PSense: Automatic Sensitivity Analysis for Probabilistic Programs

Zixin Huang(✉), Zhenbang Wang, and Sasa Misailovic

University of Illinois at Urbana-Champaign, IL 61801, USA
{zixinh2,zw11,misailo}@illinois.edu

Abstract. PSense is a novel system for sensitivity analysis of probabilistic programs. It computes the impact that a noise in the values of the parameters of the prior distributions and the data have on the program's result. PSense relates the program executions with and without noise using a developer-provided *sensitivity metric*. PSense calculates the impact as a set of symbolic functions of each noise variable and supports various non-linear sensitivity metrics. Our evaluation on 66 programs from the literature and five common sensitivity metrics demonstrates the effectiveness of PSense.

1 Introduction

Probabilistic programing offers a promise of user-friendly and efficient probabilistic inference. Recently, researchers proposed various probabilistic languages and frameworks, e.g., [12, 11, 27, 19, 10, 24]. A typical probabilistic program has the following structure: a developer first specifies the initial assumptions about the random variables as *prior* distributions. Then the developer specifies the *model* by writing the code that relates these variables selecting those whose values have been observed. Finally, the developer specifies the query that asks how this evidence changes the distribution of some of the unobserved (latent) variables, i.e., their *posterior* distribution.

In many applications, both the choices of the prior parameters and the observed data points are *uncertain*, i.e., the used values may diverge from the true ones. Understanding the *sensitivity* of the posterior distributions to the perturbations of the input parameters is one of the key questions in probabilistic modeling. Mapping the sources of sensitivity can help the developer in debugging the probabilistic program and updating it to improve its robustness.

Sensitivity analysis has a rich history in engineering and statistics [15, 22] and has also been previously studied in the context of probabilistic models in machine learning [7, 25, 5, 17]. While useful, these techniques are typically sampling-based (providing only sensitivity estimates) or work for a limited subset of discrete models. However, sensitivity in probabilistic programming has not been studied extensively. Recently, Barthe et al. proposed a logic for reasoning about the expected sensitivity of probabilistic programs [4]. While sound, this approach requires a developer to prove properties using a proof assistant, supports only expectation distance and presets results for only a few examples. Like similar techniques for deterministic programs [8], its reliance on linearity of noise propagation may result in coarse over-approximations of non-linear operations.

Key Challenges. Key challenges for an effective probabilistic sensitivity analysis include (1) automation that aims to maintain both soundness and precision and (2) ability to work with non-linear programs and sensitivity metrics. Solving these challenges can help with understanding and improving robustness of probabilistic programs.

Our Work. PSense is a system for automatic sensitivity analysis of probabilistic programs. For each parameter in a probabilistic program, the analysis answers the question: *if the parameter/data value is changed by some value ε, how much does the posterior distribution change?* The analysis is fully symbolic and exact: it produces the distance expression that is valid for all legal values of ε. It uses a developer-specified sensitivity metric that quantifies the change in the posterior distributions between the programs with and without the noise. In this paper we present analysis with five classical metrics from statistics: two versions of expectation distance, Kolmogorov-Smirnov statistic, Total variation distance, and Kullback-Leibler divergence.

PSense can also answer sensitivity-related *optimization queries*. First, it can compute the numerical value of the maximum posterior distance given that ε is in some range. More interestingly, for a given acceptable threshold of difference between the posterior distributions, PSense can compute the maximum and minimum values of ε that satisfy the threshold.

PSense operates on imperative probabilistic programs with mixed discrete and continuous random variables, written in the PSI language [10]. PSI also comes with a powerful symbolic solver for exact probabilistic inference. One of the key insights behind PSense's design is that the sensitivity analysis can directly leverage PSI's inference. However, we also identified that PSI's analysis alone is not sufficient: (1) the expressions for distribution distance cannot be easily simplified by PSI's solver and (2) PSI does not support optimization queries. We therefore formulated these (non-linear and non-convex) queries and solved symbolically with Mathematica computer algebra system [2]. PSense workflow demonstrates the synergistic usage of symbolic solvers, guided by the domain-specific information. PSense is open-source software, available at `http://psense.info`.

In addition to the exact sensitivity analysis, PSense also supports an approximate analysis via a sampling-based backend. PSense translates the sensitivity analysis queries into WebPPL programs. WebPPL [13] is a probabilistic language with support for approximate MCMC inference. This way, PSense implements a common empirical approach for estimating sensitivity in probabilistic models.

Results. We evaluated PSense on a set of 66 probabilistic programs from the literature. We ran the sensitivity analysis for five metrics and 357 parameters per metric. Both the programs and the metrics are challenging: the programs have both discrete and continuous variables and many metrics are non-linear. The results show that (1) PSense, applied on all sensitivity metrics, successfully computed the exact sensitivity for the majority of analyzed parameters and data points, with a typical analysis being well under a minute; (2) PSense's optimization is also effective in computing the maximum noise that keeps the

posterior difference below an acceptable threshold; (3) PSense's exact symbolic analysis is often significantly more precise than the sampling-based approach. Jointly, these results demonstrate that symbolic analysis is a solid foundation for automatic and precise sensitivity analysis.

