RANDOMNESS-AWARE TESTING OF MACHINE LEARNING-BASED SYSTEMS

BY

SAIKAT DUTTA

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois Urbana-Champaign, 2023

Urbana, Illinois

Doctoral Committee:

Associate Professor Sasa Misailovic, Chair
Professor Darko Marinov
Professor Vikram Adve
Associate Professor Milos Gligoric, University of Texas, Austin
Dr. Shuvendu Lahiri, Microsoft Research

ABSTRACT

Machine Learning (ML) is rapidly revolutionizing the way modern-day systems are developed. However, testing ML-based systems is challenging due to 1) the presence of non-determinism, both internal (e.g., stochastic algorithms) and external (e.g., execution environment), and 2) the absence of well-defined accuracy specifications. Most traditional software testing techniques widely used today cannot tackle these challenges because they often assume determinism and require a precise test oracle.

This dissertation presents work on automated testing and debugging of ML-based systems and on improving developer-written tests in such systems. To achieve these goals, this dissertation presents principled techniques that build on mathematical foundations from probability theory and statistics to reason about the underlying non-determinism and accuracy. The presented techniques help developers to detect more bugs and to efficiently navigate trade-offs between test quality and efficiency.

This dissertation presents contributions along two key directions. First, a key challenge in testing ML-based systems is generating tests that can systematically and effectively explore the input space of the algorithms under test. However, because the inputs to an ML algorithm are complex objects, like a deep neural network model and data, generating inputs that are both syntactically and semantically correct is challenging. Additionally, ML algorithms do not come with well-defined test oracles, which makes it difficult to reason about correctness. This dissertation presents systematic test generation and debugging techniques that tackle these challenges by combining techniques from programming languages, differential testing, and probabilistic reasoning. These techniques have helped detect more than 50 previously unknown bugs in ML libraries and enabled faster debugging of failures.

Second, this dissertation presents techniques that improve the quality of regression tests in ML libraries. When writing such tests, developers often fail to adequately account for the randomness of the algorithms under test and rely on guesswork in selecting various test configurations. Consequently, such regression tests often end up being either flaky, i.e., they pass or fail non-deterministically for same code, become expensive to run, or have lower fault-detection effectiveness. This dissertation presents novel test repair techniques that combine principled statistical methods, mathematical optimization, and domain knowledge to systematically tackle these challenges. These techniques have already improved the quality of over 200 tests in over 60 open-source ML libraries, many of which are used at companies like Google, Meta, Microsoft, and Uber as well as in many academic and scientific communities.

To my family, with love.

ACKNOWLEDGMENTS

It takes a village to raise a PhD! As I come close to the end of my PhD, I realize there are several people that have contributed to making this journey possible. I am forever grateful to each one of you!

First and foremost, I would like to thank my advisor, Sasa Misailovic, who put faith in me in the first place and offered to take me on as his PhD student. Over the years, Sasa has not only guided me on research but also helped me get through the innumerous ups and downs of the PhD life and keep my sanity. Throughout my Ph.D., Sasa has motivated me, challenged me, inspired me, listened to me, supported me, and mentored me. Whenever I needed him, Sasa has always been there for me, just a 'slack' message or a call away! I believe this can only happen when someone deeply cares about their students and their well-being. While it is impossible to repay him, I take this chance to express my gratitude to him and only hope to pay it forward.

I would also like to thank Darko Marinov, who has also watched me closely all these years and given me feedback and advice on numerous occasions. I am amazed by the sheer effort Darko puts into mentoring and guiding students (including me). I hope I can replicate at least some of it as a future faculty. I also thank Prof. Vikram Adve, Prof. Milos Gligoric, and Dr. Shuvendu Lahiri who served on my thesis committee for their time and commitment.

Several professors helped me tremendously when I was preparing for my academic job interviews with feedback, support, and advice. I am especially thankful to professors: Sayan Mitra, Vikram Adve, Jiawei Han, Christopher Fletcher, Gagandeep Singh, Nan Jiang, Yongjoo Park, Madhusudan Parthasarathy, Darko Marinov, Lingming Zhang, Charith Mendis, Reyhaneh Jabbarvand, Grigore Rosu, and Tianyin Xu. Special thanks to Darko for organizing the PhD Job Search Seminar and to the Mavis Future Faculty Fellowship Program for the weekly seminars for future faculty. I learnt a lot from the faculty panels and the guest lectures about navigating the academic job search and the faculty life after.

I was very fortunate to land up at a place with a brilliant and amazing group of students. I had a terrific experience and a lot of fun sharing the lab with Vimuth Fernando, Keyur Joshi, Zixin Huang, Jacob Laurel, Yifan Zhao, Shubham Ugare, and Ashitabh Misra. Vimuth, always at an arm's length away from me, was a great friend, co-coffee lover, and travel buddy! Zixin has been a brilliant collaborator and co-author on several of my papers. Thanks to Jacob's free driving lessons, I got my driving license. It was great having him around in the lab and at the front line during volleyball. Keyur started the same year as me, we braved many classes and projects together. I am grateful for all the fun banter we had in the lab, all the deadlines

working together, the hikes, the lunches and dinners, the quest for free food, and more.

Over the years, I have collaborated and made friends with a lot of other fellow PhD students at UIUC, including Wing Lam, August Shi, Owolabi Legunsen, Angello Astorga, Chiao Hsieh, Wenyu Wang, Umang Mathur, Adithya Murali, and Liia Butler. I am grateful for all the collaborations, discussions, and the sports we played together. I had a lot of fun co-organizing the weekly sports activity with Wing and Liia. August, having dominated the flaky tests research area himself, was a great collaborator when I started my research on flaky tests in ML. Owolabi mentored and collaborated with me on my first project during my PhD. During this project, Owolabi showed me how to transform a working implementation of an idea to a full research paper, which I had no idea about as a first year student. I could not have asked for a better mentor in my junior years. Owolabi, having conquered the job market few years before me, helped me at each stage of my academic job search from preparing materials, to preparing my job talk, onsite interviews, and decision making. He was always happy to share his experience and wisdom with me, which I am grateful for.

Parts of this dissertation were published at European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE) 2018 (Chapter 2), 2019 (Chapter 3), and 2021 (Chapter 5); International Symposium on Software Testing and Analysis (ISSTA) 2021 (Chapter 6); and International Conference on Software Testing, Verification and Validation (ICST) 2022 (Chapter 4). I thank the anonymous reviewers and the program chairs of these venues for their feedback and valuable suggestions.

During my PhD, I did two internships, one at Microsoft Research and another at Amazon Web Services. At MSR, I worked with Shuvendu Lahiri and Madan Musuvathi. I also collaborated with Max Schäfer (from GitHub) and Diego Garbervetsky. Through the internship, I learned a lot about leveraging machine learning for improving static analyses for JavaScript. At AWS, I worked with Daniel Kroening and Willem Visser, on testing deep learning compilers. Unfortunately, the pandemic forced the internships to be virtual. However, both companies ensured sure that I have a great experience and make connections.

I am grateful to several institutions that have funded my research during my PhD. In particular, my research has been funded by National Science Foundation (NSF) grants: CCF-1703637 and CCF-1846354, Facebook research gift, Facebook PhD Fellowship, and 3M Foundation Fellowship.

Finally, and most importantly, I thank my family for their love and support throughout my life. My parents and and my elder brother, through their upbringing, shaped me into the person that I am today. They made sure I got the best education and always had what I needed, irrespective of the circumstances. I am also extremely lucky to have my wife, Prasita, in my life. I am grateful for her support, her companionship, and the completeness she brings to my life.

CONTENTS

Chapter 1: INTRODUCTION

Machine Learning (ML) is rapidly revolutionizing the development of many modern-day systems. ML algorithms such as Deep Learning [1], Reinforcement Learning [2], and Probabilistic Programming [3, 4] are increasingly being used in various systems, especially in safety-critical domains like autonomous driving [5, 6], healthcare [7], and finance [8]. However, bugs in such ML-based systems can lead to catastrophic consequences, potentially leading to loss of lives and property [9, 10]. Hence, we must develop testing techniques that can ensure the reliability of ML-based systems. Prior research endeavors have commonly defined the reliability of ML-based systems by incorporating tests for various properties such as correctness, robustness, privacy, efficiency, or fairness. In this dissertation, we adopt a narrower definition of reliability, focusing specifically on testing for correctness, i.e., to ensure that the system is free of programming errors that could potentially result in crashes, non-termination, or inaccurate outcomes.

Software Testing is the most dominant approach used for improving software quality by detecting bugs proactively and during development to avoid software failures post deployment. However, ML-based systems are fundamentally different than traditional systems. In ML-based systems, the decision logic is *learned* from data, unlike traditional systems that typically operate on a concrete set of rules. Most ML algorithms used for training ML models are inherently non-deterministic in nature and lack accuracy specifications, which makes reasoning about correctness challenging. To further understand the key differences, we first show how an ML-based system is developed and then explain what it means to "test" an ML-based system.

Building an ML-based system typically involves training an ML model, which is a multi-step process [11]. Figure 1.1 presents a typical ML training workflow (adopted from [11]). An ML software developer collects data and performs data cleaning and labelling. The developer writes a program that defines the ML model architecture and chooses an ML algorithm for training. Popular deep learning frameworks like TensorFlow and PyTorch, as well as probabilistic programming libraries such as Pyro and Stan, offer implementations of various training algorithms, model components, and other utilities that developers rely on. Finally, this process yields a trained model, which can be a deep neural network in deep learning domain or a statistical model in probabilistic programming domain. The trained model is then deployed and used for inference in a target application such as a self-driving car or medical diagnosis system. Since bugs can occur at any stage of this process, developers need to test each component. We briefly introduce and summarize approaches for testing each stage in the ML training workflow below. For a more comprehensive analysis, please refer to the survey by Zhang et al. [11].

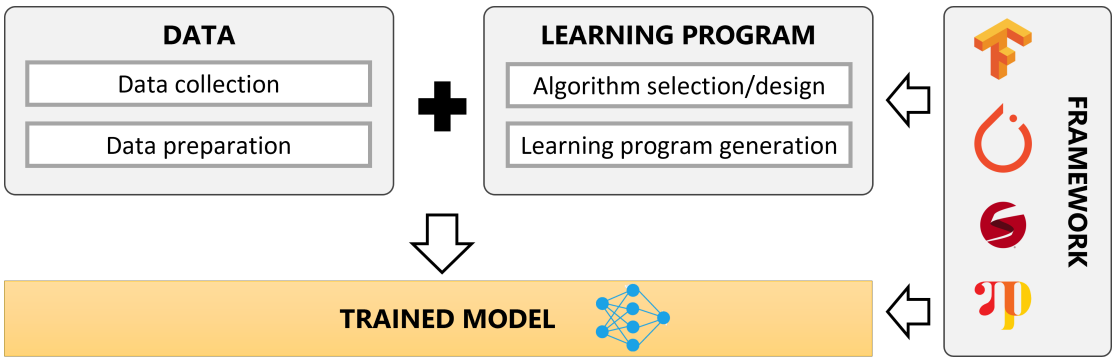


Figure 1.1: ML Training Workflow Components

Testing approaches for ML-based systems can be broadly categorized into four groups:

- **Testing ML libraries:** ML libraries like TensorFlow and PyTorch implement deep learning algorithms. They provide APIs that allow developers to do various tasks such as defining and choosing model architectures (e.g., LSTM or Transformers), model training, and evaluation. Because these libraries implement various key computations (e.g., matrix multiplications, gradient back propagation) behind the scenes, their reliability is of key importance. Bugs in ML libraries can lead to problems in the training workflow and the final system where the trained model is deployed. Prior testing approaches like CRADLE [12] and LEMON [13] proposed differential testing-based approaches to find bugs in deep learning libraries. These approaches, however, focus only on testing the inference phase of the ML models. There is a dearth of automated techniques for testing the training phase of the models, which is the focus and one of the main contributions of this dissertation.

- **Testing Data:** Data is a core component in the ML training workflow, since the model learns the business logic from the data itself. Hence, bugs in the data collection or preparation stage can negatively affect the correctness of the learned model. Previous works have proposed techniques to detect different classes of data-related bugs such as data skew [14, 15], data outliers [16], and adversarial data [17, 18].

- **Testing Trained Models:** Researchers have proposed various methods that test the behavior of the trained model. Such methods typically focus on testing model *inference* (not *training*) and generate test data inputs that lead to crashes, numerical errors, or mis-predictions. DeepXplore [6] was one of the earliest efforts that proposed a white-box differential testing technique to generate test inputs for a deep learning system. Similarly, other approaches include test generation using mutation-based fuzzing [19], metamorphic transformations [20, 21], and symbolic execution [22, 23]. More recently,

Ribeiro and Lundberg [24] combined human feedback and language models for testing and debugging ML models.

- **Testing Learning Program:** Bugs in the learning program (e.g., written in Python) can occur when developers make mistakes in implementing or configuring the training algorithm, e.g., by using incorrect activation functions or algorithm hyperparameters. Previous works [25, 26] have studied the classes of bugs that occur in deep learning programs that use ML libraries like TensorFlow and PyTorch. Murphy et al. [27, 28] proposed, as early as 2009, mutation testing-based approaches to detect bugs in ML programs.

In this dissertation, we focus on automated and systematic testing and debugging of ML libraries that implement various ML algorithms. Such algorithms primarly involve training in various areas of ML, inference for probabilistic models, and other tasks.

1.1 AUTOMATED TEST GENERATION AND DEBUGGING

Testing ML algorithms comes with a unique set of challenges that traditional testing techniques cannot handle. First, unlike traditional software, randomness is pervasive in ML algorithms. Various ML algorithms employ randomness to improve generalizability and speed up convergence, through common methods like dropout regularization, adding gradient noise, and stochastic gradient descent. Additionally, there can be randomness involved in data collection and the order in which the algorithm sees the data (e.g., data batching) as well. Hence, software testing techniques must employ specialized mechanisms to account for randomness when reasoning about the correctness of ML algorithms. Traditional software testing techniques, however, typically assume deterministic behavior. Hence, they are not well-suited for testing ML algorithms.

Second, most ML algorithms exhibit the "no-oracle" problem, i.e., they do not come with any accuracy specifications. Testing techniques typically require a method to compute expected results for any given input. But ML algorithms do not provide any analytical methods to compute the results (e.g., accuracy of trained model on test data) for any given choice of data, model architecture, and algorithm hyper-parameters. They only come with weak statistical guarantees. For example, in probabilistic programming, Bayesian inference algorithms like Markov Chain Monte Carlo (MCMC) are guaranteed to converge to the target probability distribution *in the limit* [29, 30, 31]. In machine learning or deep learning, there exists only weak guarantees such as error bounds on generalization error of the training algorithm [32] and convergence of optimization algorithms like stochastic gradient

descent [33, 34]. Most of such guarantees are, however, either loose, asymptotic, or rest on strong assumptions of data distribution (e.g., convexity) and hypothesis space. As a result, they cannot be easily used to check the correctness of an implementation of an ML algorithm. Therefore, we require alternative ways to test implementations of ML algorithms.

