# Debugging Convergence Problems in Probabilistic Programs via Program Representation Learning with SixthSense

Zixin Huang[1*], Saikat Dutta[2] and Sasa Misailovic[1]

[1*]Department of Computer Science, University of Illinois, 201 North Goodwin Avenue, Urbana, 61820, Illinois, USA.

[2]Department of Computer Science, Cornell University, 402 Gates Hall, Ithaca, 14853, New York, USA.

*Corresponding author(s). E-mail(s): zixinh2@illinois.edu;
Contributing authors: saikatd@cornell.edu; misailo@illinois.edu;

**Abstract**

Probabilistic programming aims to open the power of Bayesian reasoning to software developers and scientists, but identification of problems during inference and debugging are left entirely to the developers and typically require significant statistical expertise. A common class of problems when writing probabilistic programs is the lack of convergence of the probabilistic programs to their posterior distributions. We present 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.

**Keywords:** Probabilistic Programming, Debugging, Machine Learning

## 1 Introduction

Probabilistic programs (PP) express complicated Bayesian models as simple computer programs, used in various domains [1–4], including the important applications like epidemic modeling [5] and single-cell genomics [6]. 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) [7]. 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 [8–10] 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 [11].

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 [12]) 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.

## 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 [13–17]. 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 [18]. 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 [13–16].

**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 [19]) 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.

### 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 [20]. 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 [13] and Code2Seq [14]. 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 [13] by 8 percentage points on average and Code2Seq [14] 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%).

### 1.3 Contributions

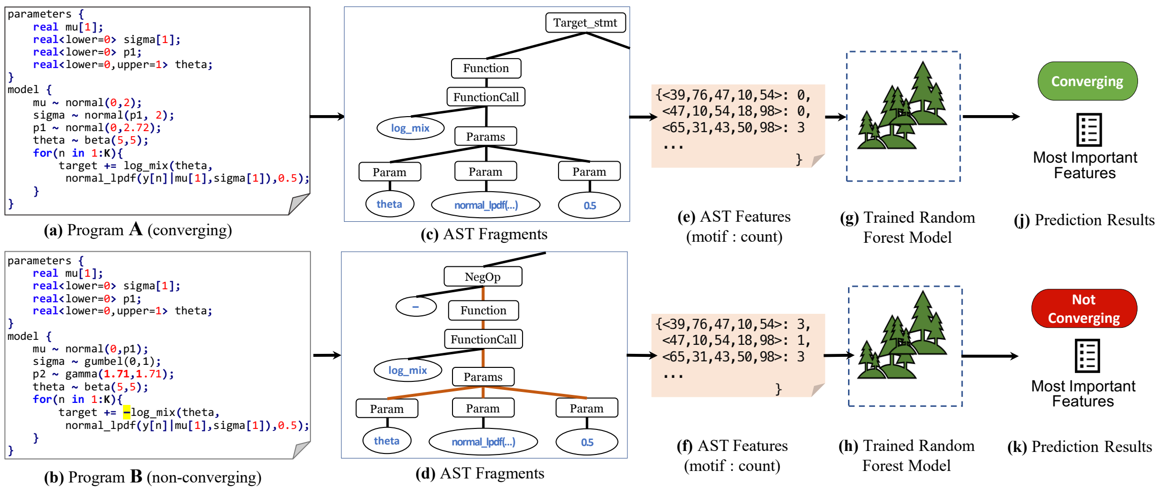
We highlight the main contributions of this paper:

- ⋆ **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.