Contributions. The paper makes the following contributions:

- ⋆ **System for Automated Sensitivity:** To the best of our knowledge, PSense is the first automated system for exact symbolic analysis of sensitivity in probabilistic programs.
- ⋆ **Symbolic Analysis and Optimization:** We present PSense's global sensitivity analysis, which solves queries exactly, by building on the capabilities of PSI and Mathematica symbolic engines. We also present how to formulate and solve sensitivity-related optimization queries.
- ⋆ **Evaluation:** We evaluated PSense on 66 probabilistic programs from the literature, with a total of 357 parameter analyses. The experiments show the effectiveness and efficiency of PSense in analyzing sensitivity and solving optimization problems for various sensitivity metrics. We also show that PSense's symbolic analysis is often significantly more precise than sampling.

2 Examples

We demonstrate the capabilities of PSense through two representative examples. The first example shows the analysis of a simple discrete program. The second shows the analysis of stochastic gradient descent algorithm.

2.1 Sensitivity Analysis of Discrete Programs

Figure 1 presents a program that flips three coins. Each coin toss is a "head" (1) or a "tail" (0). The first three statements simulate tossing three independent coins. The variable `D` sums up the outcomes. While the value of `D` is not known, the developer includes the condition that at least two heads were observed (but not for which coins). We want to know the posterior probability that the coin toss `A` resulted in a "head", given this evidence.

```
def main(){
   A:=flip(0.5);
   B:=flip(0.5);
   C:=flip(0.5);
   D:=A+B+C;
   observe(D>=2);
   return A;
}
```

Fig. 1. Example

Problem Definition. The program has three constant parameters for the Bernoulli distributions assigned to `A`, `B`, and `C`. Different values of the parameters will give different posterior distributions. We are interested in the question: *what happens to the posterior distribution if we perturb the parameter of the prior distribution?*

To estimate the change in the output distribution, we can add noise to each of our prior `flip(0.5)`. In particular, PSense interprets the first statement as `A:=flip(0.5+eps)`, where the variable `eps` represents this noise. The noise may have any legal value, such that the flip probability is between 0.0 and 1.0.

Sensitivity Results. PSense first computes the posterior distribution of the variable `A`, which is a function of the noise variable `eps`. Then it compares it to the distribution of the program without noise using a sensitivity metrics. PSense can compute several built-in metrics of sensitivity, defined in Section 3.

For instance, the Expectation distance has been defined in [4] as the absolute difference between $\mathbb{E}[\texttt{main}_{eps}]$ and $\mathbb{E}[\texttt{main}]$ the expectations of the program's output distributions with and without noise: $D_{Exp} = \mid \mathbb{E}[\texttt{main}_{eps}] - \mathbb{E}[\texttt{main}] \mid$. After changing the parameter of the first `flip` statement, PSense produces the symbolic expression of this distance: `(3*Abs[eps])/(4*(1+eps))`. It also calculates the range of legal values for `eps`, which is $[-0.5, 0.5]$. PSense can successfully obtain the symbolic expressions for all other metrics and parameters.

Other Queries. PSense can perform several additional analyses:

- It can find the maximum value of the Expectation distance with respect to the noise `eps` within e.g., $\pm 10\%$ of the original parameter value. PSense formulates and solves an optimization problem, which in this case returns that the maximum value of the Expectation distance is approximately 0.0395, when `eps` is -0.05. One can similarly obtain the maximum absolute `eps` subject to the bound on the sensitivity metric.
- It analyzes whether the distance grows linearly as the noise `eps` increases. Interestingly, even for this a simple example, the Expectation distance is not linear, because `eps` appeared in the denominator. This is due to the rescaling of the posterior caused by the `observe` statement. In the version of the program without the `observe` statement, the Expectation distance is linear.

2.2 Sensitivity Analysis of Stochastic Gradient Descent

We now turn to a more complicated example, which implements a stochastic gradient descent (SGD) algorithm, in Figure 2. It is derived from the algorithm analyzed in [4], applied to the linear regression scenario (as in [1]).

```
def main(){
  x := [1.4,1.8,3.3,4.3,4.8,6.0,
        7.5,8.1,9.0,10.2];
  y := [2.2,4.0,6.1,8.6,10.2,12.4,
        15.1,15.8,18.4,20.0];
  w1 := 0;
  w2 := uniform(0, 1);
  a := 0.01;
  for t in [0..8){
    i := t;
    xi := x[i];
    yi := y[i];
    w1 = w1-a*2*(w1+w2*xi-yi);
    w2 = w2-a*2*(xi*(w1+w2*xi-yi));
  }
  return w2;
}
```

Fig. 2. Sample SGD Program

The variables x and y are two arrays that store the observed data. We fit a simple linear regression model $y_i = w_1 + w_2 x_i$. We first set the parameters w_1 and w_2 to some initial values. Then we use the gradient descent algorithm to adjust the parameters in order to minimize the error of the current fit. To make the model simpler, we set w_1 to a concrete initial value and assume w_2 follows the uniform distribution. We set the learning rate a to 0.01. In each iteration we adjust the value of w_1 and w_2 so that the square error in the prediction moves against the gradient and

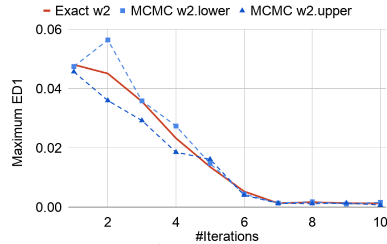

Fig. 3. Analysis of SGD

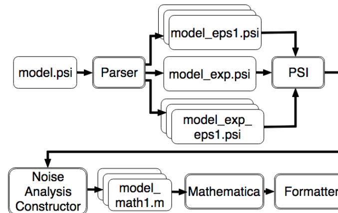

Fig. 4. PSense Workflow

towards the minimum. Finally we want to find how much does the choice of the initial value of w_2 affect the precision. Therefore, we return the distribution of w_2 after multiple iterations of the algorithm – in this experiment, between 1 and 10 (the iteration count must be fixed, which is a limitation of the underlying PSI solver). With more iterations, the value of w_2 approaches 2.0, as expected.