Third, unlike in traditional software testing, the inputs required to test ML algorithms are more complex in nature. For example, to test an ML algorithm we need to generate both input *data* and a *model*. In deep learning domain, the model is a neural network, whereas in probabilistic programming, the model is usually a statistical model (represented as a program). To generate valid inputs, we need to encode both syntactic and semantic information in the test generator. Invalid inputs are typically rejected at the parsing stage of the libraries. Hence, such inputs are not useful in exercising the deeper logic, where more important bugs may reside.

In this dissertation, we will present testing techniques that leverage domain-specific knowledge to systematically generate valid test inputs, i.e., model and data. We will also show how differential comparison of different implementations of an ML algorithm can serve as an effective test oracle. Finally, we will show how such domain-specific knowledge can be used to design efficient analyses for debugging failures in ML libraries.

Testing of other non-deterministic systems. Non-determinism is not a unique property of ML algorithms or ML-based systems. Many other software systems such as real-time systems, concurrent systems, network systems, and distributed systems also exhibit non-determinism in different ways. While researchers have proposed various testing and verification techniques to improve reliability of such systems [35, 36, 37, 38], such techniques cannot be easily extended to test ML algorithms due to fundamental limitations.

Probabilistic and approximate algorithms, such as locality sensitive hashing and count-min sketch, employ randomization to trade accuracy of results for faster execution or reduced memory consumption. Statistical model checking is a popular method often used for verifying properties of such algorithms. For example, Joshi et al. [39] proposed a sampling-based approach for verification of approximate algorithms. Sen et al. [35] leveraged statistical hypothesis testing for verification of black-box probabilistic systems against properties expressed as continuous stochastic logic. However, these approaches require a developer-specified correctness specification, which typically does not come with ML algorithms. Further, they typically only generate simple test inputs, like stream of integers. The inputs to an ML algorithm are however more complex (e.g., model and data) and require more advanced approaches to generate.

A real-time system is any information processing system that has to respond to any given input within a specified time period. Some common examples include air traffic control

systems, fall detection systems, and smartphones. Because such systems often interact with a physical environment, which is typically non-deterministic, testing real-time systems is challenging [40]. Testing techniques for real-time systems, typically aim to generate test inputs that violate a given property (e.g., an alarm should be triggered within 5 time units if a patient falls). To generate test inputs, they either use sampling-based methods [36] or leverage a model of the system under test [41]. Unlike real-time systems, most ML algorithms do not come with any well-defined properties that can be used as test oracle. Further, it is unknown how to model ML algorithms because they often involve performing complex operations (e.g., stochastic gradient descent) on an arbitrarily large number of variables (e.g., billions of model parameters).

Concurrent programs employ multi-threading to enable faster execution, but introduce non-determinism in the form of different thread inter-leavings. Hence, they are prone to bugs that lead to data races and deadlocks. Over the years, several researchers have proposed techniques for testing concurrent programs by exhaustive enumeration of thread interleavings [42], runtime monitoring of memory accesses [37], statistical testing [43], and mutation testing [38, 44]. Even though concurrent programs use multiple threads, an interesting observation is that the output of these programs is expected to remain *unchanged*, similar to traditional or single-thread programs, regardless of the specific thread schedule. Therefore, many testing approaches primarly focus on generating test inputs and thread schedules that either lead to stalled execution (e.g., due to deadlock) or different outputs (e.g., due to data races). In contrast, non-determinism in ML algorithms is typically expected to cause varying outputs across executions (for any given input). However, because there are no well-defined specifications that quantify such variance in outputs, checking for correctness in ML remains challenging.

Prior testing and verification approaches for non-deterministic programs rely either on a model of the system under test or a concrete specification of expected behavior, which is typically unavailable for ML algorithms. These shortcomings motivate the need for novel testing techniques for ML algorithms.

1.2 IMPROVING QUALITY OF REGRESSION TESTS IN ML LIBRARIES

Developers of Machine Learning (ML) libraries write *regression tests* that check for functional correctness of their implementations. Regression testing [45] is the practice of running a set of functional tests on a software whenever a developer introduces a code change. Regression testing is a key component of developer's workflow and guards the software under test against fault-inducing changes. Regression testing has been extensively studied for

general purpose software projects. However, when writing regression tests for ML libraries, developers often fail to adequately account for the randomness of the ML algorithms under test and rely on guesswork in selecting various test configurations, such as hyper-parameters for the algorithm under test and the test oracle for algorithm outputs. Such problems primarily occur because most ML algorithms do not come with strict theoretical accuracy specifications that developers could leverage. We next highlight the problems that can occur in such regression tests through an example.

Listing 1.1 presents an example regression test (simplified) from the **ML-Agents** library [46]. **ML-Agents** is an ML library that provides reinforcement learning algorithms. This example a test for the Soft-Actor Critic (SAC) Algorithm [47] – a deep reinforcement learning algorithm. Reinforcement learning algorithms aim to maximize the expected reward of an agent solving a given task (such as playing a game).

In this test, lines 2-12 initialize a simple simulation environment (*SimpleEnvironment*) and the training algorithm (**SAC**). Line 13 performs the training step. Lines 14-18 compute the score (rewards) of the trained agent for the given environment and check if the scores are above the expected value (0.8).

```
1  def test_2d_sac():
2      env = SimpleEnvironment(...)
3      config = TrainerSettings(
4          trainer_type=TrainerType.SAC,
5          hyperparameters=SACSettings(
6              learning_rate=5.0e-3,
7              batch_size=16,
8              ...
9          ),
10         max_steps=10000
11     )
12     trainer = create_trainer(env, config, ...)
13     trainer.start_learning()
14     processed_rewards = [
15         reward_processor(rewards) for rewards in env.final_rewards.values()
16     ]
17     for reward in processed_rewards:
18         assert reward > 0.8
```

Listing 1.1: Example test from **ML-Agents**

Sources of Randomness: The SAC algorithm involves several sources of randomness. Listing 1.2, shows the simplified code snippet for a function `sample_mini_batch`, which is used by the SAC algorithm [48]. In each iteration, the algorithm uses this function to compute gradients for batches randomly sub-sampled (Lines 6-12) from the agent buffer (which contains

```
1 def sample_mini_batch(batch_size, sequence_length):
2     num_seq_to_sample = batch_size // sequence_length
3     mini_batch = AgentBuffer()
4     ...
5     num_sequences_in_buffer = buff_len // sequence_length
6     start_idxes = (
7           np.random.randint(num_sequences_in_buffer, size=num_seq_to_sample)
8           * sequence_length
9        )# Sample random sequence starts
10    for key in self:
11       mb_list = [self[key][i : i + sequence_length] for i in start_idxes]
12       mini_batch[key].set(list(itertools.chain.from_iterable(mb_list)))
13    return mini_batch
```

Listing 1.2: One Source of Randomness (Batching)

```
1 def sample_action(dist):
2     ...
3     continuous_action = dist.sample()
4     return AgentAction(continuous_action)
```

Listing 1.3: Another Source of Randomness (Sampling Action)

traces of the previous steps). Listing 1.3 shows another function `sample_action`, which is used by SAC [49] to sample the next action (which can either be a discrete choice or a continuous value) using a specified distribution (Line 3) for the agent. Due to these random choices of mini-batches and actions at each step, every execution of the test can yield slightly different results (rewards).

Due to the presence of randomness, the test can sometimes pass and fail non-deterministically for the same code even in absence of bugs, thereby making it *flaky*. Flaky tests reduce the reliability of regression testing since they give incorrect signals to the developers and increase the cost of testing, because developers typically need to re-run many tests even if the failures are spurious. While flaky tests have been studied extensively for general software [50, 51, 52], flaky tests in ML are fundamentally different. Because the algorithms under test are randomized, we need to develop principled techniques that can mitigate flakiness by reasoning about the underlying randomness.

One could argue that a simple way to deal with randomness during testing is to set the seed in the random number generators, which will make the execution more deterministic. The developers can then just execute the test for a much smaller number of steps and reduce the run-time. However, setting the seeds may not always be the right choice: they can be brittle in presence of program changes and can hide bugs [53, 54].

In this dissertation, we will present novel techniques that fix flaky tests in ML libraries by adjusting assertion bounds, e.g., lower bound for `reward` on Line 18, Listing 1.1. The key insight of our approach lies in modeling the *tail* of the distribution of values of the assertion variable (`reward`). We then use this distribution to compute a new assertion bound that minimizes the chances of flaky failures to a small value (e.g., $< 0.01\%$). This approach allows us to build an efficient and accurate method for estimating such assertion bounds for stochastic tests in ML libraries.

Developers often employ various strategies to mitigate flakiness. For example, they increase the number of steps for which the algorithm is run (e.g., `max_steps` in Listing 1.1). However, changing such hyper-parameters increases execution time of the tests, and consequently the cost of continuous integration. Alternatively, developers may try to loosen the assertion bound (e.g., replace 0.8 in Listing 1.1, Line 18, with a smaller value) to reduce the chances of flaky failures. However, if not done carefully, loosening bounds can drastically reduce the fault-detection effectiveness of the test. Hence, developers must carefully consider the trade-offs between flakiness, execution cost, and effectiveness while accounting for the randomness of the algorithm under test. Prior to this dissertation, developers had no tools that could solve these problems in an automated and principled manner.

In this dissertation, we present principled and automated techniques that allow developers to manage the trade-offs between flakiness, execution cost, and fault-detection effectiveness of tests. For example, we present techniques for automatically reducing the execution time of stochastic tests in ML libraries, without making them flaky. Here, the key insight is that modestly relaxing the passing probability of a test can often result in both faster and highly reliable execution of the test. We formulate this trade-off between execution time and flakiness as an instance of stochastic optimization over the space of hyper-parameters of the algorithm under test, such as `max_steps` in Listing 1.1. We show that our approach can improve the performance of tests, while retaining the fault-detection effectiveness of the optimized tests.

Related work on flaky tests. Flaky tests in ML libraries are pre-dominantly caused by factors that are more common in ML libraries than in general software. For example, Luo et al. [50] performed the first empirical study of flaky tests in open-source Java projects. Table 1.1 presents the major root causes for flaky tests identified by their study. *Similarly, we conducted an empirical study to understand the nature of flaky tests in ML libraries* [53]. Table 1.2 presents the major causes of flaky tests identified by us in ML libraries. The first six rows present causes that are unique to ML libraries, whereas the *Other** row presents causes that are also typically found in general software. Comparing the two tables, we observe that randomness only contributes to flakiness in 2% of the cases in general software, whereas

in ML libraries randomness is the pre-dominant factor causing 60% of flaky tests. These observations motivate the need to understand flaky tests in ML libraries at a greater depth and also to develop customized approaches to fix them.

Many recent works focus on fixing specific kinds of flaky tests, such as flaky tests due to test-order dependencies [55] or due to unordered collections [56]. These fixes are, however, specific to their root causes and hence inapplicable for fixing flaky tests in ML libraries. Lam et al. [57] proposed mitigating flakiness due to asynchronous waits by automatically adjusting wait times to reduce the chance of tests failing due to waits. They employ a binary search method to determine a wait time that does not cause flaky failures (in 100 runs of the test). However, their approach does not provide any guarantees. In contrast, our method for fixing flaky tests provides statistical guarantees of the obtained bounds (Chapter 5).

Table 1.1: Flaky Tests in Java

Cause	% Flaky Tests
Async Wait	36.8%
Concurrency	15.9%
Test-Order Dependency	9.5%
Resource Leak	5.5%
Network	5.0%
Randomness	2.0%
Floating-point computations	1.5%

Table 1.2: Flaky Tests in ML Libraries

Cause	% Flaky Tests
Algorithmic randomness	60.0%
Floating-point computations	6.7%
Incorrect API Usage	5.3%
Unsynced Seeds	2.6%
Concurrency	2.6%
Hardware	1.3%
Other*	21.3%

Related works on using statistical methods for estimating bounds. Many systems often require estimating guaranteed bounds of performance. For example, in real-time systems, several techniques have been proposed for obtaining guarantees on the performance bounds of the systems, i.e., the best and worst-case task completion time [58, 59]. However, these techniques require a model of the system, which is typically not available for ML algorithms under test.

In anomaly detection, the goal is to identify outliers to expected behavior, e.g., fraudulent credit-card transactions. One popular solution approach is to leverage statistical techniques to build a stochastic model of the true data generating process and classify observations in low probability regions as anomalies [60]. However, such approaches either make strong assumptions on the underlying data distribution (e.g., Gaussian) [61], which is typically unknown for ML algorithms, or they apply non-parametric methods, such as kernel density estimation, which can be computationally expensive. In contrast, in this dissertation we leverage the insight that estimating assertion bounds for stochastic tests in ML libraries only requires reasoning about the tail distribution, which can be done accurately and more efficiently using statistical methods [62].

Our directions. This dissertation presents novel solutions to various aspects of testing Machine Learning-based systems. In particular, this dissertation focuses in two key directions:

1. **Systematic Test Generation and Debugging:** A key challenge in testing ML-based systems is generating tests that can systematically and effectively explore the input space of the algorithms under test. Since the inputs to an ML algorithm can often be complex objects like a Deep Neural Network or a statistical model, generating inputs that are both syntactically and semantically correct is hard. Additionally, ML algorithms do not come with any accuracy specifications, so we also need to design automated test oracles. This dissertation presents systematic test generation and debugging techniques that tackle these challenges, specialized for the Probabilistic Programing domain.

   Probabilistic programming is an emerging programming paradigm that allows developers to write statistical models as simple programs and automate the inference of unknown quantities in such models. Probabilistic programming systems come with a probabilistic programming language to easily express statistical models and implementations of one or more Bayesian inference algorithms. This dissertation presents techniques that systematically generate test inputs for probabilistic programming systems to detect bugs in the implementations of the inference algorithms. This dissertation also presents debugging techniques that allow developers to quickly diagnose the root cause of failures in probabilistic programming systems.

2. **Improving Quality of Developer-Written Regression Tests:** When writing regression tests for ML libraries, developers often fail to adequately account for the randomness of the algorithms under tests and rely on guesswork in selecting various test configurations. As a result, the regression tests often end up being flaky, i.e., they pass or fail non-deterministically for the same code, or become expensive to run or have lower fault-detection effectiveness. This dissertation presents techniques that combine principled statistical techniques, mathematical optimization, and domain knowledge to systematically tackle these challenges. This dissertation presents a new methodology and outlook in how to design such regression tests while accounting for the underlying randomness.

1.3 THESIS STATEMENT

The thesis statement of this dissertation is as follows:

Automated software testing techniques can leverage probabilistic and statistical reasoning to improve the reliability of ML-based systems.

