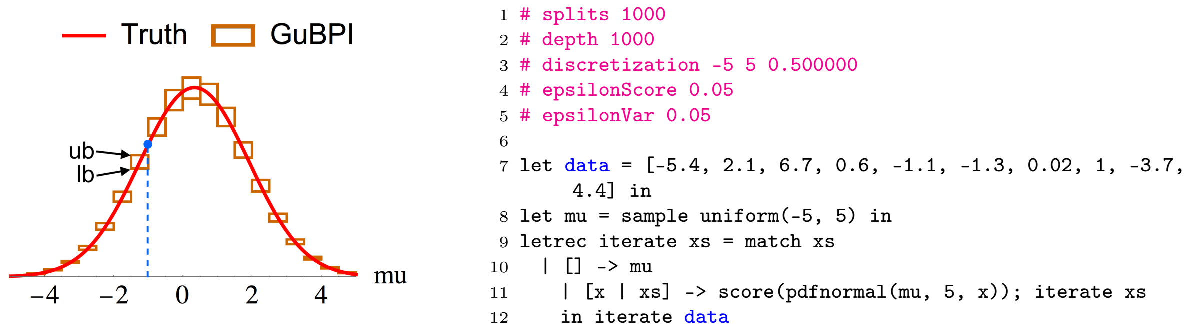
Table B.2 illustrates the results and the computation time AURA used to bound the posterior probability of each query. For each program amenable to data perturbation (i.e. those with continuous data), we formulated two distinct queries (shown in the **Query** column). The **Probability** columns show the bounds computed by AURA and the interval analysis we implemented as the baseline. The "Imp." column shows the improvement by AURA, reflected in how many times smaller AURA's interval is compared to the results from interval analysis. The **Time** column shows the time taken by AURA and the interval analysis. The results show that AURA's bounds are significanlty narrower than those of the simple interval analysis. Although AURA requires more time than the simple interval analysis, the absolute clock time to perform this computation is below 6 minutes for all benchmarks, which is acceptable for many applications.

Table B.2: Results for Queries under Data Perturbation ("|" means the same benchmark as in the previous row)

Program	**Query**	**Probability**			**Time (s)**	
		AURA	Interval	Imp.	AURA (GPU)	Interval (GPU)
human_height	$\mu > 165$	$[0.93, 1.00]$	$[0.85, 1.00]$	2.1×	0.233	0.012
|	$170 < \mu < 172$	$[0.11, 0.13]$	$[0.10, 0.14]$	1.7×	0.251	0.012
reg_logistic	$\beta_0 \geq 1.35 \lor \beta_0 < 1.15$	$[0.05, 0.07]$	$[0.00, 0.98]$	63.6×	3.499	1.373
|	$\beta_0 \geq 1.25$	$[0.33, 0.40]$	$[0.02, 1.00]$	14.0×	3.426	1.368
lightspeed	$20 \leq \beta_0 < 40$	$[0.88, 1.00]$	$[0.47, 1.00]$	4.4×	0.149	0.047
|	$\beta_0 < 30 \land \sigma < 10$	$[0.42, 0.55]$	$[0.21, 1.00]$	6.1×	0.155	0.040
anova_radon_n	$0.9 \leq a_0 \leq 1$	$[0.28, 0.34]$	$[0.14, 0.66]$	7.6×	2.310	0.053
|	$\sigma_y \leq 1 \lor a_0 > 1$	$[0.86, 1.00]$	$[0.46, 1.00]$	3.9×	1.974	0.049
reg_laplace	$1.3 \leq \beta_1 \leq 1.35$	$[0.27, 0.36]$	$[0.02, 1.00]$	11.5×	10.11	1.145
|	$\beta_0 \geq -0.5$	$[0.08, 0.11]$	$[0.01, 1.00]$	34.1×	10.20	1.149
prior_mix	$\mu_0 \leq 0$	$[0.74, 0.97]$	$[0.58, 1.00]$	1.8×	0.268	0.020
|	$\mu_0 > -2 \land \mu_0 < 2$	$[0.08, 0.11]$	$[0.06, 0.12]$	1.4×	0.276	0.017
IQStan	$\mu_1 \geq 85 \lor \mu_2 \geq 85$	$[0.87, 1.00]$	$[0.71, 1.00]$	2.2×	15.76	0.150
|	$5 < \sigma < 10 \land \mu_1 > 95 \land \mu_2 > 95$	$[0.25, 0.32]$	$[0.21, 0.38]$	2.3×	13.90	0.150
timeseries	$\alpha < -1 \lor \beta < 1 \land lag < 0.8$	$[0.69, 1.00]$	$[0.31, 1.00]$	2.3×	341.5	5.304
|	$\alpha > -0.5 \land lag > 0.5$	$[0.02, 0.04]$	$[0.01, 0.09]$	4.3×	338.2	5.252
unemployment	$\sigma < 1.2 \land \beta_0 < 3 \land \beta_1 < 0.7$	$[0.19, 0.41]$	$[0.03, 1.00]$	4.5×	129.2	0.827
|	$0.95 < \beta_1 < 1$	$[0.00, 0.01]$	$[0.00, 0.04]$	10.4×	130.6	0.843
altermu2	$1 \leq \mu_0 \leq 1.1$	$[0.02, 0.04]$	$[0.00, 0.84]$	34.0×	0.169	0.060
|	$\mu_0 < 1.5 \land \mu_1 < 1.5$	$[0.47, 1.00]$	$[0.02, 1.00]$	1.9×	0.184	0.052

B.4 ILLUSTRATION OF UNSOUND RESULTS FROM GUBPI

GuBPI, under certain configurations, is not sound. In our experience, our hyper-parameter sweeping will ignore those unsound results. Figure B.2 shows an example when GuBPI become unsound. For illustration purpose, we run GuBPI with 20 splits, while the same unsoundness would occur with any number of splits. On the plot, each orange box shows GuBPI's bounds on that interval, and the red line shows the ground truth. At several places the orange bounding boxes failed to cover the ground truth. For example, the point highlighted with a blue dot is (mu $= -1.02$) which is in the interval of (-1.5,-1) (i.e. the box to the left), but its ground truth is above the upper bound GuBPI derived for this interval.




```
1 # splits 1000
2 # depth 1000
3 # discretization -5 5 0.500000
4 # epsilonScore 0.05
5 # epsilonVar 0.05
6
7 let data = [-5.4, 2.1, 6.7, 0.6, -1.1, -1.3, 0.02, 1, -3.7,
      4.4] in
8 let mu = sample uniform(-5, 5) in
9 letrec iterate xs = match xs
10   | [] -> mu
11     | [x | xs] -> score(pdfnormal(mu, 5, x)); iterate xs
12     in iterate data
```

Figure B.2: GuBPI Unsound Result

Figure B.3: GuBPI Program which Gives Figure B.2