## 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 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

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



```
parameters {
    real mu[1];
    real<lower=0> sigma[1];
    real<lower=0> p1;
    real<lower=0,upper=1> theta;
}
model {
    mu ~ normal(0,2);
    sigma ~ normal(p1, 2);
    p1 ~ normal(0,2.72);
    theta ~ beta(5,5);
    for(n in 1:K){
        target += log_mix(theta,
         normal_lpdf(y[n]|mu[1],sigma[1]),0.5);
    }
}
```

```
parameters {
    real mu[1];
    real<lower=0> sigma[1];
    real<lower=0> p1;
    real<lower=0,upper=1> theta;
}
model {
    mu ~ normal(0,p1);
    sigma ~ gumbel(0,1);
    p2 ~ gamma(1.71,1.71);
    theta ~ beta(5,5);
    for(n in 1:K){
        target += -log_mix(theta,
         normal_lpdf(y[n]|mu[1],sigma[1]),0.5);
    }
}
```

```
{<39,76,47,10,54>: 0,
 <47,10,54,18,98>: 0,
 <65,31,43,50,98>: 3
 ...
                    }
```

```
{<39,76,47,10,54>: 3,
 <47,10,54,18,98>: 1,
 <65,31,43,50,98>: 3
 ...
                    }
```

**Fig. 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.

of values. Each sub-distribution has an associated probability (called mixing ratio) of being chosen.

The programs **A** and **B** in Figure 1(a), 1(b) have several (unknown) parameters: mean *mu* and variance *sigma* of the Normal sub-distribution; *theta* is the mixing ratio of the 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 [21], 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 1(c) and 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 1(e) and 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 [12]. *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 1(a) and *False* label for program in Figure 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.

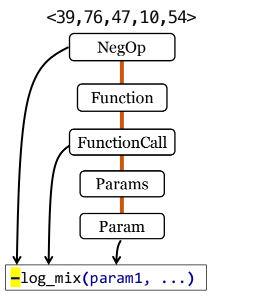


**Fig. 2**: Topmost motif in program **B**

Now consider the scenario where a novice Stan developer used negative log-likelihood in Stan's target statement, and wrote program **B** (Figure 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 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 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.

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 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 Overview

Figure 3 shows the architecture of SixthSense. We next describe each of its components.

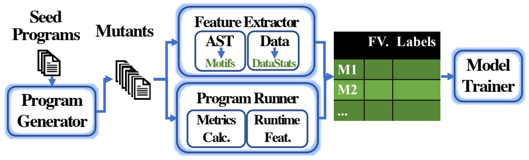


**Fig. 3**: SixthSense Training Workflow

**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 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 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.

## 3.1 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 $X_i$ $(i=1,2,...)$. 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

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

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 [12]. 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 $\theta$, $\{\theta_{mt}\}_{t=1}^{N}$ is the $m$th simulated chain. Let $\hat{\theta}_m$ and $\hat{\sigma}_m^2$ be the mean and variance of $m$th 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:

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

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 $\theta$. Finally, we can compute the Gelman-Rubin diagnostic as follows:

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

In practice, $\hat{R}$ values $\leq 1.1$ are considered as good indicators of convergence [12].

## 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.

# 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 [22] for representing probabilistic program, as shown in Figure 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

| | | |
|---|---|---|
| $x$ | $\in$ | *Vars* |
| $c$ | $\in$ | *Consts*$\cup\{-\infty,\infty\}$ |
| *aop* | $\in$ | $\{+,-,*,/,\}$ |
| *bop* | $\in$ | $\{=,>,\ldots\}$ |
| *Dist* | $\in$ | {Normal,Uniform,...} |
| *ID* | $\in$ | *String* |

```
Type      ::=  Int | Float
Decl      ::=  x : Type | x : [c+]
Expr      ::=  c | x | Expr aop Expr | Expr bop Expr
Stmt      ::=  x = Expr | Decl | observe(Dist(Expr+),x)
               | x ~ Dist(Expr+) | for x ∈ 1..n;{Stmt*}
               | if (Expr) then Stmt* else Stmt*
Query     ::=  posterior(x) | expectation(x)
Program   ::=  Stmt* Query*
```

**Fig. 4**: Syntax of Storm-IR [22]

interpret. Existing approaches Code2Vec/Code2Seq [13, 14] 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 1.** (Abstract Syntax Tree) Similar to [13], 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 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.** (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 4.** (Node Type ID) We define the function $\phi$ : 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 1 (c), the function $\phi$ 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 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 $\mathbb{M}$.