This dissertation focuses on testing of ML libraries. ML libraries are a central component in the ecosystem for building ML-based systems. Hence, the reliability of ML libraries directly impact the reliability of the systems built using them. In this dissertation, we use the term "Machine Learning or ML" as an umbrella term to cover several domains that employ various learning techniques to build models from data, such as deep learning, probabilistic programming, and reinforcement learning. This dissertation explores the following aspects:

1. Systematic and automated test generation for probabilistic programming systems.
2. Efficient techniques for debugging failures in probabilistic programming systems.
3. Automatically detecting and fixing flaky regression tests in ML libraries.
4. Automatically optimizing the execution cost of stochastic regression tests in ML libraries.

1.4 CONTRIBUTIONS

In support of this thesis statement, this dissertation makes the following contributions:

- This dissertation describes *ProbFuzz*: the first systematic and automated testing technique for Probabilistic Programming Systems (PP systems). ProbFuzz allows a developer to specify templates of probabilistic models, from which it generates concrete probabilistic programs and data for testing. ProbFuzz uses language-specific translators to generate these concrete programs, which use the APIs of each PP system. ProbFuzz finds potential bugs by checking the output from running the generated programs against several oracles, including an accuracy checker. ProbFuzz found 51 previously unknown bugs in recent versions of popular PP systems. Developers already accepted 51 bug fixes that we submitted to the three PP systems, and their underlying ML systems, PyTorch and TensorFlow.
- This dissertation describes *Storm*: a novel general framework for reducing probabilistic programs. Given a probabilistic program (with associated data and inference arguments) that causes a failure in a PP system, Storm finds a smaller version of the program, data, and arguments that cause the same failure. Storm leverages both generic code and data transformations from compiler testing and domain-specific, probabilistic transformations. Storm proposes new transformations that reduce the complexity of

statements and expressions, reduce data size, and simplify inference arguments (e.g., the number of iterations of the inference algorithm). We evaluated Storm on 47 programs that caused failures in two popular probabilistic programming systems, Stan and Pyro. Experimental results show Storm's effectiveness. For Stan, the minimized programs have 49% less code, 67% less data, and 96% fewer iterations. For Pyro, the minimized programs have 58% less code, 96% less data, and 99% fewer iterations. Finally, this work shows the benefits of Storm when debugging probabilistic programs.

- This dissertation describes the first large-scale empirical study of usage of seeds in regression tests for ML libraries and presents its implications on testing in ML libraries. Our study identifies 461 tests across 114 ML libraries that failed in absence of seeds and presents their nature and root causes. This study demonstrates that developers often use seeds as a "workaround" to mitigate the effects of randomness of ML algorithms under test, instead of fixing the root cause. This observation motivates the need for alternative and principled techniques to mitigate test flakiness, which are presented in following chapters.

- This dissertation describes *FLEX*: the first tool for automatically fixing flaky tests due to algorithmic randomness in ML algorithms. FLEX fixes tests that use approximate assertions to compare actual and expected values that represent the quality of the outputs of ML algorithms. FLEX systematically identifies the acceptable bound between the actual and expected output quality that also minimizes flakiness. FLEX's technique is based on the Extreme Value Theory [62, 63, 64], a branch of statistics, which estimates the tail distribution of the output values observed from several runs. Based on the tail distribution, FLEX updates the bound used in the test, or selects the number of test re-runs, based on a desired confidence level. FLEX's was evaluated on a corpus of 35 tests collected from the latest versions of 21 ML libraries. Overall, FLEX identifies and proposes fixes for 28 tests.

- This dissertation describes *TERA*: the first automated technique for reducing the cost of regression testing in ML libraries without making the tests more flaky. TERA solves the problem of exploring the trade-off space between execution time of the test and its flakiness as an instance of Stochastic Optimization over the space of algorithm hyper-parameters. TERA presents how to leverage statistical convergence-testing techniques to estimate the level of flakiness of the test for a specific choice of hyper-parameters during optimization. We evaluated TERA on a corpus of 160 tests selected from 15 popular ML libraries. Overall, TERA obtains a geo-mean speedup of 2.23x over the

original tests, for the minimum passing probability threshold of 99%. We also show that the new tests did not reduce fault-detection effectiveness through a mutation study and a study on a set of 12 historical build failures in studied libraries.

1.5 DISSERTATION ORGANIZATION

The rest of this dissertation is organized as follows:

Chapter 2: Testing Probabilistic Programming Systems. This chapter presents our work on systematic test generation technique for probabilistic programming systems: ProbFuzz. It presents a novel intermediate language for probabilistic programs, Storm-IR, that provides a common framework for automated testing of probabilistic programming systems. We leverage Storm-IR to design an automated testing procedure that uses differential testing as test oracle and various statistical methods to compare the probabilistic outputs of each system under test. Our Storm-IR framework heavily motivated our subsequent work on debugging and analyses for probabilistic programs. ProbFuzz found 51 previously unknown bugs in three popular probabilistic programming systems, Pyro, Stan, and Edward.

Chapter 3: Program Reduction for Probabilistic Programming Systems. This chapter presents our work on automated program reduction for probabilistic programs: Storm. Given a probabilistic program that exposes a failure in a probabilistic programming system, Storm finds a minimal version of the program, data, and inference arguments that reproduce the same failure. Storm leverages and improves the Storm-IR framework to introduce domain-specific program transformations for automated program reduction. The reduced programs enable faster debugging of failures.

Chapter 4: Empirical Study of Randomness in Regression Tests in ML Libraries. Developers often choose to alleviate test flakiness in ML libraries by setting seeds in the random number generators used by the code under test. However, this approach commonly serves as a "workaround" rather than an actual solution. Instead, it may be possible to mitigate flakiness and alleviate the negative effects of setting seeds using alternative approaches. To understand the impact of seeds and the feasibility of alternative solutions, this chapter presents the first large-scale empirical study of the usage of seeds and its implications on testing on a corpus of 114 ML libraries. We identify 461 tests in these libraries that fail without seeds and study their nature and root causes. This chapter provides a motivation to investigate alternative ways to mitigate test flakiness and alleviate the negative effects of setting seeds. The following chapters present our advances on this topic.

Chapter 5: Fixing Flaky Tests in ML Libraries. This chapter presents our work on fixing flaky tests in Machine Learning Libraries: FLEX. It presents an approach based

on extreme value theory to determine appropriate assertion bounds for tests that mitigate flakiness down to a desired level. FLEX is a follow-up to our work on flaky test detection, FLASH [53]. FLASH is automated technique that leverages statistical methods for efficient detection flaky tests in ML libraries.

Chapter 6: Optimizing Execution Time of ML Tests. This chapter presents a principled approach for how developers can effectively navigate the trade-off space between the flakiness and the execution cost of stochastic regression tests in ML libraries. It leverages statistical methods to estimate flakiness of the test and mathematical optimization techniques to find optimal test configurations. This work also inspired our subsequent work balancing effectiveness and flakiness of stochastic regression tests in ML libraries [65].

Chapter 7: Conclusions and Future Work. This chapter concludes the dissertation and highlights some key directions for future work to extend the ideas presented throughout this dissertation.

Chapter 2: TESTING PROBABILISTIC PROGRAMMING SYSTEMS

Probabilistic programming systems (PP systems) allow developers to model stochastic phenomena and perform efficient inference on the models. The number and adoption of probabilistic programming systems is growing significantly. However, there is no prior study of bugs in these systems and no methodology for systematically testing PP systems. Yet, testing PP systems is highly non-trivial, especially when they perform approximate inference.

We characterize 118 previously reported bugs in three open-source PP systems—Edward, Pyro and Stan—and propose *ProbFuzz*, an extensible system for testing PP systems. ProbFuzz allows a developer to specify templates of probabilistic models, from which it generates concrete probabilistic programs and data for testing. ProbFuzz uses language-specific translators to generate these concrete programs, which use the APIs of each PP system. ProbFuzz finds potential bugs by checking the output from running the generated programs against several oracles, including an accuracy checker. Using ProbFuzz, we found 67 previously unknown bugs in recent versions of these PP systems. Developers already accepted 51 bug fixes that we submitted to the three PP systems, and their underlying systems, PyTorch and TensorFlow.

2.1 INTRODUCTION

Probabilistic programming has recently emerged as a promising approach for helping programmers to easily implement Bayesian inference problems and automate efficient execution of inference tasks. Both research and industry have proposed various probabilistic programming systems, e.g., Church [3], Stan [66], and many others [67, 68, 69, 70, 71, 72, 73]. These systems automate various parts of common inference tasks and support many approximate inference algorithms from machine learning and statistics, including deterministic variational inference and randomized Markov Chain Monte Carlo (MCMC) simulation. Systems like Edward [74, 75] and Pyro [76] embed probabilistic inference within the general deep learning infrastructures, e.g., PyTorch [77] and TensorFlow [78].

A probabilistic programming system (PP system) typically consists of a language, a compiler, and inference procedures. A programmer writes a program in a probabilistic programming language, which extends a standard programming language by adding constructs for (1) random choice, such as sampling from common distributions, (2) conditioning on data, such as observation statements, and (3) probabilistic queries, such as obtaining a posterior distribution or an expected value of a program variable [4]. Next, a PP system compiles the

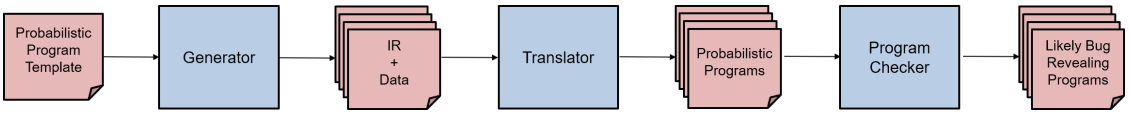


Figure 2.1: ProbFuzz Architecture

probabilistic program to an efficient inference procedure, by adapting well-known inference algorithms. Finally, the programmers run the compiled program on a set of data points to compute the query result.

Probabilistic programming systems provide many benefits to programmers who are non-experts in probability and statistics, but ensuring the correctness of probabilistic programs is notoriously difficult [79, 80]. The inherent uncertainty and complexity of probabilistic inference (which is #P-hard, even with just discrete variables [81]) make most practical inference algorithms numerically intensive and approximate. Therefore, a testing approach for PP systems must account for both numerical errors and errors due to the approximate nature of inference algorithms.

Current approaches for testing PP systems are typically manual and ad-hoc. Although recent research looked into analysis of PP systems [80, 82], none of the proposed approaches can analyze all stages of modern PP systems. Understanding previously known bugs in PP systems and finding effective approaches to improve the systems' reliability remain open research questions.

2.1.1 Bugs in Probabilistic Programming Systems

To motivate the design of tools for systematic testing of PP systems, we characterized the kinds of bugs that are common in existing open-source systems. To the best of our knowledge, this is the first systematic study of bugs in PP systems. We studied three systems: Edward [74, 75, 83], Pyro [76], and Stan [66, 84, 85, 86]. They are written in multiple programming languages, are hosted on GitHub, have been adopted by both industry and researchers, are actively developed, and implement many language features and inference algorithms that are common to most PP systems. In total, we categorized 118 of 856 commits about bugs as being PP systems-related, and describe them in more detail.

Many of the identified bugs required *domain-specific* knowledge to detect, debug, and fix. Moreover, testing PP systems often requires reasoning about result *accuracy* (in contrast to the standard notion of binary pass/fail result correctness). We identified two domain-specific classes of bugs in PP systems: *algorithmic/accuracy bugs* and *dimension bugs*. Algorithmic bugs influence computations of probability expressions and the steps of the inference algorithms, often resulting in decreased result accuracy and are typically hard to identify and fix. Dimension bugs occur when computations do not properly handle the

dimensions of data or allowable ranges of probability distribution parameters. Section 2.3 describes the lessons learned from studying these historical bugs, which we leveraged to design ProbFuzz.

2.1.2 ProbFuzz

We present ProbFuzz, a novel approach and system for systematic testing of PP systems. Figure 3.4 shows the architecture of ProbFuzz. The inputs to ProbFuzz are (1) a specification of the primitive discrete and continuous distributions, (2) the number of programs to generate, and (3) a template that specifies the skeleton of a probabilistic program (model) of interest, written in a high-level probabilistic language notation (IR). ProbFuzz outputs a set of programs that are likely bug-revealing in the PP systems. ProbFuzz has three main components:

- *Generator* completes holes in the template to produce (1) a probabilistic program in an intermediate language and (2) accompanying data necessary to run probabilistic inference. Template completion is a form of fuzzing: Generator produces many programs, with different concrete distributions, distribution parameter values, and data values. To generate programs that are more likely to identify non-trivial bugs, Generator incorporates domain-specific information, e.g., legal connections among distributions, ranges of their parameters, and data properties.

- *Translator* converts the intermediate probabilistic program to a specific API or language of a PP system under test, and selects system-supported inference algorithms. We implemented three versions of Translator, for Edward, Pyro, and Stan.

- *Program Checker* runs the generated programs and determines whether the outputs indicate likely bugs in the PP system on which the programs were run. Program Checker produces a set of likely bug-revealing programs for developers to inspect, and supports checks for standard problems (like crashes or `NaN` errors in the output) and accuracy of inference results.

We designed Generator to be general – it represents probabilistic models in the intermediate first-order probabilistic language, and can target various PP systems. We designed Translator to be flexible and extensible. Our experience is that adding support for a new PP system is relatively easy. Moreover, support for multiple PP system in the Translator enables differential testing as an oracle in the Program Checker. ProbFuzz is available at `https://probfuzz.com`.

ProbFuzz leverages the observation that testing PP systems is conceptually similar to the well-studied field of compiler testing. A prominent approach in compiler testing is *compiler fuzzing* [87, 88, 89, 90, 91, 92, 93, 94, 95, 96], which randomly generates many test programs and checks whether a compiler produces code (or crashes) and whether generated programs are correct, i.e., produce same results as reference programs. Our study of existing bugs and evaluation of ProbFuzz show the importance of (1) domain-specific knowledge about probability distributions and inference algorithms, (2) joint generation of programs and corresponding data to run inference, and (3) reasoning about accuracy. These traits are out of reach for state-of-the-art compiler fuzzing techniques.

2.1.3 Results

We evaluated ProbFuzz on three PP systems: Edward, Pyro, and Stan. Our evaluation shows the effectiveness of ProbFuzz in generating probabilistic programs and data that reveal dimension/boundary-value and algorithmic/accuracy bugs in all three systems. We discovered 67 potential previously unknown bugs in these systems. Further, we used ProbFuzz to target existing bugs in each PP system we characterized in Section 2.3, to see in how many categories per PP system ProbFuzz would have caught a bug. ProbFuzz caught at least one existing bug in 8 of 9 categories that we targeted. Section 2.5 presents quantitative results of ProbFuzz.

As part of our bug discovery and understanding process, we submitted all 67 potential bugs revealed by ProbFuzz to developers of the PP systems. So far, developers have accepted 51, rejected 8, 7 are still pending and 1 was already fixed before we could submit it. The bugs that we found and fixed were not just in Edward, Pyro and Stan, but also in the underlying software infrastructure on which they are built (i.e., PyTorch for Pyro, and TensorFlow for Edward). We describe some of the identified bugs, their fixes, and lessons that we learned in Section 2.6.