Find Sensitivity with PSense. We want to find out how the output distribution of *w2* changes if we perturb the parameters of the prior distribution. We add noise *eps* to each parameter in `uniform(0,1)`. PSense can output the results for different metrics, including expectation distance, KL divergence and total variation distance.

Figure 3 presents the change of the expectation distance subject to the fixed noise in the lower bound of the uniform distribution (w2.lower) and the upper bound of the uniform distribution (w2.upper) in the prior of $w2$. The Y-axis shows the maximum expectation distance after each iteration on the X-axis. The solid line is produced by the symbolic backend, while the dashed lines are generated by the sampling backend. The function for w2.lower and w2.upper for the symbolic backend are the same (and mark them as just w2). The results indicate that the noise in the prior has little effect on the output distribution after several iterations of the algorithm. The plot also illustrates the imprecision of the sampling backend: the computed sensitivities significantly differ in several iterations.

3 PSense System

Figure 4 presents the overview of the PSense workflow. PSense accepts programs written in the PSI language (Section 3.1). A developer also provides the sensitivity metrics that characterize parameter sensitivities (Section 3.2).

PSense supports two backends: Symbolic and Sampling. The symbolic backend first leverages PSI's analysis to get the distribution expression for the modified program. Then, it compiles the results from the programs with and without noise, and computes the sensitivity metric and other queries. PSense builds its query solver on top of Mathematica (Section 3.3). The sampling backend translates the programs into WebPPL, and instructs it to run the programs with and without noise for a specific number of times (Section 3.4).

As its output, PSense creates a table that presents the sensitivity of the program result to the change of each parameter/observation. The sensitivity is either a symbolic expression for symbolic backend, or an estimate and its variance for the sampling backend.

$n \in \mathbb{Z}$ $\quad$ $bop \in \{+, -, *, /, \hat{}\}$ $\quad$ $lop \in \{\&\&, ||\}$ $\quad$ $cop \in \{==, \neq, <, >, \leq, \geq\}$
$r \in \mathbb{R}$ $\quad$ Dist $\in \{$`Bernoulli`, `Gaussian`, `Uniform`, ...$\}$
$x \in$ Var $\quad$ $p \in$ Prog $\rightarrow$ Func$^+$
$a \in$ ArrVar $\quad$ $f \in$ Func $\rightarrow$ `def` $Id(Var^*)$ $\{$Stmt; `return` Var$^*\}$

se $\in$ Expr $\rightarrow$ n | r | x | $?x$ | a[Expr] | Dist(Expr$^+$) | f(Expr*) |
Expr *bop* Expr | Expr *cop* Expr | Expr *lop* Expr

$s \in$ Stmt $\rightarrow$ $x :=$ Expr | $a :=$ `array`(Expr) | $x =$ Expr | a[Expr] = Expr |
`observe` Expr | `assert` Expr | `skip` | Stmt; Stmt |
`if` Expr {Stmt} `else` {Stmt} | `for` x `in` [Expr..Expr) {Stmt}

Fig. 5. PSI Language Syntax [10]

3.1 Language

Figure 5 presents the syntax of PSI programs. Overall, it is a simple imperative language with scalar and array variables, conditionals, bounded for-loops (each loop can be unrolled as a sequence of conditional statements) and function calls. The language supports various discrete and continuous distributions. To sample from a distribution, a user assigns the distribution expression to a variable. The `observe` statement conditions on the expressions of random variables. PSense supports the following distributions: Bernoulli, Uniform, Binomial, Geometric, Poisson (discrete), Normal, Uniform, Exponential, Beta, Gamma, Laplace, Cauchy, Pareto, Student's t, Weibull, and Rayleigh (continuous).

PSI has the ability to symbolically analyze probabilistic programs with *uncertain variables*. They may take any value and do not have a specified distribution. Uncertain variables in PSI are specified as arguments of the main function. PSense uses uncertain variables to represent noise.

3.2 Sensitivity Metrics

To compare the distributions, PSense uses a developer-selected metric. PSense currently supports several standard metrics for continuous and discrete distributions. Let P be a cumulative distribution function over the support Ω. For probabilistic programs, we denote the distribution represented by the noisy program as $P_{\texttt{maineps}} = P[\texttt{main} \mid \texttt{eps}]$ and the distribution represented by the original program (without noise) as $P_{\texttt{main}} = P[\texttt{main} \mid \texttt{eps} = 0]$. We present the metrics for discrete distributions:

* **(ED1)** Expectation distance: $D_{\text{ED1}} = |\text{E}_{X\sim\texttt{main}}[X] - \text{E}_{Y\sim\texttt{maineps}}[Y]|$, which was defined in [4].
* **(ED2)** Expectation distance (alternative): $D_{\text{ED2}} = \text{E}\left[|X - Y|\right]$, where $X \sim \texttt{main}, Y \sim \texttt{maineps}$; It is a more natural definition of distance, generalizing absolute distance, but harder to compute (as it is not easy to decompose).
* **(KS)** Kolmogorov-Smirnov statistic: $D_{\text{KS}} = \sup_{\omega\in\Omega} |P_{\texttt{main}}(\omega) - P_{\texttt{maineps}}(\omega)|$.
* **(TVD)** Total variation distance: $D_{\text{TVD}} = \frac{1}{2}\sum_{\omega\in\Omega} |P_{\texttt{main}}(\omega) - P_{\texttt{maineps}}(\omega)|$.
* **(KL)** Kullback-Leibler divergence: $D_{\text{KL}} = \sum_{\omega\in\Omega} P_{\texttt{maineps}}(\omega) \log \frac{P_{\texttt{maineps}}(\omega)}{P_{\texttt{main}}(\omega)}$.