## 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)* [23], 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)* [18]. 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 6.** (Motif Group) Let $\mathbb{M}$ be the set of all motifs, and let $H=[h_1,h_2,...,h_n]$ be a list of $n$ hashing functions. For any motif $m\in\mathbb{M}$, define its hash code as the tuple $c(m)=(h_1(m),h_2(m),...,h_n(m))$. 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 $i \in \{1,2,...,n\}$.

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 $c_M$:

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

In our application, RDP utilizes a list of random projection vectors $[r_1, r_2, ..., r_n]$, 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 7.** (Feature Vector) A program $P$ for a fixed hashing function has a feature vector $\boldsymbol{v} = [v_1, v_2, ..., v_k]$. 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 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):

$$\begin{aligned} \mathit{GetMotifAt}(N,0) &= \varepsilon \\ \mathit{GetMotifAt}(\varnothing,L) &= \varepsilon \\ \mathit{GetMotifAt}(N,L) &= \phi(\tau(N)) :: \\ &\qquad \mathit{GetMotifAt}(\mathit{parent}(N), L-1) \end{aligned}$$

Here $\varepsilon$ 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 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 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.

**Algorithm 1** Compute Feature Vectors

```
    Input: Batch of Programs $Batch$, Motif depth $L$
    Output: Table of Feature Vectors $F$
 1: procedure CalculateFeatures
 2:     $F \leftarrow \emptyset$
 3:     for $P \in Batch$ do
 4:         $\boldsymbol{v} = [0,...,0]$
 5:         for $node \in nodes(AST)$ do
 6:             $m \leftarrow GetMotifAt(node, L)$
 7:             $c_M \leftarrow RDPSimilarityHash(PadRight(m, L))$
 8:             $\boldsymbol{v}[c_M] \leftarrow \boldsymbol{v}[c_M] + 1$
 9:         $F(P) \leftarrow \boldsymbol{v}$
10: return $F$
```

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.

## 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 [24]. 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.

## 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 runtime features can be found at [25].

## 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.

### 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 [26]. 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 [22] for testing PP systems. We added the conjugate replacer as it can simplify the inference.

### 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 [27], scaling to new mean and standard deviation, cube root transform, and random replacement of values with values from the same data set.

### 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) [23], 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 6 (Section 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 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 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 defines the *GeneratePrograms* algorithm, which is used as a subroutine of Algorithm 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.

#### 5.3.1 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 parameter's constraint is that it must 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.

**Algorithm 2** Selecting Mutants

```
Input: Seed Programs $S$, Programs $K$, BatchSize $B$
Output: Program Set $progs$
procedure SelectMutants
    $lsh \leftarrow InitializeLSH()$
    $progs \leftarrow \emptyset$
    while $|progs| < K$ do
        for $s \in S$ do
            $seed \leftarrow ChooseSeed(s, progs)$
            $cand \leftarrow GeneratePrograms(seed, B)$
            for $P \in cand$ do
                $\boldsymbol{v} \leftarrow lsh.GetFeatureVector(P)$
                if $lsh.Neighbors(\boldsymbol{v}) = \emptyset$ then
                    $lsh.StoreVector(\boldsymbol{v}, P)$
                    $progs \leftarrow progs \cup \{P\}$
return $progs$
```

**Algorithm 3** Generating Mutants

```
Input: Seed program $S$, Programs $K$, Max Changes $C$
Output: Program Set $progs$
procedure GeneratePrograms
    $progs \leftarrow \emptyset$
    $i \leftarrow 0$
    while $i < K$ do
        $P' \leftarrow P$
        for $t \in \{1..C\}$ do
            $m \leftarrow chooseMutation()$
            $P' \leftarrow m.mutate(P')$
        if $P' \neq P$ then
            $progs \leftarrow progs.append(P')$
        $i \leftarrow i + 1$
return $progs$
```

**Algorithm 4** Mutation Profiling Algorithm

```
Input: Metrics $Met$, Mutation Combinations $MC$
Output: Mutation Profile $MPS$

procedure Create Mutation Profile
    $scores \leftarrow [0,0,...,0]$
    for $i \in \{1,...,Met.length\}$ do
        $mt \leftarrow Met[i]$
        for $mut \in MC[i]$ do
            if $mt = False$ then
                $scores[mut] \leftarrow scores[mut] - 1/MC[i].length$
            else
                $scores[mut] \leftarrow scores[mut] + 1/MC[i].length$
return $scores$
```

### 5.3.2 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 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.

## 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 [20][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 [28]). These probabilistic programs come with their datasets.