2.1.4 Contributions

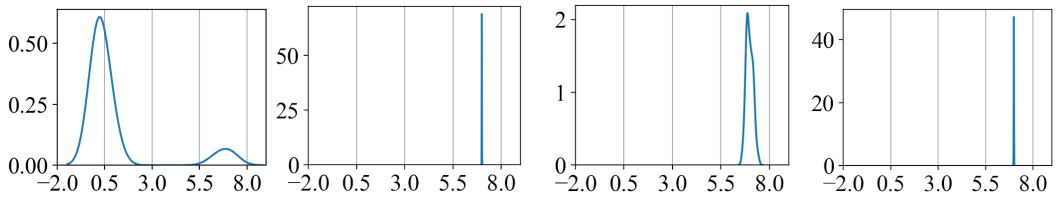
This work makes the following contributions:

⋆ **Concept.** We extend compiler fuzzing to probabilistic programming systems. We generate both probabilistic programs and data to run inference by encoding domain knowledge and reasoning about accuracy of inference results.

⋆ **Bug Characterization.** We present the first study of bugs in PP systems. Our investigation of 118 previously fixed bugs in three open-source PP systems showed that these bugs require domain knowledge to find and fix, and to reason about accuracy.

```
1 x̄ = [1.0, 2.0, ...]
2 ȳ = [7.0, 14.0, ...]
3 w = Gamma(97.5, 86.2)
4 p = Beta(44.0, 44.0)
5 observe(
6     Normal(w · x̄, p), ȳ)
7 posterior(w)
```

Figure 2.2: Probabilistic Program



(a) Stan Result (b) Edward Result (c) Pyro Result (d) BUGS Result

Figure 2.3: Example Program and the Posterior Distributions Computed by Various Systems

⋆ **Methodology and System.** We propose ProbFuzz, a novel general approach for systematically testing PP systems. Our current implementation of ProbFuzz works for three open-source PP systems and is extensible.

⋆ **Evaluation.** We evaluated ProbFuzz on both historical and recent versions of Edward, Pyro and Stan. ProbFuzz found bugs in each category of previously reported bugs. We also found and reported 67 previously unknown bugs by running ProbFuzz on recent versions of the PP systems.

2.2 ILLUSTRATIVE EXAMPLE

Figure 2.2 shows an illustrative example of a potentially bug-revealing probabilistic program generated by ProbFuzz. The program, shown in ProbFuzz's intermediate language, defines two data-sets of constants, $\overline{x}$ and $\overline{y}$. Each $\overline{y}_i$ is seven times the value of $\overline{x}_i$. The program first assigns prior distributions to the variables w and p. Then, it conditions the linear model $w \cdot \overline{x} + p$ on the observations of $\overline{y}$. The probabilistic query on line 7 seeks the posterior distribution w.

Probabilistic inference is a procedure for computing the change in the distribution of variables based on the observations of data. Most inference algorithms today are approximate, with the two dominant approaches being *Markov Chain Monte Carlo* simulation, which re-executes the computation with many random samples (and is implemented in, e.g., Stan and Edward) and *variational inference*, which approximates the posterior distribution deterministically, by substituting it with computationally simpler distributions (and is

implemented in, e.g., Edward and Pyro).

Figures 2.3a-2.3d show the posterior distributions computed by Stan, Edward, Pyro, and another probabilistic inference system called BUGS (which is a precursor of Stan, and shares most of its syntax). The X-axis presents the numerical values and the Y-axis presents its probability density function. Given the data $\overline{x}$ and $\overline{y}$, we expect the mean of the posterior of w to be equal to 7.0. The posterior distributions computed by three systems are similar, and centered at 7.0. However, Stan's distribution has a different shape, and its mean is close to 1.0. We discuss the reasons behind this accuracy problem in Section 2.6.2.

ProbFuzz generates the program in Figure 2.2, and many similar programs, with different prior distributions, distribution parameters, and data. ProbFuzz then compiles the programs down to each PP system, generating specialized API calls or DSL programs. The translation is non-trivial, and cumbersome for a human, but can be easily specified in ProbFuzz. Next, ProbFuzz runs generated programs, automatically compares the output from different PP systems, and computes accuracy metrics (Section 2.4.4). Finally, a developer can inspect ProbFuzz results and investigate any potential bugs. We discuss ProbFuzz in Section 2.4.

2.3 BUG CHARACTERIZATION STUDY

We characterized existing bugs in three open-source PP systems: Stan [66, 84, 85, 86], Edward [74, 75, 83] and Pyro [76]. Table 2.1 shows some statistics about the PP systems. The three PP systems support various approximate probabilistic inference algorithms.

Methodology. We manually searched for bug fixes among commits in the GitHub repositories of the PP systems in our study. We use commits to get a larger data set than we could get when starting from GitHub issues [97, 98]. Given the active development of these PP systems, many bugs are fixed without first being reported as "issues", and most closed issues involve one or more commits.

We obtained all commits in the three PP systems that contained the keywords, `bug|infere`

Table 2.1: Project Statistics

	Edward	**Pyro**	**Stan**
First commit date	Feb 10 '16	Jun 15 '17	Sep 30 '11
No. of contributors	74	26	61
No. of commits	1780	853	13083
Latest commit studied	992ce08	8db8972	14981a3
Lines of code	12035	11609	57770
Prog. language	Python	Python	C++
Infrastructure	Tensorflow	PyTorch	Own

Table 2.2: Breakdown of Commits

Category	**Edward**	**Pyro**	**Stan**	$\sum$
Algorithmic/accuracy	9	10	16	35
Dimension/boundary	11	14	13	38
Numerical	1	1	17	19
Language/translation	5	7	14	26
$\sum$	26	32	60	118

`nce|error|fix|nan|exception|overflow|underflow|infinity|infinite|precision|unstable|instability|ringing|unbounded|roundoff|truncation|rounding|diverge|cancellation|cancel|accuracy|accurate`. This resulted in 1837 commits. We then filtered out commits that are not specific to the domain of PP systems or probabilistic inference, and could occur in any software domain. First, we filtered out commits containing the following keywords: `typo|docstring|notes|example|examples|tutorial|print|doc|Document|messaging|test|messages|manual|doxygen|cpplint|Jenkins|submodule|header`. Next, we split the remaining 856 commits between two student coauthors, each of whom read descriptions and reasoned about modified code. Each coauthor marked a commit as an inference-related code fix, general code fix, a refactoring, or a duplicate. We filtered out refactoring, duplicates (e.g., covered by incremental commits fixing the same bug or related commits from multiple branches), merge commits with many files changed, and commits that changed only non-source files.

We were left with 455 commits that fix code, out of which our manual inspection identified 118 commits that are directly related to the domain of probabilistic inference. The remaining are general coding problems e.g., I/O errors, API misuses, and documentation problems. Two coauthors inspected these 455 commits. They compared notes and classified bugs as inference-related only if they agreed on the final classification, therefore making a conservative determination about the domain-specific nature of each bug. Similar to a previous work on analyzing numerical bugs [99], we put inference-related bugs into four categories. Our bug categories are algorithmic/accuracy, dimension/boundary-values, numerical, and language/translation. We made a second pass through the 118 bugs that satisfy the selection criteria and categorized them based on error sources and bug manifestations. When possible, we matched each commit to its related GitHub issue.

2.3.1 Characterizing Bugs in PP Systems

Table 2.2 shows the distribution of the categorized commits. Column "Category" shows category names. The second to fourth columns show the number of commits per category in each PP system. Finally, column $\sum$ presents the sum of the commits in each bug category.

The database of inference-related bugs is available at `https://probfuzz.com/db`.

Algorithmic/accuracy bugs. This category contains bugs due to incorrect implementation of inference algorithms and other related bugs in the implementations of probability distributions and statistical procedures. They manifest as inaccurate, although plausible (and therefore hard to catch) results of inference. These bugs affected a variety of inference algorithms and implementations of probability distributions in all three PP systems. In Edward, the bugs affected three inference algorithms and two built-in distributions (Bernoulli and Uniform). In Pyro, the bugs affected three inference algorithms and the Cauchy distribution. In Stan, the bugs affected two inference algorithms, one distribution (Bernoulli Logit) and two auxiliary functions.

These bugs can be further subdivided into logical errors, mathematical errors, and one regression error. Examples of logical bugs include re-normalizing already normalized data [100], "double-counting" the values of specific variables [101], and using only the first element instead of a whole collection to fill a tensor [102]. Examples of mathematical errors include incomplete formulae (e.g., missing terms [103, 104]) and wrong formulae (e.g., [105]). Finally, a regression in Stan led to lower statistical efficiency [106].

Dimension/boundary-value bugs. These bugs occur when functions do not properly handle the dimensions of input data (a scalar, vector, matrix, or cube), the ranges of input data, and the ranges of distribution parameters. They manifest as exceptions or special numerical values, e.g., `NaN` or `Inf`, in the output (in the case of boundary-value bugs). The examples of dimension bugs include those where the functions assumed a particular dimension of input data (e.g., scalars [107]) and crash if data with different dimension is passed as input, or assumed a wrong dimension of output which caused crashes in the function's clients (e.g., [108]). One bug resulted from using only one ordering of a list (a vector) to compute entropy, instead of using all possible orderings (a matrix) [109].

Missing boundary condition checks often happen in implementations of various probability distributions, e.g., not checking for boundary values of a parameter leading to `NaN` [110]. Such bugs typically manifest substantially late during inference, e.g., computing `log` of zero resulting in `NaN` [111]. We also observed some off-by-one errors (e.g., in [112, 113]), where `if` conditions used $<$ instead of $\leq$.

General numerical bugs. These bugs are found in general mathematical functions, and may manifest as an inaccurate result or a special value (`NaN` or `Inf`). Most of these bugs are in Stan, which implements its own mathematical back-end, in contrast to Edward and Pyro, which use external back-ends (TensorFlow and PyTorch, respectively). Example numerical bugs that we identified include improper handling of `Inf` (e.g., [114, 115]) or `NaN` (including when these special values propagate to the output [116]), initializing `Integer` values to `NaN`,

overflow errors, and convergence bugs.

Language/translation bugs. These bugs occur due to wrong use of features in the programming language in which the PP system is written. They can manifest as failed builds, runtime errors, or wrong results. These can be errors in the interface (e.g., [117], returning a `real` instead of an array as expected from the API specification), errors in the back-end or changes in their implementations (e.g., [105]), errors that break compilation or error reporting (e.g., [118]), and errors in using functionality. One functionality usage error involved calling a stateful inference function, making different runs of the same probabilistic program producing widely different results [119].

2.3.2 Discussion

We highlight several important observations from our characterization study, which motivate our approach for testing PP systems:

Observation 1: Domain knowledge is required to detect, analyze, and fix bugs. Most of the inspected Algorithmic/accuracy and Dimension/boundary-value bugs, and some Numerical bugs require knowledge of theory of probability or inference. Bugs in the Dimension/boundary-value category are similar to general bugs that occur when one does not satisfy the specification of a method. However, without specification-related assertions (which require domain-specific knowledge, and are tedious to write) in the code, such bugs occur in the PP systems, resulting in `NaN` or silent errors.

Observation 2: Algorithmic bugs require detailed reasoning about accuracy. For many of the inference and accuracy bugs, the developers report (in)accuracy of the results and compare the results either to known (expected) values or against another tool (e.g., Edward or Pyro against Stan). For algorithmic errors, existing numerical analyses [120, 121] are typically not applicable. Identifying errors and their causes requires probabilistic reasoning, detailed error reports and discussions with PP system developers in order to diagnose the error (e.g., [122]).

Observation 3: Testing PP systems requires careful generation of both programs and valid data. Reproducing many of the bugs that we manually inspected required both a probabilistic program and the data to run it on. The GitHub issues related to the commits that we inspected had *both* programs (or program fragments) and data necessary to reproduce the bug. Such data is sampled from probability distributions and is required for setting up priors and posteriors, distribution parameters, and as inputs for inference. This is different from compiler testing [87, 88, 89, 90, 91, 92, 93], where it is sufficient to simply generate programs that take no inputs and encode arbitrary scalar values of variables.

Observation 4: Many errors are revealed by small programs. Most GitHub issues related to the commits that we inspected had small reproducible programs. The observation that many bugs can be found by small programs is well-known [123], and has been used extensively in conventional testing. While standard compiler testing (e.g., CSmith [87]) often generates large programs to maximize bug-finding capability, small programs seem sufficient for successful detection and debugging in the PP system domain.

2.4 PROBFUZZ

ProbFuzz takes as inputs the template of the probabilistic model, the number of programs to generate and the systems to test. The developer writes the templates of probabilistic models in an intermediate probabilistic language with holes, which represent missing distributions, parameters, or data (Section 2.4.1).

Figure 2.4 presents the pseudo code of the ProbFuzz algorithm. The Generator generates probabilistic models by completing holes in the template with concrete distributions, parameters and data (Section 2.4.2), resulting in a program in an intermediate language. The Translator then translates the probabilistic program from the intermediate language into a program that uses the API of the target PP system (Section 2.4.3). Next, ProbFuzz runs the programs, collects output, and its Program Checker computes metrics and checks for symptoms that may reveal potentially buggy programs (Section 2.4.4). Finally, ProbFuzz reports any warnings issued by the Program Checker to the developer.

2.4.1 Template and Intermediate Language

ProbFuzz represents the templates and the generated programs in an intermediate language (IR). Figure 3.3 presents the syntax of the IR language of ProbFuzz. The key aspect of the template is *a hole*, denoted as "??". It represents a missing distribution or parameter. The distributions and parameters are completed with concrete values (from respective sets *Dists* and *Consts*) by replacing the hole.

A template consists of four sections, which specify data, prior distributions, model that relates posterior and prior distributions, and the query. The data section presents the input and the output data set(s). A data vector is a typed (multidimensional) array, which is instantiated by ProbFuzz, or a specific list of numerical constants. The *Prior* section specifies the prior distributions of the program variables. A prior distribution can be an instance of a distribution or a hole. Similarly, one or more parameters of the distribution can be either expressions or holes. The expressions are typical, with arithmetic and comparison operators.

INPUTS: Program count n, Template t,
PP systems under test S
OUTPUT: Likely bug-revealing programs report P

function ProbFuzz(n, t, S)
  $P \leftarrow \varnothing$
  for $i = 1$ to n **do**
    $prog_{IR}, data \leftarrow Generator(t)$
    $Results \leftarrow \varnothing$
    for $s \in S$ **do**
      $prog_s \leftarrow Translator_s(prog_{IR}, data)$
      $status_s, out_s \leftarrow ExecuteProgram(s, prog_s, data)$
      $Results \leftarrow Results \cup \{(status_s, out_s)\}$
    end for
    $warnings \leftarrow ProgramChecker(Results)$
    if $warnings \neq None$ **then**
      $P \leftarrow P \cup \{warnings\}$
    end if
  end for
  return P
end function

Figure 2.4: ProbFuzz Algorithm

The language is similar to the loop-free fragment of the Prob language from [4].