The metrics for continuous distributions are defined analogously, replacing sums with the corresponding integrals. The metrics provide several computational challenges, such as (1) integrations in ED2, TVD, and KL, (2) mathematical optimization in KS, and (3) non-linearity in ED2, KS, and KL.

3.3 PSense Symbolic Analysis

Algorithm 1 presents the pseudo-code of PSense's analysis algorithm. The symbolic analysis goes through several stages and synergistically leverages the capabilities of PSI and Mathematica. We describe each stage below.

Algorithm 1 PSense Algorithm

INPUT : Program Π, Sensitivity Metric M
OUTPUT : Sensitivity Table $T : Param \rightarrow Expr \times Bool$

```
 1: procedure PSense
 2:     P ← IdentifyParams(Π)
 3:     d ← PSI(Π)
 4:     for p ∈ P do
 5:         Π′ ← transformProgram(Π, p)
 6:         d_ε ← PSI(Π′)
 7:         s ← distrSupport(d_ε)
 8:         Δ_0 ← M(d, d_ε)
 9:         Δ ← MathematicaSimplify(Δ_0, s)
10:         if (doApproximate ∧ hasIntegral(Δ)) then
11:             Δ ← approximateIntegral(Δ, s)
12:         end if
13:         l ← isLinear(Δ, s)
14:         T[p] ← (Δ, l)
15:     end for
16: return T
17: end procedure
```

Identifying Noisy Parameters. PSense's front end identifies all parameters that are used inside the distribution expressions (such as `flip(0.5)` in the first example) and observations (such as `observe(D>=2)` in the same example). For each of these parameters, PSense generates a probabilistic program that expresses uncertainty about the value of the parameter. PSense leverages the uncertain variables, such as `eps`, to generate legal PSI programs with noise.

Computing Posterior Distribution with Noise. For each program with uncertain variables, PSI computes symbolic distribution expressions (both probability mass/density and cumulative distribution functions) parameterized by the uncertain variables. PSI can work with programs that have discrete or continuous distributions. Many of PSI's simplification and integration rules can operate on programs with uncertain variables and produce posterior distributions that fully solve integrals/summations. In the analysis of while loops (which are

unrolled up to some constant, after which a status assertion will fail), the dependence of the iteration count on `eps` will be reflected through the probability of failure, which will also be a function of `eps`.

Both PSense and PSI analyses keep track of the legal parameter values for the distribution parameters. Based on this, PSense can then automatically determine the legal bounds of the noise variables. For instance, for `flip(0.7+eps)`, the computed distribution expression will specify that the variable `eps` should be between -0.7 and 0.3 (because the parameter of Bernoulli is between 0 and 1).

Computing Sensitivity Metrics. In general, one can define the computation of the sensitivity as a product program that has two calls to `main` with and without noise, however we optimize the analysis to skip the computation of the posterior for the original program (without noise), since we can obtain it by substituting `eps` with zero. After computing the distribution expression for one program with PSI, PSense calls Mathematica to compute and simplify the expression of the sensitivity metric. Some metrics, such as KS and ED2 may take advantage of the support Ω of the distribution, to successfully simplify the expression. PSense implements a support computation as Mathematica code.

To address these challenges, we combine the solving mechanisms from Mathematica and PSI. Our experience is that Mathematica has a more powerful simplification engine (when it scales) and has capabilities to perform symbolic and numerical optimization and interpolation, which are out of the scope of PSI.

To support the symbolic analysis and provide an additional context to the user, we implemented several procedures that check for various properties of the functions of the noise variable `eps`:

- ***Linearity Check:*** We have two linearity checks. The exact version checks the standard property from calculus, that the derivative of the function is a non-zero constant with respect to `eps`. An alternative approximate version searches for the upper and lower linear coefficients that tightly bound the function (as tangents that touch it). If the distance between these lines is within a specified tolerance, PSense reports approximate linearity.
- ***Convexity/Concavity Check:*** For this check, we also implement the test in Mathematica based on the standard definition from calculus: a function of a variable `eps` is convex (resp. concave) if the second derivative is non-negative (resp. negative) for all legal values of `eps`. To establish this property, we set the appropriate inequalities and query Mathematica to find counterexamples. PSense returns if the expression is in any of these categories.
- ***Distribution support:*** A distribution support is a set of values for which the distribution mass/density function is non-zero. Knowing support is critical for efficient computation of sums and integrals that appear in the distance expressions, especially for optimization problems and for the optional approximate integration. Surprisingly, solving for support in Mathematica is not straightforward. Among several alternatives, we found that the most effective one was the built-in function `FunctionDomain[f]`, which returns

the interval on which the function `f` is defined. To use it, we redefine the query to check the domain of a fractional function that is defined only when `dist[eps]` is non-negative.

