SixthSense: Debugging Convergence Problems in Probabilistic Programs via Program Representation Learning

Saikat Dutta(✉), Zixin Huang, and Sasa Misailovic

University of Illinois, Urbana, Illinois, 61820, USA
{saikatd2,zixinh2,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 [22, 38, 44, 54], including the important applications like epidemic modeling [23] and single-cell genomics [42]. 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) [26]. 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 [28, 40, 48] 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 [9].

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. Gelman-Rubin diagnostic [25]) 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 Gelman-Rubin diagnostic). The decisions of the classifier are interpretable and can be used to suggest which program features leads 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 [4, 5, 31, 37, 47]. 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 [8]. 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 [4, 5, 31, 47].

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 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 [6]) 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 programs, across the three classes, from an open source repository of Stan programs [52]. For each class, SixthSense generated more than 10,000 mutants. 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 [5] and Code2Seq [4]. 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 prediction and recall). SixthSense, with just code features outperforms Code2Vec [5] by 8 percentage points on

average and Code2Seq [4] 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 further demonstrate, by studying 40 non-converging programs, that SixthSense can pinpoint the locations in the code that cause non-convergence for 29 programs. In contrast, Stan's runtime warnings point to non-convergence causes in only 5 programs.

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. In our case study SixthSense helps pinpoint likely cause 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 describe how SixthSense computes motifs, trains the predictor and demonstrate 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 N independent sub-distributions of values. Each sub-distribution has an associated probability (called mixing ratio) of being chosen.

The programs **A** and **B** in Figure 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**,

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

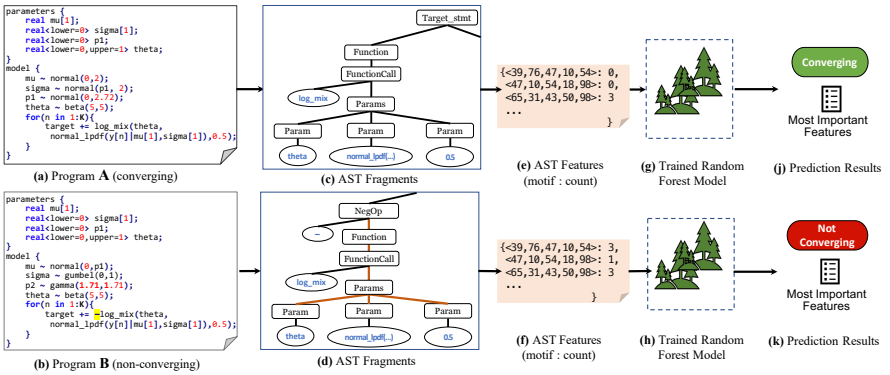


Fig. 1: An example of two models with different convergence behaviors. We obtain the features from the Abstract Syntax Tree (AST) of source code and data (not shown here). We use them as inputs to the trained Random Forest Model for predicting the label (Converging/Not Converging). We can also obtain the most important features which likely contributed to (non)-convergence.

consider a novice developer, who was confused about Stan's target statement [51], calculated the negative likelihood instead.

When run with Stan's default NUTS inference algorithm for 1000 iterations, the program **A** converges and the program **B** does not converge. Our goal is to predict, before running the programs, whether they will converge. If they do not converge, we would also want to know why and use this information to debug the program.

Feature Extraction. First, we extract different classes of features for each program in the corpus of mutants. These include *motifs* – fragments of the 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 are the sequence of the node type IDs as feature vectors.

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 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 (i.e., 'converging', 'not-converging'), SixthSense runs them for the default 1000 iterations using Stan's MCMC algorithm (NUTS). For convergence, we calculate a

well-known diagnostic called Gelman-Rubin ($\hat{R}$) statistic [25]. *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, SixthSense extracts the features and gets 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 for users without expertise or through 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 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 can easily explain its decisions – proves effective in identifying the parts of the program that impede convergence. Namely, we can employ SixthSense's random forest classifier to identify top features. When SixthSense predicts non-convergence, the user can debug the program according to the top features.

Now consider the scenario where a novice Stan developer used negative log-likelihood in Stan's target statement, and wrote program **B** (Figure 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 by altering program code that corresponds to the top features. For program **B**, after the topmost motif indicates the location that contributes to non-convergence, removing the negative sign would allow program to converge. 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.