The *Model* section conditions the random variables to the specific observations. The `observe` clause states that the observations of the model specified as the first parameter are found in the vector denoted as the second parameter (as is a standard interpretation in most probabilistic languages). The models can also be composed using conditionals. Finally, the *Query* instructs the probabilistic language to return the marginal posterior distributions for the specified variables, or their expected values.

Examples. Figure 2.5b presents a template from our experiments and Figure 2.5c presents an example program that has the holes completed. The template is for a linear regression model, which has two sets of observations x and y (both are one-dimensional vectors of length 10). The prior parameters are weight w, bias b, and the noise p, with unknown distributions. The template conditions an unspecified distribution with two parameters (the first is the linear expression $w \cdot \overline{x} + b$, the second is p) on the data from the vector $\overline{y}$.

Distribution Specification. For each distribution, ProbFuzz specifies its properties, including the names and ranges of parameters and the range of the distribution support. Knowing the properties of the distributions allows ProbFuzz to complete the template with the concrete values of parameters.

To illustrate, the specification of the Normal distribution is:

```
"name" : "Normal",
```

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

Type ::= `Int` | `Float` | `Range<`c, c`>` | Type[c]
Data ::= x `:` Type | x `:=` [c^+] | x `:=` Expr
Expr ::= c | x | Expr *aop* Expr | Expr *bop* Expr
Param ::= `??` | Expr
Prior ::= x `:= ??` | x `:=` Dist(Param$^+$)
Model ::= `observe(`Dist(Expr$^+$), x `)`
| `observe(`??(Expr$^+$), x `)` | x = Expr
| `if (`Expr`) then` Model `else` Model
Query ::= `posterior`(x^+) | `expectation`(x^+)
Template ::= Data$^+$ Prior$^+$ Model$^+$ Query

(a) Grammar for Probabilistic Program Templates

$\overline{x}$: `Float`[10]
$\overline{y} := c_1 * \overline{x} + c_2$
$w =$??
$b =$??
$p =$??
`observe`(??$(w \cdot \overline{x} + b, p), \overline{y})$
`posterior`(w, b, p)

(b) Linear Regression Template

$\overline{x} := [1.0, ...]$
$\overline{y} := [2.0, ...]$
$w = \mathit{Gamma}(0.3, 5.2)$
$b = \mathit{Normal}(0.3, 2.1)$
$p = \mathit{Exponential}(1.2)$
`observe`$(\mathit{Normal}(w \cdot \overline{x} + b, p), \overline{y})$
`posterior`(w, b, p)

(c) Linear Regression Example

Figure 2.5: Grammar and Example for ProbFuzz Input Templates

```
"type" : "Continuous",
"support" : "Float"
"args" : [ { "name" : "mu", "type" : "float"},
           { "name" : "sigma", "type" : "float+" } ],
```

It specifies that the distribution is continuous and its support (the range of values that can be sampled from the distribution) is not constrained. It has two parameters, the mean `mu` is an arbitrary floating-point value, while the standard deviation `sigma` must be positive. The support and parameters of the distribution can be bounded. For instance, in the case of Gamma distribution, the support is only positive real numbers, and in the case of Bernoulli, the support is {`0,1`}.

2.4.2 Generator

The Generator generates a concrete program and data from the provided template. A concrete program consists of complete IR and data. In a concrete program, all "??" symbols have been replaced by the corresponding distribution expressions or constant expressions (as in Figure 2.5c). The user-defined program templates plus domain knowledge about distributions and data ranges enable Generator to achieve more targeted fuzzing.

The Generator has two components, the *distribution selector*, which matches the distribution expressions with holes ("??") in the template and the *data selector*, which produces the

concrete values of the parameters of the distributions and computes the values of the data points. For each generated program, the Generator performs the following steps:

- **Complete the distribution of the model.** For the model expression, the distribution selector finds all the distributions that can match the pattern (e.g., have two parameters) and uniformly at random selects one of those distributions to fill in the hole. Once fixed, this distribution provides the legal values for the data to generate (based on the distribution support) and the constraints on the parameters. This bounds the set of allowed distributions of the priors in the template. For instance, if we select the Normal distribution for the linear regression template (Figure 2.5b), the model constrains the distribution of the variance p to have positive support.

- **Complete the distributions of the priors.** Based on the constraints from the model, the distribution selector randomly selects a distribution whose support satisfies the range of values admissible by the model's parameter. To propagate the information about distributions, we implement a simple dependence analysis with interval analysis to keep track of the ranges. For instance, in Figure 2.5 the distribution selector may choose Exponential as the distribution of the prior for p, but not Normal (since its support is all floating-point values, but p can have only positive values).

- **Complete the distribution parameters.** Data selector picks the numerical values of the parameters of the distributions with holes using a method that randomly chooses between two strategies. The first strategy randomly selects a value within the range of the parameter, as denoted in the distribution specification. A developer may express preference for larger or smaller values to be inserted here. The second strategy randomly picks values that are close to the boundary values of the parameter ranges; these values may be either legal or illegal and can stress-test the sensitivity of PP systems to boundary conditions and numerical instabilities. The developer can provide a probability that prefers one strategy over the other. For instance, in Figure 2.5c, data selector picks the values 0.3 and 5.2 as the parameters of the Gamma distribution in the prior of w. Similarly, it could try generating programs where the second parameter of Gamma (which should be positive) is 0.0 or -1.0 to test the capability of the PP system to identify wrong values.

- **Generate the inputs/outputs.** Data selector uses the input range and formulas provided by the developer to compute expected outputs. It then randomly generates the desired number of elements in the input vectors and computes the values in the output vectors.

2.4.3 Translator

Translator produces a legal program in the language of the target PP system. The inputs to the Translator are the concrete program and data produced by Generator. Each PP system has its own Translator. In addition, Translator takes a configuration file with the list of inference algorithms and a mapping of distributions to corresponding PP system-specific API calls.

Translator in Edward. First, the Inference Selector chooses an inference algorithm that the PP system supports, based on the concrete specification. Second, the Translator replaces distribution names in the input programs with the corresponding API call in Edward, and creates one AST node each for the input data ($\overline{x}$), the model in the program, and the selected inference. Third, several AST nodes are created for the following: (1) one node for the posteriors or each prior, depending on the inference algorithm to be run, (2) a node for a placeholder for $\overline{x}$, and (3) (optional) one node for the proposals for each prior, which is needed for some inference algorithms, e.g., the Metropolis-Hastings (MH) sampling algorithm. Fourth, a `dict` node is created which connects the node for each prior to its respective proposal and posterior nodes, and a `dict` node is created which connects nodes for the data placeholder and the output data, $\overline{y}$. Fifth, the `dict` nodes from the last step are merged with the node for inference. Sixth, the data node, the model node and the inference node are combined together as the final AST. Finally, this AST is converted into a Python program.

Translator in Pyro. The first two steps in the Translator are the same as for Edward: select inference algorithm, replace distributions with corresponding API calls and make AST nodes for $\overline{x}$, the model and the selected inference. The third step is to create a function node for a *Pyro model*, a combination of posterior nodes for each prior which are then connected to the data node. Then a function node for a *Pyro guide* is created with a posterior node for each prior. Next, if the selected inference algorithm is a variational algorithm, an optimization algorithm is chosen together with its parameters based on the concrete specification, and a node is created. Finally, a node for running the inference is created. The generated AST is converted to a Python program.

Translator in Stan. Stan's Translator does not create ASTs. Rather, each model is translated line by line to Stan code stored in `model.stan` file, with data stored in `data.json` file. Finally, a file, `driver.py` is generated and used to run the Stan model.

2.4.4 Program Checker

The task of the Program Checker is to decide whether output from running the generated programs may be indicative of bugs in the PP system on which the program was run. For Edward and Pyro, the generated Python programs are run directly. The `driver.py` script is

run for Stan. Program Checker performs a battery of checks, inspired by the bugs from our characterization study (Section 2.3):

- **Crash checks**: they find problems with unexpected termination or assertion failures. Crash checks will output programs which crash as likely bug revealing, since all programs generated by ProbFuzz are syntactically and semantically valid.
- `NaN` **and overflow checks:** they will report programs that neither crash nor produce exceptions, but contain `NaN` as output values; as observed in Section 2.3, they are often related to numerical and boundary checking problems. Programs which produce `NaN` as output values are potentially bug revealing because it means that the PP system allowed invalid computations to "succeed", instead of warning the developers.
- **Performance checks:** they report if one PP system converges much slower than other PP system.
- **Differential testing with exact result:** these checkers aim to identify accuracy bugs by comparing the results of approximate inference with the exact result. The exact result can be obtained in two ways: (1) using optional data generators , or (2) using exact inference engine. For exact inference, we translate programs to PSI [124]. Exact inference (when it scales) removes approximation and numerical errors, modulo bugs in the exact inference tool. This approach works when the generated programs are small.
- **Differential testing with approximate results:** these checkers aim to identify accuracy bugs by comparing the differences in the results produced by (1) different tools and (2) different algorithms within a single tool or even different versions of the same algorithm (e.g., [122]), and (3) different interfaces to the same inference algorithm. Result comparison across tools or algorithms is useful for accuracy and numerical bugs. Comparisons across different interfaces of the same PP system (e.g., RStan, PyStan) can primarily help find language/translation bugs. The Program Checker issues a warning about a program from which the results of one approximate-inference PP system differs significantly from all other approximate-inference PP systems and the other systems produce similar outputs, or if the outputs of all approximate inference differ from the expected output.

Accuracy Comparisons. Analysis of accuracy is a key challenge in testing PP systems. The computations have various sources of noise: some inference algorithms are randomized (e.g., MCMC), while others make algorithmic approximations (e.g., variational inference). In both cases, there may be rounding errors or overflows.

To quantify the magnitude of errors, ProbFuzz allows a developer to specify custom comparison metrics. Here, we compute an accuracy metric based on relative error of the

mean. *Symmetric Mean Absolute Percentage Error* [125] computes the distance between the means of the posterior distributions computed by two systems (or comparing the result from one system to the exact result). It is computed as:

$$SMAPE(x_1, \ldots, x_n, y_1, \ldots, y_n) = \frac{1}{n} \sum_{i=1}^{n} \frac{|x_i - y_i|}{|x_i| + |y_i|} \tag{2.1}$$

The arguments $x_1, \ldots, x_n$ are the means produced by the first system and $y_1, \ldots, y_n$ are the means produced by the second system. In contrast to the usual relative error, which divides the difference by the value from one of the systems, SMAPE does not prefer the result of any of the systems, and is always guaranteed to produce a result in the range $[0, 1]$.

A program may have an accuracy bug if the value of the metric is above a threshold (which effectively acts as a knob for how many programs to inspect). If so, ProbFuzz reports the generated program as revealing a potential accuracy bug. When more than 2 systems are involved, we do a pairwise comparison. If only one of the PP systems shows a significant deviation from the others, ProbFuzz reports that system as likely faulty.

2.5 QUANTITATIVE EVALUATION

We describe the research questions that we answer, our experimental setup and the quantitative aspects of the results in this chapter. We answer the following research questions:

RQ1 How many *new* bugs per category does ProbFuzz find?

RQ2 How many categories of *existing* bugs does ProbFuzz find?

RQ3 How sensitive is the accuracy metric to the threshold choice?

RQ4 How does ProbFuzz compare with conventional fuzzing?

Experimental Setup. For our experiments, we used four templates. We discussed *linear regression* template in Section 2.4.1. Other templates include *simple posterior template*, which samples from a single distribution, with a prior for each of its parameters and conditions on data, *conditional template*, which chooses between two models based on the if expression, and *multiple linear regression* template with a weight vector for the prior instead of scalar as in linear regression and conditioned on 2-dimensional data sets. We also varied the data vector sizes. We generated 1000 programs per template for each tool. We group the programs based on the determination that Program Checker makes, and then randomly sample a subset of programs in each class for manual inspection. To find performance bugs,

we randomly sampled for manual inspection the programs that did not run to completion in the default time-out limit of `3 mins`. For accuracy bugs we used the accuracy metric discussed in Section 2.4.4 to select wrong programs to manually inspect. The threshold for `SMAPE` that we used in selecting the programs for our manual inspection was `0.1`. We ran all experiments on an Intel Xeon 3.60GHZ machine with 6 cores and 32GB RAM.

2.5.1 RQ1: New Bugs Discovered by ProbFuzz

Table 2.3 shows the number per category of the new bugs found during our evaluation of ProbFuzz. Columns (except $\sum$) are the PP systems in our study, while the rows (except $\sum$) are the various categories for which found some bugs that we found. Bug categories were described in Section 2.3. We counted as bugs either as the number of distinct code locations where we made a fix in pull requests, or one bug for each issue that we submitted to the developers without a corresponding pull request. Note that, by counting each (yet-to-be-fixed) submitted issue as one bug, we are under counting the number of bugs in the code, and the actual number of bugs that ProbFuzz found in our experiments is likely higher.

We submitted 15 issues (each one counts as one bug), and 7 pull requests which fixed 51 bugs in the code. The results show that the dimension/boundary-value bugs are the most common among the bugs that we found. We provide more details in Section 2.6.1 about how prone the PP systems are to dimension/boundary-value bugs. Among the PP systems, we found the least number of bugs in Stan, followed by Edward and then Pyro. Interestingly, this matches the maturity of the PP systems. We also discuss in Section 2.6.1 one step that Stan developers have taken over the years to reduce the amount of bugs in this category.

One key benefit that ProbFuzz provides in the testing of PP system is the ability to find accuracy bugs, and not just bugs that lead to crashes or invalid values (e.g., `NaN` or `Inf`) in the output. Accuracy bugs are much more tricky to find and debug; coming up with oracles that catch them is quite involved and requires domain knowledge. As shown in Table 2.3, we found 5 potential accuracy bugs in all three PP systems during our manual inspection.

We reported all the bugs in Table 2.3 to the developers of each PP system. So far, developers have accepted 51, rejected 8, 7 are still pending and 1 was already fixed before we could submit it; 30 accepted bugs were in a single pull request to PyTorch.

2.5.2 RQ2: Old Bugs Rediscovered by ProbFuzz

This experiment checks whether ProbFuzz can catch a variety of previously fixed bugs that we identified during our characterization study (Section 2.3). For each PP system, we

Table 2.3: New Bugs per Category Discovered by ProbFuzz

Category	**Edward**	**Pyro**	**Stan**	$\sum$
Algorithmic/accuracy	2	1	2	5
Dimension/boundary	13	41	0	54
Numerical	0	0	3	3
Language/translation	1	3	1	5
$\sum$	16	45	6	67

attempted to reproduce at least one bug per category, such that they cover all categories of interest (Algorithmic/accuracy, Dimension/boundary-value, and Numerical). We did not target Language/Translation bugs, which are specific to each PP system and targeting them requires more involved back-ends. We first checked if these bugs may be reproduced by re-running the tests that failed due to the bug or programs in the corresponding GitHub issue. We stopped if we could no longer run the tests/programs. We did not try to reproduce bugs that cannot be exercised by our four templates. Since some older versions of the PP systems use different syntax and API to specify models or have since undergone major changes, we had to create four additional versions of Translator (for bugs [101, 107, 116, 126]). In addition, we found the versions of the infrastructure (PyTorch and TensorFlow) which were in use in the older versions. For accuracy and numerical bugs, we manually reasoned whether the difference was caused by the bug.