– ***Numerical Integration:*** The analysis of continuous distributions may produce complicated (potentially multidimensional) integrals. Since not all integrals have a closed-form, PSense implements numerical approximation that evaluates integrals that PSI/Mathematica could not solve. The numerical integration can be optionally selected by the user. The approximation creates a hypercube of the parameter values and samples the values of `eps` and the other variables at the regular boundaries. It uses the distribution support computed by PSense's analysis and relies on the user to set up the lower/upper integration bounds.

Properties. Soundness of the technique follows from the soundness of the underlying solvers: given the legal intervals for the uncertain variables, both PSI and Mathematica do sound simplification of mathematical expressions; In addition, PSense's analyses for determining distribution support, linearity and convexity are derived from the standard mathematical definitions. The time complexity of the analysis is determined by the underlying inference (which is #P for discrete programs) and algebraic simplifications.

Global vs. Local Sensitivity. In the present analysis, the value of `eps` is bound only by the legality range and can assume any value. This therefore enables us to conduct *a global* sensitivity analysis, which asks a question, whether some property about the distribution holds for *all* values of `eps`. This is in contrast to *a local* sensitivity analysis, which assumes that `eps` is a constant small perturbation around an input x_0, e.g., $x_0 - 0.1$ and $x_0 + 0.1$. Computing the local analysis follows directly from the result of the global analysis.

Our approach can, in principle, also analyze multiple uncertain variables in parallel (*multi-parameter* sensitivity analysis). While PSense algorithm would apply to this setting, we note that when selecting all variables as noisy, the current solvers would not be able to apply effective simplification on such expressions (unless most of noise variables are 0).

3.4 Sampling-Based Sensitivity Algorithm

We also implemented a sampling backend as an approximate alternative to the exact symbolic analysis. For a concrete numerical value of noise (e.g., 1%, 5%, or 10% of the original value), the sampling backend translates the program with and without noise to WebPPL, a popular probabilistic language with an approximate MCMC backend and runs its inference. The translation between PSI and WebPPL is mostly straightforward, except for the loops, which are translated as recursions. The translated program calls the two functions, `main` and `maineps`, which are the translated functions, and `eps` is a constant:

```
var sensitivity = function() {
   var eps = 0.01;
   var r1 = main();
   var r2 = maineps(eps);
   return sensitivity_metric(r1, r2);
}
var dist = Infer({method: 'MCMC', samples: 1000},  sensitivity);
```

While the sampling-based sensitivity analysis will typically work for a wider variety of probabilistic programs than the symbolic analysis, it has at least three important limitations: (1) it may produce imprecise results, especially when `eps` is small and therefore a user cannot rely on it soundness, and (2) it works only for concrete values of `eps`, and cannot express global properties (for all `eps`), and (3) it cannot be used in the optimization queries we describe next.

4 Optimization

PSense can leverage the results from the symbolic analysis to formulate and solve sensitivity-related optimization problems.

Maximum Acceptable Perturbation. This optimization problem seeks the answer to the question: *What is the maximum absolute noise of the input such that the distance between the output distributions does not exceed a provided constant?* A user provides an acceptable threshold τ on the distribution distance of their choice. We then leverage PSense analysis (Section 3) to get the symbolic expression for the distribution distance $\Delta(\varepsilon)$ for a noise variable ε. We define the optimization problem as follows:

$$
\begin{array}{ll}
\textbf{\textit{Maximize:}} & |\,\varepsilon\,| \\
\textbf{\textit{Constraints:}} & 0 \leq \Delta(\varepsilon) \leq \tau \\
 & \textsc{LegalityChecks}(\varepsilon) \\
\textbf{\textit{Variable:}} & \varepsilon \in \textsc{Domain}
\end{array}
$$

The optimization problem maximizes the absolute value of ε subject to the constraint given by the distance expression. In general, a distance expression Δ may have multiple branches (expressed as `Boole` functions). In such cases, we break Δ into non-overlapping branch components and make sure all of them are within the bound τ. We also support a non-symmetric optimization problem that independently maximizes ε and $-\varepsilon$ to get more precise bounds.

As already mentioned, PSense keeps track of the legal values of the distribution parameters for each standard distribution. These checks typically have the form $a \leq \varepsilon \leq b$. It is possible for a variable to have multiple such (sub)intervals, which we add all to the optimization problem. Finally, ε's domain may be either reals or integers. While most parameters are real (e.g., for Bernoulli and Gaussian), integer noise exists in distributions such as uniform for integers (upper and lower bounds) or negative binomial (first parameter).

The optimization problem is univariate, but the constraint on Δ can be nonlinear. We use Mathematica's function `Maximize[]`, which symbolically solves optimization problems, producing the infinite-precision value for ε. In addition,

PSense runs an auxiliary convexity check, which can indicate whether the found optimum is global one (if the function is convex, then a local maximum is also the global maximum).

Optimization for Local Sensitivity. We can similarly formulate the local-sensitivity query: *What is the maximum distance between the output distributions when the input noise is within an interval* $[x_0 - \sigma, x_0 + \sigma]$*?* ($x_0$ is the original value, and the constant σ is a radius of the ball around it). The optimization problem is formulated similarly as the previous one. The optimization objective is to maximize $\Delta(\varepsilon)$, subject to the constraint $\varepsilon \in [-\sigma, \sigma]$ and legality checks for ε. If $\Delta(\varepsilon)$ has multiple terms, we solve for each and combine. For this problem, we also use Mathematica's `Maximize[]` to compute the (exact) symbolic solution.