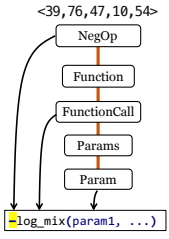


Fig. 2: Topmost motif in program **B**

3 Overview

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

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.

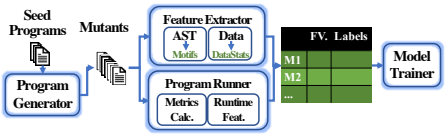


Fig. 3: SixthSense Training Workflow

Program Generation. The generator uses the input set of seed 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) Predict iteration count for which the model will converge.

Deploying the Trained Model. 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 [19] for representing probabilistic program, as shown in Figure 4.

x ∈ *Vars*
c ∈ *Consts*∪{−∞,∞}
aop ∈ {+,−,∗,/,}
bop ∈ {=,>,...}
$Dist$ ∈ {Normal,Uniform,...}
ID ∈ *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 [19]

Representing Program Paths. To understand the causes of non-convergence and for better debuggability, we select a representation that is easy to train and interpret. Existing approaches Code2Vec/Code2Seq [4, 5] 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 [5], we define an AST for a program P as a tuple $<N,T,X,s,\delta,\phi,\psi>$. N is a set of non-terminal nodes, T is the set of terminal nodes, X is a set of values, $s \in N$ is the root node, $\delta : N \rightarrow (N \cup T)*$ is a function which maps each non terminal node to a list of its children, $\phi : T \rightarrow X$ is a function which maps each terminal node to some value, and $\psi : N \rightarrow \mathbb{N}$ maps each non-terminal node to a unique natural number.

Definition 2. (AST Path) An AST path is a path between the nodes in the AST, which starts from one non-terminal node and ends at another non-terminal node, passing through the ancestors of each node at each step.

Definition 3. (Motifs) A Motif encodes an AST path from a node passing through the ancestors of length up to L. For a given AST Path : $\langle N_1, N_2, ..., N_L \rangle$, where $N_i \in \delta(N_{i+1})$, $\forall i \in 1..L-1$, we can define the motif as the list: $\langle I_1, I_2, ..., I_L \rangle$, where $I_m = \psi(N_m), \forall m \in 1..L$.

4.1 Extracting Features from Programs

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 *Random Discretized Projections (RDP)* [8], hashing technique for reducing dimensionality of large feature vectors. It is well-known in data mining, not been used for big-code representation. RDP calculates hash values that are used to group similar items into the same buckets with high probability based on a similarity metric (e.g. cosine similarity). The hash value represents the motif-group in the feature vector.

Extracting Features from Individual Programs. Line 5-9 in Algorithm 1 describes the procedure to extract motifs from a program. We iterate over the nodes in the AST and for each node, to extract a sequence of nodes by visiting the parent nodes up to level L, using the function *GetMotifAt* (line 6), which we define recursively as $GetMotifAt(N,L) = N :: GetMotifAt(parent(N),L-1)$ and base cases $GetMotifAt(\varnothing,L) = \varnothing$ and $GetMotifAt(N,0) = \varnothing$.

The function *SimilarityHash* (line 7) computes a hash key of each motif using the Random Discretized Projections (RDP) [8]. 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. We increase the count for the hash each time a similar motif has obtained the similar hash function (line 8). The RDP has a flexible number of projections and the size of bins. These parameters can be tuned to make similarity more or less fine-grained. They also control, indirectly, the size of the feature vector, the construction of which we describe next.

Calculating Feature Vectors. Given a batch of programs *Batch* and the motif length L, we iterate over the batch to extract the motifs for each program (line 5-9), as described in the paragraph above. Then, to store all the motifs, we first use *InitFVTable* to create a feature vector table F whose column length is equal to the number of programs and the row length is equal to the number of unique motifs (features) across all programs in the batch (line 10). Each row of F is the feature vector of the program *prog*, and each cell stores the count of a motif m in *prog* (line 11-13). *index* maps between the motif hash code and the column index in F. Finally, we output all the feature vectors.