Table 2.4 shows the numbers and links to bugs that we successfully reproduced with ProbFuzz. For each of these bugs, ProbFuzz generated a program and the data to exercise it. Each cell contains the bug count in each category per PP system. In addition, each cell contains the exact reference to the commit with the bug fix.

The results show that ProbFuzz successfully found bugs in eight out of nine categories of interest. Out of these bugs, six ([107, 126, 127, 128, 129, 130]) were found using the simple posterior template, three using the linear regression template [101, 116, 131] and one using multiple linear regression template [132]. Overall, these results demonstrate that ProbFuzz could have caught a variety of existing bugs, had it been available prior to the discovery of those bugs. Comparison of Tables 2.3 and 2.4 shows that ProbFuzz was able to reproduce existing bugs in categories where we did not find any new bug on recent versions of the PP systems (e.g, Stan-Dimension/boundary). Also, ProbFuzz reproduced the only previously known numerical bugs in Edward and Pyro from our characterization study.

Table 2.4: Old Bugs per Category Rediscovered by ProbFuzz

Category	**Edward**	**Pyro**	**Stan**	$\sum$
Algorithmic/accuracy	1 [131]	1 [101]	0	2
Dimension/boundary	1 [107]	1 [128]	2 [129, 132]	4
Numerical	1 [126]	1 [127]	2 [116, 130]	4
Language/translation	n/a	n/a	n/a	n/a
$\sum$	3	3	4	10

2.5.3 RQ3: Sensitivity of Accuracy Threshold

The number of programs to inspect depends on the threshold set for the accuracy metric. Figure 2.6 presents the sensitivity of the number of programs reported to potentially reveal accuracy problems as a function of the bound on the `SMAPE` metric for the linear regression template. The X-axis presents the threshold. The Y-axis presents the fraction of the programs whose accuracy metric (compared to the reference solution) is above the threshold. For the computation, we removed (1) the programs that crashed, (2) the programs that resulted in `NaN`, and (3) the programs that timed out.

The results show that the threshold value can serve as a knob for the fraction of the programs to return. For instance, if the threshold is 0.8, then the number of programs with large accuracy loss is less than 10% for Pyro and Stan, and around 14% for Edward. Stan shows an interesting trend of having many programs that have small accuracy loss of the mean, while Edward and Pyro have more programs that have larger accuracy differences.

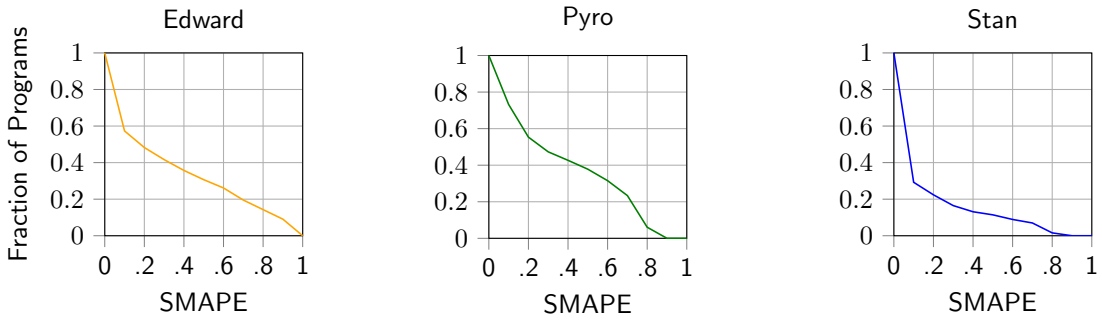


Figure 2.6: Sensitivity of Accuracy Metric

2.5.4 RQ4: Benefit of Domain Knowledge

We compared our results with an "uninformed" fuzzer that does not utilize domain knowledge about distributions and legal ranges. Table 2.5 shows the detailed comparison for 1000 generated programs per tool per template. Each cell contains the percentage of

generated programs that produced results without crashing, numerical errors, or timeout for Uninformed ('U') or Informed ('I') fuzzer. On average, less than 21% (Stan 6.35%, Edward 49.07%, Pyro 5.82%) of programs generated by the uninformed fuzzer produce useful results, compared to over 84% (Stan 89.35%, Edward 80.3%, Pyro 83.2%) with ProbFuzz. As such, uninformed approach can be fit for boundary-condition bugs, e.g., when a system fails to recognize a program with wrong values, but it will not be efficient for bugs that can be revealed by only well-formed probabilistic programs.

Table 2.5: Comparison of Informed vs Uninformed Fuzzing

Template	**Stan**		**Edward**		**Pyro**	
	U	I	U	I	U	I
Simple	16.6	93.4	52.5	80.2	14.0	87.2
LR	6.6	94.3	43.3	80.7	5.2	79.7
MLR	1.2	79.6	43.8	81.2	4.1	83.3
Conditional	1.0	99.0	56.7	79.1	0.0	82.6

2.6 QUALITATIVE ANALYSIS

During our development and evaluation of ProbFuzz, we encountered several potential bugs in PP systems for which we created fixes and submitted pull requests to the developers. We made fixes in Edward, Pyro, Stan, and also contributed patches to the underlying frameworks, PyTorch and TensorFlow. We present interesting cases, lessons learned, and developer responses.

2.6.1 Dimension/Boundary Bugs Are Common

Dimension/boundary-value bugs accounted for 54 previously unknown bugs, and 38 bugs in our characterization study. In Pyro, we found 41 bugs in this category. One of these bugs in Pyro would lead to a crash whenever the input data is of size `1`; another bug caused an overflow in the Adam Optimizer. We also found similar, previously unknown Dimension/boundary-value bugs in Edward: four bugs were also due to failure to check that parameter values are in the correct range. Interestingly, ProbFuzz did not find any previously unknown dimension/boundary-value bugs in Stan, despite the fact that our characterization study revealed eight dimension/boundary-value bugs that were previously reported in Stan. We attributed this to Stan's relative maturity, compared with Pyro and Edward. Indeed,

since March 2014, Stan developers have added a milestone to every major release with the title, *"make sure all distributions throw exceptions at undefined boundaries"* [133].

Lesson Learned: The similarity of dimension/boundary-value bugs found across PP systems suggests that that these bugs are commonly introduced by the developers of the PP systems that we studied. Going forward, developers should continuously test their probabilistic codes for this kind of problems. Automated testing, such as ProbFuzz, can be quite effective for these problems.

2.6.2 Accuracy Problems Are Hard to Analyze

Accuracy problems can be difficult to identify and debug, and they can have serious consequences. Section 6.2 presented one such problem. While this problem was present in Stan, it is interesting that Stan's precursor, BUGS, which shares most of its modeling syntax and principles, computes the correct result. For a non-expert, it is often hard to figure out the reasons behind this discrepancy. Next, we provide some insights into how we analyzed this particular case.

We observed that the error is reproduced for any value of the parameters of Beta distribution, which is the prior for p. Stan produced warning messages stating that the random variable used for computing the beta logpdf in a particular step is negative but was expected to be positive. The Stan manual describes such messages as follows: *Warning messages arise in circumstances from which the underlying program can continue to operate* [134]. Stan often converges to the correct result despite such warnings, but in this case, it did not. When such warnings persist, Stan developers suggest "investigating the arithmetic stability of the Stan program" [134].

One way to address the accuracy problem is to change the model. Stan developers often recommend to manually bound the variables that have finite support [135]. For p in Figure 2.2, we can set the bounds as follows: `real⟨lower=0, upper=1⟩ p;` This makes Stan produce the correct output: 7.0. The origin of the problem lies in the way Stan does sampling. For any sampling statement of the form: `p ~ beta(a,b)`, Stan computes the log probability as: `target += beta_lpdf(p | a,b)` and updates the log density (logpdf) of the model. Beta distribution has a support of $(0, 1)$. If p is assigned a value outside this range, it causes logpdf to be undefined, which affects convergence. When the bounds are manually bound, Stan ensures that the parameter is in the valid range.

Such properties that are important for inference are not enforced. Stan's development has been influenced by "Folk Theorem" [136], which implies that in case of a wrong inference, the problem can be overcome by changing the model, and moreover the inference algorithms

should not be made to work for various uncommon extreme program/data cases [136]. However, the "Folk Theorem" assumes that a developer has an intuition about the correct result, which may often not be the case as the PP systems are becoming mainstream. To help developers overcome such challenges, PP systems should provide additional information about the problems in the interaction between the model and the system. Recently, Stan developers proposed a "pedantic mode", as a way to diagnose various errors and bad modeling practices before running the inference [137], including range checks. We find this an interesting direction that can demonstrate the power of both probabilistic reasoning and static analysis, similar to lightweight static analyses in the traditional software development, e.g., [138].

Lesson Learned: Debugging accuracy problems requires not only domain knowledge but also a reasonable understanding of the PP system under test. The warning messages often provide hints if there is something wrong with the model. But the messages might not be informative enough to guide the user in fixing the model. This parallels the observations from compiler research on the importance of informative warnings for subsequent developer action [90, 139, 140].

Going forward, we note a promising application of static analysis to provide explicit hints about the model and its interaction with the inference algorithm without having to run the program. Like in compilers, they could provide *"useful warnings to alert developers to potentially problematic code fragments" and "suggestions to eliminate the warning"* [90] in the probabilistic setting. Tools like ProbFuzz have the potential to empirically discover the kinds of models that do not work well with a specific inference algorithm and inform such static analysis.

2.6.3 Fixing Bugs in PP systems Is Non-trivial

We found out that fixing the bugs, even the relatively straightforward dimension/boundary-value ones, is highly non-trivial and often involves changes to the design of the infrastructure (e.g., PyTorch and TensorFlow), that PP systems are built on.

As an example of a non-trivial problem, we reported a bug to Stan developers, which appears in some situations when the model is provided with an empty data array. In those cases the programs fails unexpectedly. The developers acknowledged the issue immediately, but even after an extensive discussion, the developers still have not been able to resolve the problem after several months.

In Edward, we submitted a pull request to ensure that the `n_samples` parameter of KLPQ inference was ≥ 0. The developers asked for the same fix to be made in several places in the KLQP inference: *"Cool! Can you also add this change to* ***`klqp.py`*** *for each* ***`initialize()`***

method?" We did as requested and our pull request was accepted and merged.

In Pyro, we identified that many distributions used from PyTorch do not have range checks. As we were discussing the potential fix with the developers, a PyTorch contributor independently started implementing their version of the fix. We discovered that the contributor's proposed fix had several bugs. Our tests that revealed bugs in the contributor's fix were driven by the failures that we had seen while running ProbFuzz-generated programs. Consequently, the contributor agreed to let us lead the fix, which has been approved for merging to the PyTorch repository.

Lesson Learned: Bugs in PP systems are not trivial to fix. Tests generated by ProbFuzz can help identify the causes of the problems. Moreover, automated testing can help discover incorrect and incomplete fixes.

2.6.4 Fixes Extend Across System Boundaries

As we were analyzing bugs and developing fixes, we found out that the failures that manifest in a PP system are often due to faults in the underlying infrastructure (PyTorch and TensorFlow). Therefore, some of the fixes we submitted were accepted by the developers of the underlying infrastructure. The accepted changes include a part of distribution checks in PyTorch plus many accompanying test cases and a fix to a bound for error reporting in TensorFlow.

We submitted our initial issues to the developers of Edward and Pyro, and they would typically direct us to propagate the fix to the underlying infrastructure. For Edward, the developer's response to our request to enable detailed checking of distribution ranges was *"That's an interesting suggestion...potentially useful utility. Can you raise this in TensorFlow?"* In Pyro, we submitted a pull request which added a check to prevent division by zero errors. Multiple Pyro developers responded and asked us to make the checks in PyTorch, so more people in the community will benefit: *"hello. thanks for the contribution! ... a more appropriate place... would be in pytorch... Thanks, this looks helpful...Thumbs up! I agree the PyTorch folks would appreciate better error checking, then a larger community could benefit from this fix."* The fix was accepted by PyTorch.

Lesson Learned: In PP systems, the errors and fixes can often extend across the boundaries of individual systems. ProbFuzz was effective in identifying such bugs since it analyzed and compared the end-to-end results of these composed systems.

Comparison with Traditional Testing Approaches. Inference bugs often require probabilistic reasoning and reasoning about accuracy, using, e.g., domain-specific oracles, metamorphic relations, or multiple implementations. As such, this category of bugs can be hard to catch using traditional testing techniques. Common techniques, such as coverage-based testing, would have problems because many of these bugs were caused by "faults of omission" [141]. Further, even bugs in covered code may require special values to manifest. Mutation testing of PP system code can potentially identify some bugs that result in program crashes or special values, but non-equivalent mutant survivals may indicate valid approximations rather than bugs [142], especially as tests in PP systems often only check whether the result lies in a loosely defined interval.

For the other bug categories, we give examples of previously unknown bugs that illustrate the advance of ProbFuzz over traditional testing approaches. For example, a dimension/boundary-value bug in Pyro manifested only when required parameters in two different functions were simultaneously out of acceptable ranges [143]. Conventional boundary-value analysis that targets one function at a time will not reveal this bug. As another example, in Edward, intermediate floating-point values produced by the SGLD inference algorithm led to `NaN` output when those values are "close enough" to the support bound [144]. Traditional boundary-value analysis may need to try many values near the bound to catch this bug. This bug remains open even after two workarounds that required advanced domain knowledge from the Edward developers.

A language/translation bug in Stan led to program crashes only on empty `int` arrays in the data, but not on empty `real` arrays [145]. Empty arrays are allowed in the data. The root-cause of the bug was that empty `int` arrays were implemented to be of data type `float`. Interestingly, the bug does not manifest in Stan itself, but in Stan's `PyStan` and `RStan` front-ends. Without the combination of domain knowledge on valid data elements and fuzzing, it will be difficult to catch such bugs with traditional testing techniques.

Scope. In our experiments, we used four templates, which focused on simple probabilistic models. Simple models can help developers understand potentially faulty executions and they were effective in finding bugs in the PP systems, but we did not aim for completeness of models in our evaluation. Going forward, PP system developers may also be interested in other common models that can be represented as templates in our language (e.g., hierarchical models, mixture models), and can be used to test various inference procedures, general or specialized for different model classes. However, ProbFuzz cannot generate arbitrary probabilistic programs, since its template language does not support while loops. Also,

ProbFuzz is not suited for bugs that require precise analysis, e.g., [146, 147].

Threats to Validity. They include internal, external, and construct.