**Inference Engine and Sampling.** NUTS, the default inference engine of Stan [29]. 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

[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.

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) [30]. 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 for training. We use 5-fold cross validation for training. We extract top features using TreeInterpreter [31].

**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.

## 6.1 Baselines

We compare SixthSense to three baselines: The first, **Code2Vec** [13], and the second, **Code2Seq** [14], 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.

## 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:

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

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:

$$\textit{Precision} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Positives}}$$
$$\textit{Recall} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Negatives}}$$

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

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 [32] 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.

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

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

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

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 [33].

## 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 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 [32] 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.

# 7 Evaluation

## 7.1 Predicting Convergence of Inference

Figure 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 6.3).
**Comparison with Code2Vec/Code2Seq.** Figure 5 shows that SixthSense, with solely AST motifs is better than Code2Vec and Code2Seq (see also the ablation study in Section 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.

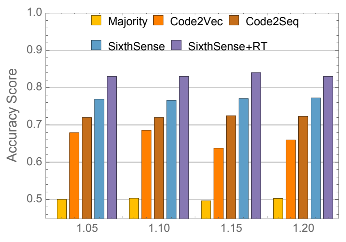
**Table 1**: Precision (**P**) and recall (**R**) ($\hat{R}=1.05$)

| **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 |

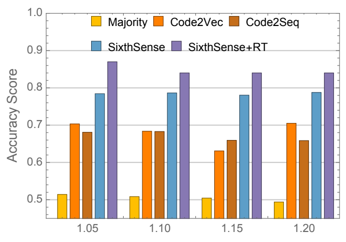
**Table 2**: AUC scores ($\hat{R}=1.05$)

| **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 |

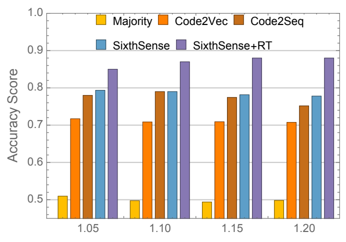
Table 1 presents the precision and recall for $\hat{R}=1.05$. SixthSense exhibits consistently higher precision over Code2Vec (8 to 10 percentage points) and Code2Seq (5 to 10 percentage points). SixthSense also has higher recall than Code2Vec (1 to 7 percentage points), while the recalls of SixthSense and Code2Seq are comparable (within 2 percentage points). Recall



(a) Regression models



(b) Mixture models



(c) Time Series models

**Fig. 5**: SixthSense Prediction Accuracy for Convergence (Measured Using the Gelman-Rubin Diagnostic)

that the precision/recall are averaged over those for different splits and can be more sensitive to small and unbalanced splits.

Table 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 1 and 2 persist for higher thresholds of $\hat{R}$.

**Comparison to Majority Label Classifier.** Figure 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 5 presents the impact of SixthSense's AST features augmented with runtime features (Section 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 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 8.