Algorithm 1 Compute Feature Vectors

```
   Input: Batch of Programs Batch, Motif depth L
   Output: Feature Vectors F
 1: procedure CalculateFeatures
 2:     batchMotifs ← ∅
 3:     for prog ∈ Batch do
 4:         progMotifCount = {0,...,0}
 5:         for node ∈ nodes(AST) do
 6:             m ← GetMotifAt(node,L)
 7:             h ← SimilarityHash(PadRight(m,L))
 8:             progMotifCount[h] ← progMotifCount[h]+1
 9:         batchMotifs(prog) ← progMotifCount
10:     F,index ← InitFVTable(Batch,batchMotifs)
11:     for prog ∈ Batch do
12:         for m ∈ batchMotifs(prog) do
13:             F[prog][index(m)] ← batchMotifs(prog)[m]
14: return F
```

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 [2]. 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. These features are algorithm specific. For NUTS, they include *posterior log density* (log probability that the data is produced by the model using current set of the parameters), *tree depth*, *divergence* of the simulated trajectory, *acceptance rate* of the generated sample, *step-size* (the distance between consecutive samples), *leapfrog steps*, and *energy* estimate of Hamiltonian.

5 Program Generation for Training Set Augmentation

In this section, we describe our approach of generating mutant programs from a corpus of seed programs. To produce mutants from the original seed 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. These include *Auxiliary Parameter Creator* which converts a distribution argument to a parameter in the program, *Conjugate Replacer* which replaces prior distributions with distributions conjugate [46] to the likelihood when possible, *Dimension Expander* which expands the dimension of a scalar parameter to match the data dimension, *Constant Replacer* lifts a constant in the program to a parameter with an appropriate distribution, and *Data to Parameter Transformer* randomly replaces a real valued data array with a parameter with the same dimension.

Reducing Mutations. The transformations include *Arithmetic Simplifier*, which replaces arithmetic expressions with either of the operands or changes the arithmetic operation, *Conditional Eliminator* which replaces conditional statements with either of the branches *Distribution Simplifier* which replaces complex distributions like *Laplace*, *Weibull* with common distributions like *Normal* or *Uniform*, *Math-Function Call Eliminator* which replaces common math functions like *log*, *exp*, etc. with constants. These transformations have been previous used by [19] for testing PP systems.

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 [49], 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 important to explore programs with diverse semantic and syntactic features. Our mutation algorithm randomly applies several mutations to the original program. However, to diversify the generated mutants it uses a nearest neighbor based algorithm (Locality Sensitive Hashing [12]), which only selects a representative set of mutants in multiple rounds.

Algorithm 2 Selecting Mutants

```
Input: Seed Programs S, Programs M, BatchSize B
Output: Program Set progs
procedure SelectMutants
    rdp ← InitializeLSH()
    progs ← ∅
    while |progs| < M do
        for s ∈ S do
            seed ← chooseSeed(s,progs)
            p ← GeneratePrograms(seed,B)
            for k ∈ p do
                fv ← feature_vector(k)
                if rdp.neighbours(fv) < 1 then
                    rdp.store_vector(k)
                    progs ← progs.append(k)
return progs
```

Algorithm 3 Generating Mutants

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

Algorithm 2 presents the mutant selection algorithm. The inputs for the algorithm are seed programs S, total number of programs to generate M, and the number of programs to generate in each batch B from each seed program. The algorithm returns the selected mutant programs set *progs* as output. First, we initialize the LSH (Locality Sensitive Hashing) engine. We used four *Random Discrete Projections* hash functions. Next, in each round, we first choose a seed program using the *chooseSeed* function. The *chooseSeed* function randomly chooses among the original seed program s and the mutants generated (in *progs*) from it in earlier rounds. Next, we generate a new batch of programs of size B using *generatePrograms*.

For each new generated program k, we compute its *feature_vector* and number of neighbors among the already generated programs. 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 M programs. The *generatePrograms* algorithm (Algorithm 3) generates M 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 obtained a diverse set of probabilistic programs with a good balance of converging/non-converging behavior.

6 Methodology

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

Seed Probabilistic Models. We collected a corpus of probabilistic models from the most comprehensive open-source repository of Stan Models [52] [2]. Out of total 505 models, we selected the three most common categories: Regression (120 models), Time-Series (23), and Mixture Models (23, augmented with 3 from [33]). The models come with their datasets.