Internal. The results of our bug study depend on the set of PP systems and bugs we examined. We mitigated this risk by studying real bugs in three state-of-the-art PP systems. We may have wrongly characterized existing bugs as being inference-related. To mitigate this, two coauthors independently inspected the bugs and (when possible) the corresponding GitHub issues. We only mark a bug to be inference related if both coauthors eventually agree, thus achieving a conservative estimate of the number of inference-related bugs. We mitigate ProbFuzz implementation errors with unit testing. As differential testing may wrongly flag a program as potentially buggy, so we had multiple rounds of discussion among ourselves, and finally reported potential bugs to the PP system developers to make the final decision.

External. The results of the characterization study and ProbFuzz may not generalize to all PP systems. Certain aspects of our experimental design help to mitigate this risk. The three PP systems are being actively developed, well-tested, and adopted. We also demonstrated that ProbFuzz can reproduce existing bugs in each of the three bug categories across the PP systems.

Construct. ProbFuzz is designed to catch the categories of bugs identified by our study and may not find arbitrary bugs in PP systems. Discovery of these bugs is not exclusive to ProbFuzz. Other general and emerging testing techniques can, in principle, find some of the bugs identified in our evaluation.

2.8 RELATED WORK

Verification and analysis of probabilistic programs. There are various approaches for verification of probabilisitic programs, including probabilistic abstract interpretation [148, 149], symbolic execution [150, 151, 152, 153], probabilistic model checking [154], and other methods [69, 155, 156, 157]. Unlike these systems, ProbFuzz aims to find bugs in the systems on which probabilistic programs run, and not for debugging or analyzing probabilistic programs.

Program Generation for Compiler and System Testing. Several techniques have been proposed for generating programs that are used in system testing. These include techniques for generating programs to test compilers [87, 88, 89, 90, 91, 92, 93, 94, 95, 96] and to test refactoring engines and symbolic execution engines [158, 159, 160]. ProbFuzz also generates programs, but does so for a different class of systems: PP systems, which are characterized by various probabilistic constraints on how to construct programs and measure accuracy of the output (instead of binary correctness). One technical difference between ProbFuzz and

earlier program-generation approaches is that ProbFuzz can generate programs in multiple languages—we currently generate Stan and Python from ProbFuzz, but more can be easily added. Lastly, ProbFuzz generates both programs and the data needed to run the programs, whereas all prior techniques generate only the programs (for compiler and system testing), or only the data (for testing programs).

Fuzzing. Researchers have previously proposed many fuzzing techniques [87, 161, 162, 163, 164, 165, 166, 167, 168, 169, 170, 171]. Grammar-based fuzzers [87, 162, 164, 165] encode knowledge about the structure of valid programs (more generally, inputs), but have no knowledge about the domain for which programs are typically written. The closest fuzzing approach that we found to ProbFuzz in terms of encoding domain knowledge is LangFuzz [161]. LangFuzz improves grammar-based fuzzing by first generating valid programs according to the grammar, and then mutating the programs based on knowledge about programs that previously caused invalid behavior. Therefore, LangFuzz incorporates domain knowledge in the form of historical invalid behaviors. In contrast to LangFuzz, the generation of programs by ProbFuzz already incorporates domain knowledge, without needing to perform any mutation or consider history.

Differential and Metamorphic Testing. Differential testing [172, 173, 174], or multiple-implementation testing [175, 176, 177, 178] use multiple implementations as oracles to find programs that can likely reveal bugs in PP system. ProbFuzz uses such approach in its Program Checker. Many problems in machine learning do not have a reference result, known as the 'no oracle' problem [179]. One solution to this problem is metamorphic testing [179, 180, 181, 182, 183, 184, 185, 186, 187], where *metamorphic relations* between the inputs and outputs of a program (or function) are leveraged to find inputs which cause outputs to diverge. Because metamorphic relations are hard to design, Srisakaokul et al. [175] recently proposed multiple-implementation testing of supervised machine-learning algorithms to find bugs. Implementations which classify differently from the majority are considered potentially buggy. Our differential testing of multiple inference algorithms is similar to [175].

2.9 SUMMARY

We presented the first study of existing bugs in probabilistic programming systems (PP systems), and proposed ProbFuzz for testing for such bugs. ProbFuzz generates probabilistic programs from a user-specified template for three PP systems: Edward, Pyro, and Stan. Our study of historical bugs in Edward, Pyro, and Stan showed that numerical bugs, accuracy bugs and dimensional and boundary-value bugs form the majority of bugs. We demonstrated

the ease of extending ProbFuzz by supporting several PP system versions in our study, and the applicability of ProbFuzz by showing that it can find existing bugs in the aforementioned categories in all three PP systems. ProbFuzz is already providing practical value: we reported 67 previously unknown bugs that we found by running ProbFuzz on recent versions of the three PP systems. We created pull requests with fixes for many of these bugs, 51 of which have been accepted by the developers. We believe that ProbFuzz opens a new line of research on testing probabilistic programming systems.

Chapter 3: PROGRAM REDUCTION FOR PROBABILISTIC PROGRAMMING SYSTEMS

3.1 INTRODUCTION

Probabilistic programming languages offer an intuitive way to model uncertainty by representing complex probabilistic models as simple programs [3, 67, 68, 69, 71, 72, 73, 75, 85, 124, 188, 189, 190, 191, 192, 193]. A key novelty of probabilistic programming is the separation between the *probabilistic modeling* and *probabilistic inference*. An end-programmer expresses probabilistic models in a *high-level language* with constructs for random choice (e.g., sampling from common distributions), conditioning on data (e.g., observation statements) and probabilistic queries (e.g., posterior distribution) [4].

A *probabilistic programming system* (*PP system*) automates many intricate details of probabilistic inference, while executing one of the common inference algorithms, such as Monte-Carlo sampling [194, 195] or Variational inference [86]. A PP system takes three inputs: 1) a probabilistic program, 2) a set of data points on which to perform inference, and 3) arguments of the inference algorithm. Typically, PP systems compile the probabilistic program into an efficient low-level inference procedure, which includes initializing the underlying inference algorithm, translating of probabilistic programs (models) to an intermediate representation, simplifying the model, compiling to low-level API (e.g., Tensorflow), and many others.

The numerical and approximate nature of PP systems and implementation complexity make it hard to ensure their correctness, and subtle bugs can easily remain unnoticed [79, 80, 157]. Our recent study [196] showed that over 25% of all bugs in three popular systems are domain specific, including algorithmic, numerical, boundary condition, dimensional, and accuracy bugs. The bugs manifest as wrong results, crashes, infinite loops, or numerical exceptions.

If a failure occurs during the execution of a PP system, the developer typically has to figure out the source of the problem manually. This is not an easy task: while probabilistic programs are intuitive to write, they can be notoriously hard to analyze [157, 197]. Probabilistic programs typically have a small number of lines of code, but they exercise many functionalities of the underlying PP system. For instance, an execution of a simple 10-line program in Stan [66], one of the most popular PP systems, can execute over 6000 lines of code of Stan implementation.

To be able to reproduce and analyze the failures in PP systems, the developers of PP system suggest bug reports with self-contained and minimalistic tests, e.g., in Stan: *"the key to a successful bug report is to provide as much context as possible, ideally in the form of a*

small reproducible example" [198]. This requirement is similar to the one from the standard compilers (e.g., [199]). Minimized programs help with both debugging (e.g., calculating the reference result) and speeding up regression testing (by executing the programs faster). However, coming up with minimal programs requires significant manual effort through trial-and-error.

Program reduction has been instrumental in the tasks of compiler testing and debugging. For standard compilers, researchers proposed various methods to reduce the size of bug-revealing programs such that the reduced program still exposes the bug, but helps the developer better understand and debug the program execution [199, 200, 201, 202]. Our insight is that the same conceptual approach can apply for debugging PP systems: just like a compiler that translates an input program, PP system translates and/or executes a triple of probabilistic program, data, and inference arguments. However, the existing program reduction approaches for conventional languages, either operate on textual representation [200], potentially generating many illegal programs, or use only syntactic information about the programs [201, 202], but do not leverage semantic information; they are also oblivious to the inference arguments (e.g., the number of Monte-Carlo samples).

Our Work. We present Storm, a novel approach for automatically reducing probabilistic programs and show its utility in the scenarios of testing and debugging PP systems. Storm applies various transformations to reduce the probabilistic programs. Unlike existing approaches for conventional programs, Storm leverages program analysis and probabilistic reasoning to simplify bug-revealing probabilistic programs. We show the benefits of the domain-specific and probabilistic information about the programs.

We formulate our problem in the spirit of [202]: Given a probabilistic program P, data d, and inference arguments θ that have a property ψ (e.g., a PP system execution fails with a particular error message), the goal of probabilistic program reduction is to find a smaller (P', d', θ'), that has the same property, $\psi(P', d', \theta') = \psi(P, d, \theta)$ (e.g., the PP system execution fails with the same error message).

Storm is a *generic framework* that uses both syntactic and domain-specific, semantic information about probabilistic programs to generate only valid probabilistic programs. We designed Storm to be language-agnostic: it translates programs from the existing systems to a common intermediate representation, Storm-IR. We define all our analyses and transformations on Storm-IR and finally output the reduced program back to the source language. We present the translation of two popular languages (Stan [66] and Pyro [191]) with significantly different syntax and language models.

Storm is a *transformation-based framework*. It supports both conventional program

transformations and novel probabilistic transformations. Novel transformations include *data reducer* (which aims to keep specific bug-revealing values in the data set), *distribution simplifier* (which replaces complex distributions or parameters with appropriate constants, expressions, or simpler distributions), *parameter remover* (which removes a parameter and replaces its references with a suitable constant), *math-function call remover* (which replaces common mathematical functions with constants), and *inference argument reducer* (which finds a minimum number of samples or iterations of the inference algorithm required to reproduce a failure). They augment the basic program transformers, such as arithmetic simplifier, removers for conditionals, loops, function calls, and assignments (similar to C-Reduce [199]). Storm's reduction algorithm reduces the program size by iteratively applying both the basic and domain-specific transformations and performing lightweight analysis on the program's intermediate representation (including dependence, interval, type, and data-flow analysis).

Results. We use the reducer to generate smaller programs that reveal failures in two state-of-the-art PP systems: Stan [66, 85, 86], one of the most mature and frequently-used PP systems, and Pyro [191], a Python-based deep probabilistic modeling framework from Uber. We studied three sources of bugs: 1) a probabilistic bug database we created in previous research [196], which includes test programs that were already minimized by a human, 2) new bugs discovered using ProbFuzz [196], and 3) a repository with representative Stan models that offers larger probabilistic programs [203]. In total, we analyzed 47 programs (34 from Stan and 13 from Pyro).

Our results show that Storm's reduction strategies often generates significantly smaller programs than those provided by the users or developers. In particular, Storm was able to remove non-trivial program constructs in 45 programs, reduce the data size in 30 programs out of 33 programs that have data, and reduce the execution time of the inference algorithm (e.g., by reducing the number of Monte-Carlo simulations or Variational inference iterations) in 46 programs. Storm shows a significant improvement in the number of removed data points and program constructs over the baseline approach that applies only basic transformations.

Contributions. This chapter makes the following contributions:

- ⋆ We present Storm, which is, to the best of our knowledge, the first reduction framework for probabilistic programs.
- ⋆ We introduce a program reduction algorithm that is aware of probabilistic information and guided by program analysis.

⋆ We introduce domain specific transformations in addition to basic transformations used for conventional languages.

⋆ We evaluate Storm on existing bug-revealing programs from popular probabilistic programming systems, Stan and Pyro.

3.2 EXAMPLE

Figure 3.1 presents a bug-revealing program in Stan, taken from the bug issue *Stan1610* [204]. We will demonstrate Storm's ability to reduce this program while still revealing the same bug in Stan.

3.2.1 Original Program

The program in Figure 3.1 represents a variant of Latent Dirichlet Allocation (LDA) model [205]. In LDA, each document is assumed to contain a mixture of topics and each topic is assumed to use a small set of words frequently. Using LDA model, users try to infer the distribution of words and topics in observed documents. The program in Figure 3.1 represents the topic distributions for users and items instead of documents. It consists of three parts:

- *Data block* (lines 1-13): It specifies the type and dimension of the input data which is to be used to condition the probabilistic model. It contains the dimensions and the names of all constants, priors, and observed data points.

- *Parameters block* (lines 14-18): It contains the random variables whose posterior distribution Stan should infer.

- *Model* (lines 19-36): The model establishes the relationship between the observed and unobserved variables. First, it assigns a *prior* to all the parameters, which denotes the user's belief of the distribution of their values. Here, all the parameters are assigned priors from *Dirichlet* distribution (lines 20-25). Then it specifies the relation of the variables to the data. It implements LDA using custom log-density updates (lines 27-33). The built-in function *increment_log_prob* updates explicitly the log density of the posterior distribution with the value of the inner expression.

Dataset:

U = 28; I = 84; N = 326; K = 10; V = 17
word = [(326 float values)];
item = [(326 float values)];
user = [(326 float values)];
alpha_user = [(10 float values)];
alpha_item = [(10 float values)];
beta = [(17 float values)];

Model:

```
data {
  int K;
  int V;
  int U;
  int I;
  int N;
  int word[N];
  int item[N];
  int user[N];
  vector[K] alpha_user;
  vector[K] alpha_item;
  vector[V] beta;
}
parameters {
  simplex[K] item_topics[I];
  simplex[V] word_topics[K];
}
model {
  for (i in 1:I)
    item_topics[i] ~ dirichlet(alpha_item);
  for (u in 1:U)
    user_topics[u] ~ dirichlet(alpha_user);
  for (k in 1:K)
    word_topics[k] ~ dirichlet(beta);
  for (n in 1:N) {
    real gamma[K];

    for (k in 1:K){

      gamma[k] <- log(item_topics[item[n], k] +
          user_topics[user[n], k]) + log(word_topics[k, word
          [n]]);
    }
    increment_log_prob(log_sum_exp(gamma));
  }
}
```

Inference Arguments:

Engine = ADVI; Iters = 1000

Figure 3.1: Example – Original Code and Data

Dataset:

K = 1; V = 2
beta = [0.0588235,
    0.0588235];

Model:

```
data {
  int K;
  int V;
  vector[V] beta;
}
parameters {
  simplex[V]
      word_topics[K];
}
model {
  for (k in 1:K)
    word_topics[k] ~
        dirichlet(beta);
}
```

Inference Arguments:

Engine = ADVI; Iters =
    125

Figure 3.2: Example: Reduced Code and Data

Data. In addition to the program, the test case consists of data points, which give concrete values to all the constants and the vectors (the actual values omitted from Figure 3.1). For this program, the number of users `U` is 28, items `I` is 84, word instances `N` is 326, topics `K` is 10, unique words `V` is 17. Overall, the data size is around 4 KB.

Inference. The program runs with Stan's ADVI (variational) inference engine [86], which approximates the posterior distribution to a family of distributions with unknown parameters and converts the inference problem into an optimization problem. The algorithm then runs the model and tries to minimize the distance between the posterior and the chosen family of distributions for a given number of iterations. This number is given as the argument (1000).