## 7.2 Debugging Non-Converging Programs

**Table 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 [12, 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 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 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 Section 2. In this example, a novice Stan developer has created a probabilistic program labeled **B** (Figure 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 4. Different from Figure 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 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 4**: Top Motifs for Program **B**

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

## 7.3 Prediction For Different Iteration Counts

**Table 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 6**: AUC Scores for Different Iterations($\hat{R}$=1.05)

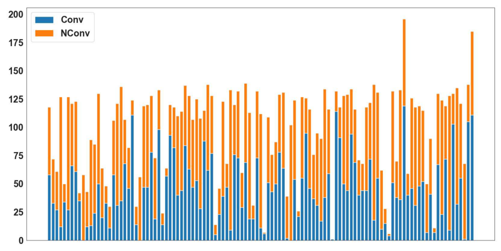
| Class\Iterations | 100 | 400 | 600 | 800 |
|---|---|---|---|---|
| Regression | 0.78 | 0.80 | 0.79 | 0.80 |
| TimeSeries | 0.76 | 0.78 | 0.79 | 0.78 |
| Mixture | 0.75 | 0.74 | 0.74 | 0.75 |

We explore the generality of SixthSense by conducting predictions when the number of MCMC iterations is fewer than the original 1000. Table 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 6 presents the AUC scores for the same experiment.

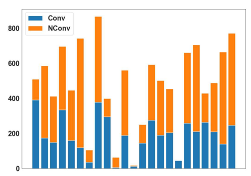
The results shows that even for different distributions of positive and negative labels, SixthSense performs well. Although the labels of individual probabilistic programs change, sometimes significantly (e.g., for 100 iterations), the F1 scores remain consistently high. For AUC we observe a similar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

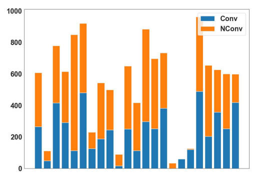
## 7.4 Characteristics of Generated Mutants

Figures 6a, 6c, 6b show the distribution of converging/non-converging mutants in the program set generated by our mutation algorithm (Algorithm 2), after running 1000 iterations. Y-Axis



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





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

**Fig. 6**: Distribution of Converging Mutants for All Seeds. Blue programs are converging, orange are non-converging.

presents the total number of mutants. Each bar gives the count of mutants 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 4 is particularly helpful in generating harder mutants to get around 50% non-converging cases. One might also observe in Figure 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.

### 7.5 Training and Testing Times and Prediction Model Sizes

Table 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 runtime features, as discussed in Section 4.3 for 10-100 iterations. Column 6 shows the time to compute the program's static features (Section 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 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.

## 8 Sensitivity Analysis

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

### 8.1 Feature Ablation Study

Table 9 shows the Accuracy score for convergence predictions when trained with different combinations

**Table 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 |

**Table 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 |

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.

## 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 [34], 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 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 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.

## 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 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 7.1 and 7.2).

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.

## 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.

**Table 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 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 |

**Table 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 |

## 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, 7, 11, 35–41]. 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 [42] 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 [22, 43] target lower-level implementation bugs in probabilistic programming systems. Meanwhile, the statistics community explores enhancing model robustness against data noise [44], 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 [45, 46], usage of seeds in tests [47], and speeding up expensive regression tests [48].

**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 [15], predicting patches [49], summarizing code [16, 50], API discovery [13, 51], and bug detection/repair [52–54]. 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 [13, 14], 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 [55–57], 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) [58–60]. 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 [61–64], 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 Alpha-Code [65], ChatGPT [66] and Codex [67] that can generate a program in seconds for a given natural language prompt. Other systems like [68] 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).

## 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.

**Acknowledgments.** This research was supported in part by NSF Grants No. CCF-1846354, CCF-1956374, CCF-2008883, CCF-2217144, USDA NIFA Grant No. NIFA-2024827, a gift from Facebook, a Facebook Graduate Fellowship, and Microsoft Azure Credits. We would also like to thank Prof. Jian Peng for the useful comments on an earlier draft.

The previous version of this work appeared in FASE 2022 [69].