5 Evaluation

Our evaluation focuses on the following research questions:

- ⋆ **RQ1:** Is PSense effective in computing the sensitivity of the parameters of prior distributions?
- ⋆ **RQ2:** Is PSense effective in computing the sensitivity of the observations?
- ⋆ **RQ3:** How does the precision of PSense symbolic approach compare to a sampling-based sensitivity analysis?
- ⋆ **RQ4:** Is PSense effective in finding maximum allowed parameter sensitivity subject to the bound on the final noise?

Benchmarks. We evaluated PSense on three sets of programs: (1) 21 benchmark programs from the PSI paper [10], (2) a subset of the programs from the book *Probabilistic Models of Cognition* [14] that we translated into PSI, and (3) three code examples from [4]: SGD that we specialized for regression, one-dimensional population dynamics, and path coupling.

Table 1. Benchmark Statistics

#Progs	66
#Params	357
#Priors/Prog	min: 1 avg: 4.6 max: 16
#Observe/Prog	min: 0 avg: 0.77 max: 10
#LOC	min: 3 avg: 16.8 max: 76

Table 1 presents the statistics of the benchmark programs. In addition to the total number of programs and the constant parameters that can be changed, it also presents the number of statements that specify prior distributions per benchmark, the number of observation statements, and the number of lines of code. Note that even when a probabilistic program has only a few lines of code, they still represent complicated probabilistic models that can be challenging for automated analyses.

Setup. We analyzed the programs with five sensitivity metrics defined in Section 3.2. We set the timeout for computing the individual metric to 10 minutes. We performed the experiments on Xeon CPU E5-2687W (3.00GHz) with 64GB RAM, running Ubuntu 16.04.

Table 2. Sensitivity to Perturbation of Priors

	Discrete					Continuous				
Metric	OK	Fail	T/O	N/A	Time(s)	OK	Fail	T/O	N/A	Time(s)
ED1	94	44	5	25	4.47±2.08	49	30	39	20	9.84±2.62
ED2	136	1	6	25	18.7±6.82	39	0	79	20	115±30.0
KS	142	2	24	0	27.1±7.90	38	19	81	0	81.3±23.6
TVD	127	7	34	0	19.1±6.58	55*	16	67	0	87.9±26.9
KL	128	17	23	0	23.1±6.18	32*	17	89	0	114±28.0

5.1 Sensitivity to Perturbation of Priors

We computed the sensitivity of the result to the change in each prior parameter. Table 2 presents the counts of the outcomes of parameter sensitivity, separately for discrete and continuous/mixed programs. The first column presents the metrics from Section 3.2. Column "OK" counts the cases for which PSense successfully computed the symbolic noise expression (we denote * if we applied approximate integration). Column "Fail" counts the cases for which PSense was unable to compute the result automatically; we discuss the reasons below. Column "T/O" counts the cases that did not complete within the timeout. Column "N/A" counts cases for which the metrics cannot be applied (e.g., when the program returns a tuple). Finally, Column "Time" presents the average time and standard deviation of the analysis runs.

The results show that PSense can be effective in producing sensitivity information for many benchmark programs. For discrete programs, we analyze all programs fully symbolically and provide exact difference expressions. In addition, for all these programs, we were able to compute the linearity and the maximum input noise that corresponds to the pre-specified output noise in the case of KS distribution distance. For continuous programs, PSense can compute the expectation and KS distances exactly, but for TVD and KL, the majority of the integrals do not have the closed form, and therefore we instructed PSense to compute the approximate integrals.

Some of the PSense analyses failed to produce the results. We manually inspected these programs. For expectation distance, all failures are due to expectation expressions that have multiple cases. For instance, one case when `eps >= 0` and another when `eps < 0`. We currently do not support the sensitivity of such composite expressions, but plan to do so in the future. For KS distance, the failures were due to the internal exceptions in PSI (problems computing results) or in Mathematica's Maximize (returns "Indeterminate"). For TVD/KL, the failures happen when PSense cannot find the distribution support. For continuous distributions, Mathematica's numerical integration (NIntegrate) can result in 0 in the denominator or raise an "Infinity, or Indeterminate" exception. In some cases, we cannot apply the computation – e.g., expectation distances ED1 and ED2 are not defined when the program returns a tuple.

The execution time consists of three components: (1) the time to do PSI analysis, (2) the time to determine the distribution support, and (3) the time to compute the sensitivity metric. Out of those, our current computation of the

Table 3. Sensitivity to Perturbations of Observed Data

	Discrete					Continuous				
Metric	OK	Fail	T/O	N/A	Time(s)	OK	Fail	T/O	N/A	Time(s)
ED1	6	19	1	6	0.45±0.31	6	5	6	2	5.07±3.20
ED2	25	0	1	6	7.58±7.15	8	0	9	2	39.4±23.6
KS	28	0	4	0	4.27±2.46	9	1	9	0	3.22±2.60
TVD	28	0	4	0	3.61±1.94	11*	1	7	0	56.5±35.6
KL	9	20	3	0	22.7±16.1	3*	1	15	0	1.34±0.83

distribution support takes about 20s (for most programs), while the computation of the sensitivity metric takes between 4s (ED2) and 20s (KL). Continuous distributions typically take more time, since the analysis needs to solve complicated integrals or optimizations (e.g., ED2, KS), in contrast to the discrete cases, which only have finite sums. For continuous TVD and KL, the time of approximate integration is proportional to the number of points for which the integrals are numerically computed. Finally, complex integrals cause more timeouts for continuous programs.