Inference Engine and Sampling. NUTS, the default inference engine of Stan [24]. We executed all 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 number (the convergence metrics do not change significantly even for 10^6 iterations for the seed models).

Feature Extraction. We used a Python based implementation of Randomized Discretized Projection [1]. We set its hyper-parameters 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 [56].

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, Metrics, and Classification

Baselines. We compare SixthSense to three baselines: The first, Code2Vec [5], and the second, Code2Seq [4], 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. Finally, the third baseline, the majority classifier assigns the most likely label during the training to all the predicted programs. It indicates the prediction 'hardness' when the training set is disbalanced.

Metrics. We used a common metric for measuring convergence, called the *Gelman-Rubin* ($\hat{R}$) [25] diagnostic. Ideally, the value of this metric should be close to 1.0. If the observed value of $\hat{R}$ is e.g., 1.05 it is considered as good indication of convergence. The larger values, e.g., 1.5 and greater, are considered as weaker evidences for convergence. Given the threshold, we assign the label *True* to a program if the metric value is within the threshold and *False* otherwise.

[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 models. As a further challenge, Stan programs require the corresponding data set of sufficient size, which many Stan programs on Github do not have. Finally, most of publicly available programs are tuned to converge to their available data-sets.

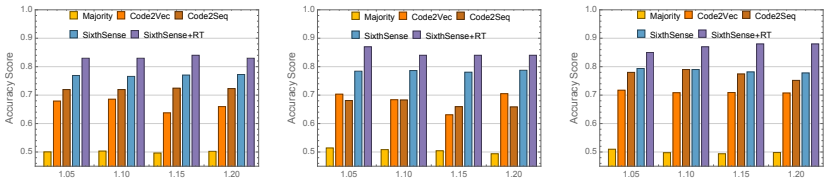
6.2 Evaluation Experimental Setup

Training and Test Sets. We generate a corpus of mutants programs for each seed 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 model and its mutants in test set allows us to stress-test the model predictor. We choose this evaluation strategy because the number of original seed programs in each class is low compared to conventional big-code data-sets. 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, Accuracy, and AUC [21] 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.

Accuracy and AUC are adequate metrics for our scenario: Since we perform training by creating a test-train split for every seed program and its mutants (Section 6.2), in some cases the test-set can become imbalanced, e.g. no or few positive labels/no true and false positives or extremely different sizes of the splits.



(a) Regression models (b) Mixture models (c) TimeSeries models

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

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

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
Time Series	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
Time Series	0.86	0.89	0.79

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 6.2).

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 models, 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 TimeSeries models, the accuracy scores of SixthSense is by 1 percentage point higher than Code2Seq.

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 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

When SixthSense's learned model predicts that a model 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 model 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 model 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 model 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 model 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 model, we run inference to see if it converges. We further check if the model become accurate (or correct) after the fix, since non-convergence often causes inaccurate (or wrong) result. For each model, we apply accuracy tests from Bayesian model checking [25, Ch.6]: we compute the mean squared error to compare the new model result 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 models. Column 2 (**#Models**) presents the number of mutant models 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.

Table 3: Debugging Non-Converging Models

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

Overall, we were able to identify the problem and let 29 updated models converge out of 40 models. Specifically, we corrected 16 models 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 models that we were not able to update, we believe that the model correction would require more complex changes than those we specified in setup above.

Out of 29 updated, now converging models, we ran SixthSense again. It correctly predicted that 21 will converge (with 8 from Regression, 8 from TimeSeries 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 model. 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 models (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 model. 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.

8 Sensitivity Analysis

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

Table 4: 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 5: Training w. 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

8.1 Feature Ablation Study

Table 4 shows the Accuracy score for convergence predictions when trained with different combinations of feature groups (AST features, AST and data features, and all features). Runtime features are from 200 warmup iterations. The AST features (motifs) alone contribute a major portion to the Accuracy scores in all cases. Data features do not have much impact on these models. Runtime features, after a certain number of iterations further improve prediction (they are in fact a strong predictor, but do not establish a relation with the program code). Obtaining runtime statistics comes at a cost of compiling and running the program. This cost is often over 30 seconds for Stan.