## References

[1] Minka, T., Winn, J.M., Guiver, J.P., Webster, S., Zaykov, Y., Yangel, B., Spengler, A., Bronskill, J.: Infer.NET 2.5. Microsoft Research Cambridge. http://research.microsoft.com/infernet (2013)

[2] Tehrani, N.K., Arora, N.S., Noursi, D., Tingley, M., Torabi, N., Lippert, E.: Bean machine: A declarative probabilistic programming language for efficient programmable inference. In: PGM (2020)

[3] Modeling Censored Time-to-Event Data Using Pyro. https://eng.uber.com/modeling-censored-time-to-event-data-using-pyro/ (2019)

[4] Flaxman, S., Mishra, S., Gandy, A., Unwin, H.J.T., Mellan, T.A., Coupland, H., Whittaker, C., Zhu, H., Berah, T., Eaton, J.W., et al.: Estimating the effects of non-pharmaceutical interventions on covid-19 in europe. Nature, 1–5 (2020)

[5] Gelman, A.: Stan being used to study and fight coronavirus. Stan Forums (2020). https://discourse.mc-stan.org/t/stan-being-used-to-study-and-fight-coronavirus/14296

[6] Obermeyer, F.: Deep probabilistic programming with Pyro. Models, Inference, and Algorithms (2020). https://www.broadinstitute.org/talks/deep-probabilistic-programming-pyro

[7] Goodman, N., Mansinghka, V., Roy, D.M., Bonawitz, K., Tenenbaum, J.B.: Church: a language for generative models. arXiv preprint arXiv:1206.3255 (2012)

[8] Robert, C., Casella, G.: Monte Carlo Statistical Methods. Springer, New York (2013)

[9] Hoffman, M.D., Gelman, A.: The no-u-turn sampler: adaptively setting path lengths in hamiltonian monte carlo. Journal of Machine Learning Research **15**(1), 1593–1623 (2014)

[10] Neal, R.M.: An improved acceptance procedure for the hybrid monte carlo algorithm. Journal of Computational Physics **111**(1), 194–203 (1994)

[11] Carpenter, B., Gelman, A., Hoffman, M., Lee, D., Goodrich, B., Betancourt, M., Brubaker, M.A., Guo, J., Li, P., Riddell, A.: Stan: A probabilistic programming language. JSTATSOFT **20**(2) (2016)

[12] Gelman, A., Stern, H.S., Carlin, J.B., Dunson, D.B., Vehtari, A., Rubin, D.B.: Bayesian Data Analysis. Chapman and Hall/CRC, New York (2013)

[13] Alon, U., Zilberstein, M., Levy, O., Yahav, E.: code2vec: Learning distributed representations of code. Proceedings of the ACM on Programming Languages **3**(POPL), 40 (2019)

[14] Alon, U., Brody, S., Levy, O., Yahav, E.: code2seq: Generating sequences from structured representations of code. In: International Conference on Learning Representations (2019). https://openreview.net/forum?id=H1gKYo09tX

[15] Raychev, V., Vechev, M., Krause, A.: Predicting program properties from big code. In: ACM SIGPLAN Notices, vol. 50, pp. 111–124 (2015). ACM

[16] Iyer, S., Konstas, I., Cheung, A., Zettlemoyer, L.: 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), pp. 2073–2083 (2016)

[17] Mendis, C., Renda, A., Amarasinghe, S., Carbin, M.: Ithemal: Accurate, portable and fast basic block throughput estimation using deep neural networks. In: ICML (2019)

[18] Bingham, E., Mannila, H.: Random projection in dimensionality reduction: applications to image and text data. In: Proceedings of the International Conference on Knowledge Discovery and Data Mining (KDD) (2001). ACM

[19] Andoni, A., Indyk, P.: Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. Communications of the ACM **51**(1), 117 (2008)

[20] Stan Example Models. https://github.com/stan-dev/example-models (2018)

[21] Stan. Using target += syntax. https://stackoverflow.com/questions/40289457/stan-using-target-syntax (2016)

[22] Dutta, S., Zhang, W., Huang, Z., Misailovic, S.: Storm: Program reduction for testing and debugging probabilistic programming systems. In: FSE (2019)

[23] Datar, M., Immorlica, N., Indyk, P., Mirrokni, V.S.: Locality-sensitive hashing scheme based on p-stable distributions. In: Proceedings of the Twentieth Annual Symposium on Computational Geometry, pp. 253–262 (2004). ACM