5.2 Sensitivity to Perturbations of Observed Data

Similarly, we ran PSense to add a noise variable to the expressions within each observation statement. Table 3 presents the counts of observation sensitivity analyses (one for each observe statement) and their outcomes, separately for discrete and continuous/mixed programs. The columns have the same meaning as for Table 2. We identify the same trends for the ability of PSense to analyze sensitivity as in the case of the prior distributions in Section 5.1 for the majority of metrics. The exceptions are ED1 and KL (discrete cases), where the sensitivity expressions are more likely to be discontinuous or nonlinear because noise variables in observations result in more complicated constraints.

5.3 Solving Optimization Problem

We also present the results of solving the optimization problem, which seeks the maximum absolute value of the noise variable, subject to the bound on the program's final distance. We set the maximum acceptable program threshold to 10% of the true distance. We analyzed only the programs for which PSense (Sections 5.1 and 5.2) gave an "OK" status. We only optimized the exact symbolic expressions, therefore skipping TVD and KL distances for continuous programs.

Table 3 presents the counts of problems that were successfully solved. The columns of the table have the same meaning as in the previous sections. The results show that many of the problems can be successfully (and exactly) solved by the Mathematica backend that PSense calls. For the several cases that failed to produce the result, Mathematica was not able to generate initial points that satisfy the inequality or the solution failed to converge, typically for programs with discrete variables, which resolve to plateaus in optimization. Only a small fraction of analyses experienced timeout, indicating that the current symbolic techniques are effective in solving a variety of problems.

Table 4. PSense Results for Solving Optimization Problems

	Discrete					Continuous				
Metric	OK	Fail	T/O	N/A	Time(s)	OK	Fail	T/O	N/A	Time(s)
ED1	99	0	1	31	3.11±1.81	54	0	1	22	7.26±2.25
ED2	160	1	0	31	14.7±5.43	42	3	2	22	94.7±25.0
KS	138	23	9	0	163±12.4	25	7	15	0	105±29.6
TVD	148	7	0	0	5.72±2.51	-	-	-	-	-
KL	113	21	3	0	13.4±4.46	-	-	-	-	-

Table 5. Symbolic vs. Sampling Algorithm for Expectation Distances

Metric	Total	Diff. (>1-stderr)	Diff. (>2-stderr)	Time Symbolic	Time Sampling
ED1	86	57 (66%)	35 (41%)	0.45±0.027	0.29±0.002
ED2	78	62 (79%)	44 (56%)	0.21±0.003	0.29±0.002

5.4 Comparison with Sampling Sensitivity Analysis

Finally, we compared the results and the execution times of PSense compared to estimating sensitivity using the sampling (WebPPL-based) backend. We set the value of the noise variable to 10% of the original value and run 1000 simulations. Since the sampling backend has to operate only on the concrete values of noise variables, we evaluated symbolic analysis too with the specific noise value. We selected only the programs for which PSI returned that `Pr[error]` (probability of error state) is zero. For each analysis run, we checked if there exists a significant difference between the exact symbolic sensitivity and approximate sensitivity from the simulation by running a statistical t-test with one ($p = 0.32$) and two standard errors ($p = 0.05$).

Table 5 presents the comparison. Column "Total" presents the total number of sensitivity analyses run. Column "Different" presents the number of simulation runs that were significantly different from the exact result, according to the t-test. This backend therefore complements PSense's symbolic analysis. Columns "Time Symbolic" and "Time Sampling" present the average execution times in seconds for the two analyses. Since both analyses operate with a particular numerical value for the noise variable, the run time is much shorter than for the previous analyses that considered the symbolic noise variable. The results show that for a substantial fraction (41% of ED1 analyses and 57% of ED2 analyses), sampling produced a sensitivity estimate that is more than two standard errors away from the exact sensitivity. The trend is even more visible with one standard error distance (66% and 79% of the analyses have a significantly different result). Both indicate that sampling-based analysis is imprecise (for a similar execution time).

6 Related Work

Probabilistic Programming Systems. Recent years have seen a significant interest in probabilistic programming languages [11, 27, 20, 12, 10, 19]. A devel-

oper who wants to check the sensitivity of their models needs to manually modify the programs for every parameter, and since most languages support only approximate inference, the analysis is only valid for concrete values or distribution of noise. In comparison, the goal of PSense is to fully automate the sensitivity analysis and present exact results via symbolic analysis.

Researchers have also looked into various static analyses that compute safe upper bounds of the probabilities of assertions in the program executions, e.g., [23, 3, 9, 26, 16, 21]. We anticipate that the future advances in static analysis and appropriate abstractions, as well as principled combinations of analysis with sampling [20] will improve the scalability of PSense and related analyses.

Comparison with PSI and Mathematica. While PSense leverages PSI's core analysis, PSI alone cannot identify locations of noise variables, compute the distance, run optimization for computing KS distance and other optimization and linearity/continuity queries. PSI's engine cannot solve various integrals arising from TVD and KL. On the other hand, Mathematica is limited when simplifying arbitrary program state expressions [10]. PSense builds on and reformulates the problems hard for PSI as more efficiently computable Mathematica queries and computes hints, e.g., distribution supports, to make the analysis feasible.