Impact of the noisy labels on the prediction. To evaluate the robustness of our prediction, we perturb the class labels in the training set with different noise levels. We use the version of SixthSense, which applies Rank Pruning [41]. Table 5 shows the Accuracy scores for the different model classes for several noise levels (1-5%). For each noise level, *Robust* column shows the scores when trained using the Rank pruning algorithm and *Basic* column shows the scores for baseline 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%) even when 5% labels are wrong.

Other sensitivity studies. We also performed other sensitivity studies on the features and generated programs. 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 then removed overlapped motifs, which resulted in the reduction of the Accuracy scores (by 2 to 5 percentage points). Other experiments, such as different LSH configurations to remove syntactically similar programs from the training set did not show substantial deviation from the reported scores.

9 Related Work

Probabilistic Programming. Probabilistic programming languages (PPLs) and their underlying inference systems have recently gained significant interest from research and industry [9, 10, 26, 27, 29, 36, 38, 45, 55, 58]. Tyically, 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 [39] requires augmenting Bayesian network representation with additional labels and requires extending the inference algorithm.

However, its applicability is limited since state-of-the-art PP systems typically do not use Bayesian network representation. Our approach learns program features useful for debugging without modifications to the inference algorithm. Existing tools [15, 19] find lower-level implementation bugs in probabilistic programming systems.

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 [17, 18], usage of seeds in tests [14], and speeding up expensive regression tests [16].

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 [47], predicting patches [35], summarizing code [3, 31], and API discovery [5, 57]. However, all of these works apply learning on 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 [4, 5], 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 [11, 50, 53], 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) [7, 32, 34]. 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 [13, 20, 30, 43], but their techniques and applications are orthogonal to ours.

10 Conclusion

We presented SixthSense, a novel approach and system, which predicts convergence for probabilistic programs and helps guide the debugging of convergence issues. We show SixthSense is effective in extracting features from probabilistic programs and learning a prediction model. Compared to the state-of-the-art techniques, our results show significant improvement in accuracy.

Acknowledgments

This research was supported in part by NSF Grants No. CCF-1846354, CCF-1956374, CCF-2008883, 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.

References

1. Nearpy (2011), https://github.com/pixelogik/NearPy
2. Prior choices (2011), https://github.com/stan-dev/stan/wiki/Prior-Choice-Recommendations
3. 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)
4. 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
5. 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)
6. Andoni, A., Indyk, P.: Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. Communications of the ACM **51**(1), 117 (2008)
7. Balunovic, M., Bielik, P., Vechev, M.: Learning to solve smt formulas. In: Advances in Neural Information Processing Systems. pp. 10338–10349 (2018)
8. 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). ACM (2001)
9. 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)
10. Claret, G., Rajamani, S.K., Nori, A.V., Gordon, A.D., Borgström, J.: Bayesian inference using data flow analysis. In: FSE (2013)
11. Cubuk, E.D., Zoph, B., Mane, D., Vasudevan, V., Le, Q.V.: Autoaugment: Learning augmentation policies from data. arXiv preprint arXiv:1805.09501 (2018)
12. 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. ACM (2004)
13. Deng, B., Yan, J., Lin, D.: Peephole: Predicting network performance before training. arXiv preprint arXiv:1712.03351 (2017)
14. 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)
15. 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. ACM (2018)
16. Dutta, S., Selvam, J., Jain, A., Misailovic, S.: Tera: Optimizing stochastic regression tests in machine learning projects. In: ISSTA (2021)
17. Dutta, S., Shi, A., Choudhary, R., Zhang, Z., Jain, A., Misailovic, S.: Detecting flaky tests in probabilistic and machine learning applications. In: ISSTA (2020)
18. Dutta, S., Shi, A., Misailovic, S.: Flex: fixing flaky tests in machine learning projects by updating assertion bounds. In: FSE (2021)
19. Dutta, S., Zhang, W., Huang, Z., Misailovic, S.: Storm: Program reduction for testing and debugging probabilistic programming systems. In: FSE (2019)
20. 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)