[24] Prior Choice Recommendations in Stan. https://github.com/stan-dev/stan/wiki/Prior-Choice-Recommendations (2011)

[25] http://mc-stan.org/users/documentation/index.html (2018)

[26] Raiffa, H., Schlaifer, R.: Applied statistical decision theory (1961)

[27] Sakia, R.: The box-cox transformation technique: a review. Journal of the Royal Statistical Society: Series D (The Statistician) **41**(2), 169–178 (1992)

[28] Inference case studies in knitr. https://github.com/betanalpha/knitr_case_studies (2019)

[29] Gelman, A., Lee, D., Guo, J.: Stan a probabilistic programming language for bayesian inference and optimization. Journal of Educational and Behavioral Statistics (2015)

[30] NearPy. https://github.com/pixelogik/NearPy (2011)

[31] Tree Interpreter Package. https://github.com/andosa/treeinterpreter (2020)

[32] Fawcett, T.: An introduction to roc analysis. Pattern recognition letters **27**(8), 861–874 (2006)

[33] Davis, J., Goadrich, M.: The relationship between precision-recall and roc curves. In: Proceedings of the 23rd International Conference on Machine Learning, pp. 233–240 (2006). ACM

[34] Northcutt, C.G., Wu, T., Chuang, I.L.: Learning with confident examples: Rank pruning for robust classification with noisy labels. In: Proceedings of the Thirty-Third Conference on Uncertainty in Artificial Intelligence. UAI'17. AUAI Press, Sydney, Australia (2017). http://auai.org/uai2017/proceedings/papers/35.pdf

[35] Wood, F., van de Meent, J.W., Mansinghka, V.: A new approach to probabilistic programming inference. In: AISTATS (2014)

[36] Mansinghka, V., Selsam, D., Perov, Y.: Venture: a higher-order probabilistic programming platform with programmable inference. arXiv preprint 1404.0099 (2014)

[37] Goodman, N.D., Stuhlmüller, A.: The design and implementation of probabilistic programming languages. Retrieved 2015/1/16, from http://dippl. org (2014)

[38] Tran, D., Kucukelbir, A., Dieng, A.B., Rudolph, M., Liang, D., Blei, D.M.: Edward: A library for probabilistic modeling, inference, and criticism. arXiv (2016)

[39] Pyro. http://pyro.ai (2018)

[40] Claret, G., Rajamani, S.K., Nori, A.V., Gordon, A.D., Borgström, J.: Bayesian inference using data flow analysis. In: FSE (2013)

[41] Huang, Z., Dutta, S., Misailovic, S.: Aqua: Automated quantized inference for probabilistic programs. In: International Symposium on Automated Technology for Verification and Analysis, pp. 229–246 (2021). Springer

[42] Nandi, C., Grossman, D., Sampson, A., Mytkowicz, T., McKinley, K.S.: Debugging probabilistic programs. In: MAPL (2017)

[43] Dutta, S., Legunsen, O., Huang, Z., Misailovic, S.: 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, pp. 574–586 (2018). ACM

[44] Huang, Z., Dutta, S., Misailovic, S.: Astra: Understanding the practical impact of robustness for probabilistic programs. In: Uncertainty in Artificial Intelligence, pp. 900–910 (2023). PMLR

[45] Dutta, S., Shi, A., Choudhary, R., Zhang, Z., Jain, A., Misailovic, S.: Detecting flaky tests in probabilistic and machine learning applications. In: ISSTA (2020)

[46] Dutta, S., Shi, A., Misailovic, S.: Flex: fixing flaky tests in machine learning projects by updating assertion bounds. In: FSE (2021)

[47] Dutta, S., Arunachalam, A., Misailovic, S.: To seed or not to seed? an empirical analysis of usage of seeds for testing in machine learning projects. In: ICST (2022)

[48] Dutta, S., Selvam, J., Jain, A., Misailovic, S.: Tera: Optimizing stochastic regression tests in machine learning projects. In: ISSTA (2021)