Sensitivity Analyses. Sensitivity techniques from machine learning [6, 7, 25, 5, 17] are typically numeric and mainly analyze local sensitivity. For instance, Darwiche and Chan present a framework for testing individual discrete-only parameters of Belief networks [6] and later present how to extend the analysis for multiple parameters and capture their interactions [7]. Like [6], PSense focuses on individual parameters, but can analyze both discrete and continuous distributions. Recently, Llerena et al. [18] present an analysis of perturbed Markov Decision Processes, but only analyze models of systems and do not analyze program code. Barthe et al. presented a logic for reasoning about probabilistic program sensitivity [4]. Unlike PSense, it is manual, requiring a developer to prove properties using a proof assistant, but it supports overapproximation. In contrast, PSense is fully automated and computes various non-linear sensitivity metrics.

7 Conclusion

We presented PSense, a system for automatic sensitivity analysis of probabilistic programs to the perturbations in the prior parameters and data. PSense leverages symbolic algebra techniques to compute the exact sensitivity expressions and solve optimization queries. The evaluation on 66 programs and 357 parameters shows that PSense can compute the exact sensitivity expressions for many existing problems. PSense demonstrates that symbolic analysis can be a solid foundation for automatic and precise sensitivity analysis of probabilistic programs.

Acknowledgements. We thank the anonymous reviewers for the useful comments on the previous versions of this work. This research was supported in part by NSF Grants No. CCF 17-03637 and CCF 16-29431.

References

1. Wikipedia: SGD. https://en.wikipedia.org/wiki/Stochastic_gradient_descent.
2. Mathematica, 2015. https://www.wolfram.com/mathematica/.
3. A. Albarghouthi, L. D'Antoni, S. Drews, and A. Nori. Fairsquare: probabilistic verification of program fairness. In *OOPSLA*, 2017.
4. G. Barthe, T. Espitau, B. Grégoire, J. Hsu, and P. Strub. Proving expected sensitivity of probabilistic programs. In *POPL*, 2018.
5. E. Borgonovo and E. Plischke. Sensitivity analysis: a review of recent advances. *European Journal of Operational Research*, 248(3):869–887, 2016.
6. H. Chan and A. Darwiche. When do numbers really matter? *Journal of artificial intelligence research*, 17:265–287, 2002.
7. H. Chan and A. Darwiche. Sensitivity analysis in bayesian networks: From single to multiple parameters. In *UAI*, 2004.
8. S. Chaudhuri, S. Gulwani, R. Lublinerman, and S. Navidpour. Proving programs robust. In *FSE*, 2011.
9. A. Filieri, C. Păsăreanu, and W. Visser. Reliability analysis in symbolic pathfinder. In *ICSE*, 2013.
10. T. Gehr, S. Misailovic, and M. Vechev. PSI: Exact symbolic inference for probabilistic programs. In *CAV*, 2016.
11. A. Gelman, D. Lee, and J. Guo. Stan: A probabilistic programming language for bayesian inference and optimization. *J. Educational and Behavioral Stats.*, 2015.
12. N. Goodman, V. Mansinghka, D. Roy, K. Bonawitz, and J. Tenenbaum. Church: A language for generative models. In *UAI*, 2008.
13. N. Goodman and A. Stuhlmüller. The design and implementation of probabilistic programming languages, 2014.
14. N. Goodman and J. Tenenbaum. Probabilistic Models of Cognition. probmods.org.
15. P. Gustafson, C. Srinivasan, and L. Wasserman. Local sensitivity analysis. *Bayesian statistics*, 5:197–210, 1996.
16. S. Holtzen, T. Millstein, and G. Broeck. Probabilistic program abstractions. In *UAI*, 2017.
17. B. Iooss and A. Saltelli. Introduction to sensitivity analysis. *Handbook of Uncertainty Quantification*, pages 1–20, 2016.
18. Y. Llerena, G. Su, and D. Rosenblum. Probabilistic model checking of perturbed mdps with applications to cloud computing. In *FSE*, 2017.
19. V. Mansinghka, D. Selsam, and Y. Perov. Venture: a higher-order probabilistic programming platform with programmable inference. *ArXiv 1404.0099*, 2014.
20. P. Narayanan, J. Carette, W. Romano, C. Shan, and R. Zinkov. Probabilistic inference by program transformation in hakaru. In *FLOPS*, 2016.
21. F. Olmedo, B. Kaminski, J. Katoen, and C. Matheja. Reasoning about recursive probabilistic programs. In *LICS*, 2016.
22. A. Saltelli et al. *Global sensitivity analysis: the primer*. 2008.
23. S. Sankaranarayanan, A. Chakarov, and S. Gulwani. Static analysis for probabilistic programs: inferring whole program properties from finitel many paths. In *PLDI'13*.
24. D. Tran, A. Kucukelbir, A. Dieng, M. Rudolph, D. Liang, and D. Blei. Edward: a library for probabilistic modeling, inference, and criticism. *arXiv:1610.09787*, 2016.
25. L. van der Gaag, S. Renooij, and V. Coupé. Sensitivity analysis of probabilistic networks. *Advances in probabilistic graphical models*, pages 103–124, 2007.
26. D. Wang, J. Hoffmann, and T. Reps. PMAF: an algebraic framework for static analysis of probabilistic programs. In *PLDI*, 2018.
27. F. Wood, J. van de Meent, and V. Mansinghka. A new approach to probabilistic programming inference. In *AISTATS*, 2014.