21. Fawcett, T.: An introduction to roc analysis. Pattern recognition letters **27**(8), 861–874 (2006)
22. 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 pp. 1–5 (2020)
23. Gelman, A.: Stan being used to study and fight coronavirus (2020), https://discourse.mc-stan.org/t/stan-being-used-to-study-and-fight-coronavirus/14296, Stan Forums
24. Gelman, A., Lee, D., Guo, J.: Stan a probabilistic programming language for bayesian inference and optimization. Journal of Educational and Behavioral Statistics (2015)
25. Gelman, A., Stern, H.S., Carlin, J.B., Dunson, D.B., Vehtari, A., Rubin, D.B.: Bayesian data analysis. Chapman and Hall/CRC (2013)
26. Goodman, N., Mansinghka, V., Roy, D.M., Bonawitz, K., Tenenbaum, J.B.: Church: a language for generative models. arXiv preprint arXiv:1206.3255 (2012)
27. Goodman, N.D., Stuhlmüller, A.: The design and implementation of probabilistic programming languages (2014)
28. 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)
29. 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. Springer (2021)
30. 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)
31. 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)
32. 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)
33. Inference case studies in knitr (2019), https://github.com/betanalpha/knitr_case_studies
34. 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)
35. Long, F., Rinard, M.: Automatic patch generation by learning correct code. In: ACM SIGPLAN Notices. vol. 51, pp. 298–312. ACM (2016)
36. Mansinghka, V., Selsam, D., Perov, Y.: Venture: a higher-order probabilistic programming platform with programmable inference. arXiv preprint 1404.0099 (2014)
37. Mendis, C., Renda, A., Amarasinghe, S., Carbin, M.: Ithemal: Accurate, portable and fast basic block throughput estimation using deep neural networks. In: ICML (2019)
38. Minka, T., Winn, J., Guiver, J., Webster, S., Zaykov, Y., Yangel, B., Spengler, A., Bronskill, J.: Infer.NET 2.5 (2013), microsoft Research Cambridge. http://research.microsoft.com/infernet
39. Nandi, C., Grossman, D., Sampson, A., Mytkowicz, T., McKinley, K.S.: Debugging probabilistic programs. In: MAPL (2017)
40. Neal, R.M.: An improved acceptance procedure for the hybrid monte carlo algorithm. Journal of Computational Physics **111**(1), 194–203 (1994)
41. 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 (2017), http://auai.org/uai2017/proceedings/papers/35.pdf

42. Obermeyer, F.: Deep probabilistic programming with pyro (2020), https://www.broadinstitute.org/talks/deep-probabilistic-programming-pyro, models, Inference, and Algorithms
43. 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. ACM (2016)
44. Modeling censored time-to-event data using pyro (2019), https://eng.uber.com/modeling-censored-time-to-event-data-using-pyro/
45. Pyro (2018), http://pyro.ai
46. Raiffa, H., Schlaifer, R.: Applied statistical decision theory (1961)
47. Raychev, V., Vechev, M., Krause, A.: Predicting program properties from big code. In: ACM SIGPLAN Notices. vol. 50, pp. 111–124. ACM (2015)
48. Robert, C., Casella, G.: Monte Carlo statistical methods. Springer Science & Business Media (2013)
49. Sakia, R.: The box-cox transformation technique: a review. Journal of the Royal Statistical Society: Series D (The Statistician) **41**(2), 169–178 (1992)
50. Simard, P.Y., Steinkraus, D., Platt, J.C.: Best practices for convolutional neural networks applied to visual document analysis. In: Icdar. vol. 3 (2003)
51. Stan. using target += syntax (2016), https://stackoverflow.com/questions/40289457/stan-using-target-syntax
52. Stan Example Models (2018), https://github.com/stan-dev/example-models
53. Taylor, L., Nitschke, G.: Improving deep learning using generic data augmentation. arXiv preprint arXiv:1708.06020 (2017)
54. 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)
55. 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)
56. Tree interpreter package (2020), https://github.com/andosa/treeinterpreter
57. Wang, K., Su, Z.: Learning blended, precise semantic program embeddings. ArXiv, vol. abs/1907.02136 (2019)
58. Wood, F., van de Meent, J.W., Mansinghka, V.: A new approach to probabilistic programming inference. In: AISTATS (2014)