[49] Long, F., Rinard, M.: Automatic patch generation by learning correct code. In: ACM SIGPLAN Notices, vol. 51, pp. 298–312 (2016). ACM

[50] Allamanis, M., Peng, H., Sutton, C.: A convolutional attention network for extreme summarization of source code. In: International Conference on Machine Learning, pp. 2091–2100 (2016)

[51] Wang, K., Su, Z.: Learning blended, precise semantic program embeddings. ArXiv, vol. abs/1907.02136 (2019)

[52] Allamanis, M., Jackson-Flux, H., Brockschmidt, M.: Self-supervised bug detection and repair. Advances in Neural Information Processing Systems **34**, 27865–27876 (2021)

[53] Pradel, M., Sen, K.: Deepbugs: A learning approach to name-based bug detection. Proceedings of the ACM on Programming Languages **2**(OOPSLA), 1–25 (2018)

[54] Xia, C.S., Zhang, L.: 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, pp. 959–971 (2022)

[55] Simard, P.Y., Steinkraus, D., Platt, J.C.: Best practices for convolutional neural networks applied to visual document analysis. In: Icdar, vol. 3 (2003)

[56] Cubuk, E.D., Zoph, B., Mane, D., Vasudevan, V., Le, Q.V.: Autoaugment: Learning augmentation policies from data. arXiv preprint arXiv:1805.09501 (2018)

[57] Taylor, L., Nitschke, G.: Improving deep learning using generic data augmentation. arXiv preprint arXiv:1708.06020 (2017)

[58] Leyton-Brown, K., Hoos, H.H., Hutter, F., Xu, L.: Understanding the empirical hardness of np-complete problems. Communications of the ACM **57**(5), 98–107 (2014)

[59] Khalil, E.B., Le Bodic, P., Song, L., Nemhauser, G., Dilkina, B.: Learning to branch in mixed integer programming. In: Thirtieth AAAI Conference on Artificial Intelligence (2016)

[60] Balunovic, M., Bielik, P., Vechev, M.: Learning to solve smt formulas. In: Advances in Neural Information Processing Systems, pp. 10338–10349 (2018)

[61] Istrate, R., Scheidegger, F., Mariani, G., Nikolopoulos, D., Bekas, C., Malossi, A.C.I.: Tapas: Train-less accuracy predictor for architecture search. arXiv preprint arXiv:1806.00250 (2018)

[62] Dutta, S., Joshi, G., Ghosh, S., Dube, P., Nagpurkar, P.: Slow and stale gradients can win the race: Error-runtime trade-offs in distributed sgd. arXiv preprint arXiv:1803.01113 (2018)

[63] Deng, B., Yan, J., Lin, D.: Peephole: Predicting network performance before training. arXiv preprint arXiv:1712.03351 (2017)

[64] Pu, Y., Narasimhan, K., Solar-Lezama, A., Barzilay, R.: sk_p: a neural program corrector for moocs. In: Companion Proceedings of the 2016 OOPSLA, pp. 39–40 (2016). ACM

[65] Li, Y., Choi, D., Chung, J., Kushman, N., Schrittwieser, J., Leblond, R., Eccles, T., Keeling, J., Gimeno, F., Dal Lago, A., *et al.*: Competition-level code generation with alphacode. Science **378**(6624), 1092–1097 (2022)

[66] ChatGPT: Optimizing Language Models for Dialogue. https://openai.com/blog/chatgpt (2022)

[67] Chen, M., Tworek, J., Jun, H., Yuan, Q., Pinto, H.P.d.O., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., et al.: Evaluating large language models trained on code. arXiv preprint arXiv:2107.03374 (2021)

[68] Svyatkovskiy, A., Deng, S.K., Fu, S., Sundaresan, N.: 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, pp. 1433–1443 (2020)

[69] Dutta, S., Huang, Z., Misailovic, S.: Sixthsense: Debugging convergence problems in probabilistic programs via program representation learning. In: International Conference on Fundamental Approaches to Software Engineering, pp. 123–144 (2022). Springer, Cham