Bug. This program produces `NaN` (Not-a-Number) in the output after 70 iterations when run using ADVI in Stan 2.7.0. The failure was due to a bug in the inference engine, which does not adapt its step-size sequence argument correctly, leading to `NaN`.

3.2.2 Reduced Program and Data

Figure 3.2 presents the test case minimized by Storm. The program now only samples from one Dirichlet distribution (instead of the previous complicated computation) and hence does not need to compute the posterior distribution. This program now has only 12 lines of code (compared to 36 in the original) and 12 instead of the original 70 program constructs. Table 3.1 presents the full reduction statistics. Because the model does not compute the posterior distributions, the new model does not need any of the user provided data (`word`, `item`, and `user`), which significantly simplifies reasoning about its correctness. Our manual inspection shows that the reduced program still reproduces the same bug as the original program, despite its much smaller size.

Table 3.1: Example: Reduction Statistics

Reduction	%	Ratio
Lines of Code	69%	(11 / 36)
Code Constructs	83%	(12 / 70)
Data Points	98%	(41 B/4 KB)
Algorithm Iters.	87.5%	(125 / 1000)

To reduce this program, Storm translates the source code to the intermediate language Storm-IR, transforms the program, and outputs the source of the reduced program. The transformations include basic (e.g., removing statements or expressions) and those specific to

the probabilistic domain (e.g., Math-Function Call Remover, which replaces mathematical functions with appropriate constant values, and Data Reducer, which reduces data size). Storm applied basic transformations 17 times and domain-specific transformations 25 times to reduce the program. Eleven of these transformations were distinct. The results shows that Storm effectively leverages both basic and domain specific transformations.

3.2.3 Benefits of Program Reduction

Simplified Debugging and Fault-Localization: The reduced program immediately points out that the problem with this program may be caused by some interaction between the Dirichlet distribution and the inference engine. This is in contrast to the original program, where a developer would need to think about various aspects of the implementation – e.g., does the code correctly represent the model, how to simulate discrete distributions with log-probabilities – and the data – e.g., are the values and the parameters in range.

Easier to Derive the Reference Solution: The reduced program simply samples values from the Dirichlet distribution. Its probability density function can be easily obtained from a textbook:

$$p(x_1, x_2|\mathit{beta1}, \mathit{beta2}) = \frac{\Gamma(\mathit{beta1} + \mathit{beta2})}{\Gamma(\mathit{beta1}) \cdot \Gamma(\mathit{beta2}) \cdot x_1^{\mathit{beta1}-1} \cdot x_2^{\mathit{beta2}-1}} \tag{3.1}$$

By replacing *beta1* and *beta2* with values from Figure 3.2, the developer can compute the distribution of the program: $p(x_1, x_2) = 0.029 \cdot x_1^{-0.941} \cdot x_2^{-0.941}$. Then, the developer simply needs to check that the inference results conform to this probability distribution.

Reduction also helps for programs that do not have a closed-form solution. A common strategy is to use a different language or an inference language version and run Monte-Carlo simulation for a large number of iterations (e.g., over 10000 times) to get a good estimate of the distribution. Reduced program takes significantly less time to run than the original program, e.g., running Stan's NUTS engine for 10000 iterations on our reduced example takes 0.26 s, while running the original example takes 463 s (1781x slower).

Faster Regression Testing: Reducing the computation, data, and the iteration count directly translate to faster regression testing. Obtaining the reference solution also helps in creating effective regression test. Running the regression test for our example takes 0.02 s, while the original program is 214 times slower.

$$
\begin{array}{rcl}
x & \in & \mathit{Vars} \\
c & \in & \mathit{Consts} \cup \{-\infty, \infty\} \\
\mathit{aop} & \in & \{+, -, *, /, \hat{}\} \\
\mathit{bop} & \in & \{=, >, ...\} \\
\mathit{Dist} & \in & \{\text{Normal, Uniform, Beta, ...}\} \\
\mathit{ID} & \in & \mathit{String}
\end{array}
$$

$$
\begin{array}{rcl}
\text{Range} & ::= & \text{<Expr ,Expr>} \\
\text{Dims} & ::= & [\ \text{Expr}^{+}\ ] \\
\text{Type} & ::= & \texttt{Int} \mid \texttt{Float} \mid \text{Type Dims} \\
\text{Decl} & ::= & x\ \texttt{:}\ \text{Type Limits? Dims?} \mid x\ \texttt{:}\ [c^{+}] \\
\text{Expr} & ::= & c \mid x\ \text{Dims?} \mid \text{Expr } \mathit{aop}\ \text{Expr} \mid \text{Expr } \mathit{bop}\ \text{Expr} \\
 & & \mid \text{Function} \mid \mathit{String} \\
\text{Query} & ::= & \texttt{posterior}(x) \mid \texttt{expectation}(x) \\
\text{Function} & ::= & \text{ID}(\text{Expr}^{*}) \\
\text{FunctionDef} & ::= & \texttt{def}\ \mathit{ID}\ ((\text{Type } \mathit{ID})^{*})\ \{\ \text{Statement}^{*}\ \} \\
\text{Limits} & ::= & \text{Range} \\
\text{Statement} & ::= & x = \text{Expr} \\
 & & \mid \text{for } x \in \text{Range; } \{\ \text{Statement}^{*}\ \} \\
 & & \mid \texttt{observe}(\mathit{Dist}(\text{Expr}^{+}),\ x) \\
 & & \mid \texttt{if}\ (\text{Expr})\ \texttt{then}\ \text{Statement}^{*}\ \texttt{else}\ \text{Statement}^{*} \\
 & & \mid x\ \texttt{:=}\ \mathit{Dist}(\text{Expr}^{+}) \\
 & & \mid \text{Function} \\
 & & \mid \text{Decl} \\
\text{Program} & ::= & \text{FunctionDef}^{*}\ \text{Statement}^{*}\ \text{Query}^{*}
\end{array}
$$

Figure 3.3: Syntax of Intermediate Representation

Other Applications of Program Reduction: Since it has the flexible choice of reduction objective ψ, Storm can be successfully used for other scenarios than reproducing bugs. We discuss one such case, minimizing the program while maintaining coverage in Section 3.7.

3.3 STORM OVERVIEW

Figure 3.4 presents the high-level overview of Storm. The inputs to Storm are 1) a probabilistic program, 2) data, and 3) the arguments of the inference (e.g., the number of samples). In each step, Storm checks whether the transformed (reduced) program satisfies the *reduction property* (ψ), a logical predicate that relates the outputs and the status of the original and the reduced programs. In this chapter, we mainly consider the property that the reduced program reproduces the same error status and message as the original program.

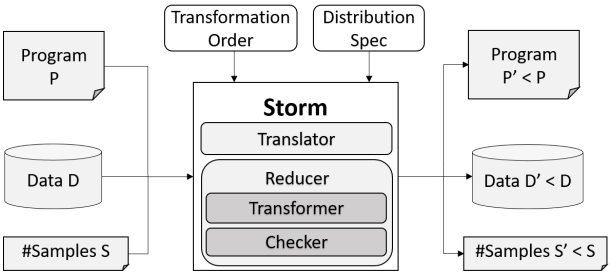


Figure 3.4: Storm Overview

3.3.1 Translators and Storm-IR

Storm translates each program to its intermediate representation, Storm-IR, on which it applies analysis and transformations. The translator is responsible for converting the program from the language of the existing PP system to the Storm intermediate representation *and* converting the reduced program and data from the intermediate representation back to the source language of the PP system. We developed translators for Stan and Pyro using Antlr [206].

Figure 3.3 presents the core syntax of Storm-IR. Storm-IR is an imperative language with standard constructs like arithmetic operations, *conditionals* and *loops*, and probabilistic constructs like distributions (*Dist*) and *observe* (which conditions the model based on given data). Each program in Storm-IR has three main components: user-defined functions (*FunctionDef*), a set of statements which describe the probabilistic model, and one or more probabilistic queries on the model. A query can be either be for the posterior distribution (*posterior*) of a parameter in the model or for its expected value (*expectation*). The Storm-IR has three kinds of variables: data variables, parameters, and local variables. The variables are declared as either primitives or n-dimensional arrays. Optionally, the parameters have a *Limits* construct which specifies the range in which the values of the parameter must be constrained during inference.

Stan and Pyro have significant differences in both syntax and the core design. For instance, Stan users need to specify the model in Stan's domain-specific language, which clearly separates the data, parameters, and the model code into different blocks. Pyro programs are written in Python, which makes it easier to write and compose different models. Unlike Stan, Pyro requires defining the posterior distribution for each parameter. Storm-IR is general enough to represent the core of majority of the example programs included in the repositories of these languages and allows the translators to handle the language-specific features discussed above. Our intermediate language draws inspiration from Probfuzz [196], but improves expressivity and generality (e.g., it allows arbitrary inter-leavings of statements like sampling, assignment, observes and loops). This allows the Storm-IR to represent a

Algorithm 3.1: Storm Algorithm

Input: Program P_{in}, Data D_{in}, #Samples S_{in},
Transformation Order O
Output: Reduced Program P, Data D, #Samples S

procedure MINIMIZE
  $Changed \leftarrow True$
  $P, D, S \leftarrow P_{in}, D_{in}, S_{in}$
  while $Changed$ **do**
    $Changed \leftarrow False$
    for T in O **do**
      $P_{red}, D_{red}, S_{red}, C \leftarrow Transform(T, P, D, S)$
      if C **then**
        $P, D, S \leftarrow P_{red}, D_{red}, S_{red}$
        $Changed \leftarrow True$
      end if
    end for
  end while
return P, D, S
end procedure

richer and more diverse set of probabilistic programs used across different PP systems.

3.3.2 Reduction Algorithm

Storm simplifies the structure of the programs by applying transformations and analyses on the intermediate representation. We describe the transformations we implemented in Section 3.4.

Helper Analyses. To ensure that the transformed program is syntactically correct, our transformers use several helper analyses for probabilistic programs. They include Dimensional, Type, Interval and standard DataFlow analyses (e.g., Def-Use). Dimensional analysis computes the dimension and type of any given expression. Interval analysis helps compute the range of values valid for a given expression. Def-Use analysis finds the uses of a variable in the model after it is declared. The transformations that simplify distribution expressions or replace parameters with constants can use the results of the analysis to make sure that the reduced program will not fail to run due to the range or dimension mismatches (e.g., it prevents setting the negative constant as the variance).

To support these analyses, Storm needs additional (domain-specific) information about common distributions and mathematical and probabilistic functions. The information includes the names and ranges of parameters and their support (the ranges of their outputs). For

Algorithm 3.2: Transform Algorithm

Input: Transformation T, Program P_{in}, Data D_{in}, #Samples S_{in}
Output: Reduced Program P, Data D, #Samples S, Changed status C

procedure TRANSFORM
  $C \leftarrow False$
  $P, D, S \leftarrow P_{in}, D_{in}, S_{in}$
  $L \leftarrow T.getLocations(P, D, S)$
  $i \leftarrow 0$
  while i ¡ $L.length$ **do**
    $P', D', S', Modified \leftarrow T.Transform(P, D, S, L(i))$
    if *Modified* **then**
      $Reproduced \leftarrow Checker(P', D', S')$
      if *Reproduced* **then**
        $P, D, S \leftarrow P', D', S'$
        $C \leftarrow True$
        $L.remove(i)$
        $i = 0$
      else
        $i = i + 1$
      end if
    end if
  end while
return P, D, S, C
end procedure

instance, the specification of Normal distribution states that the distribution is continuous and has unconstrained support; its first parameter (the mean) is an unbounded real and the second one (the variance) is a positive real.

Main Algorithm. Algorithm 3.1 presents the reduction algorithm. It takes the program P, data D, and number of samples or iterations S (if available). It can optionally take the order of transformations O which is to be used during reduction process. The algorithm is iterative fixed point computation, which in each step tries to apply the transformations and then checks whether the reduced programs satisfy the reduction property. The variable *Changed* tracks whether any transformation was successfully applied during the current iteration. In each iteration, the algorithm tries to reduce the program using each transformer T according to the pre-specified order O. The algorithm stops when the iteration cannot apply any reduction.

The *Transform* algorithm (Algorithm 3.2) takes as input a transformer T, program P_{in}, data D_{in}, and number of samples S_{in}. It finds all candidate locations for the transformation in the program for transformer T using *getLocations* function. For transformations which reduce data, this would return the data items in the program. For Inference Argument

Reducer, this returns the inference parameters to reduce (in this case only Samples S). Next, for each candidate, the transformer T tries to transform it and check if the reduced program still reproduces the same failure as the non-reduced version using the *Checker* function. If it succeeds, then the triple of program, data, and, samples are updated, the candidate is removed from the list L and i is reset to 0. Otherwise, it moves to the next candidate. Resetting i to zero allows the transformer to re-check the previous locations which may now be modifiable after the recent change. Finally, the algorithm returns the program, data, and samples along with the indicator of whether any of them were transformed.

The *Checker* translates the program in Storm-IR back to the source code, runs it on the data, and monitors the execution. For program failures, it is often sufficient to expect the *exact* exception strings in output, e.g., *"Input vector ... is -nan"* or *"error: invalid cast from type 'stan::math::var' to type 'double'"*. For infinite loops, we set a reasonable timeout interval; the transformed program reproduces the failure if it still times out (like the original program). The Checker returns "True" if the execution of the candidate program still has the property of interest (e.g., the same error message), or returns "False" if the transformed program does not have this property.

3.4 TRANSFORMATIONS

We divide Storm's transformations into *basic* – typical structural reduction transformations that do not require probabilistic domain-knowledge – and *probabilistic* – that use the domain-knowledge. We describe the transformations next, and formally define them in the Section 3.4.4.

3.4.1 Basic Transformations

Storm implements the common statement-level and expression-level transformations, which can be found in conventional program reduction tools, such as C-Reduce [199]. The transformations include *Loop Remover* (removes entire loop), *Loop Variable Remover* (the loop variable is replaced with a constant), *Conditional Remover* (randomly chooses one of the two branches), *Function Statement Remover* (removes a function call statement), and *Dead Variable Remover* (finds variables and data items which have been assigned constant values or sampled from distributions but never used).

Storm also has an *Arithmetic Simplifier*, which reduces arithmetic expressions by replacing variables with constants, complex expressions with simpler, etc. For instance, it can convert $a + b$ to either a or b. Since the operands can be arrays or vectors or matrices, Storm

performs type and dimensional analysis of the expression and replaces the expression with an appropriate constant-valued data structure. To reduce non-determinism, the arithmetic simplifier always tries to remove the first operand first. Only if the reduced program fails to reproduce the failure, it tries to remove the second operand and checks for failure reproduction again.

3.4.2 Probabilistic Transformations

Data Reducer. The input data in Storm-IR contains primitives like integers or floating-point numbers or more complex data structures like vectors and matrices. In some cases, boundary or special values in the data may cause run-time failures. Isolating these values in a smaller data set can ease debugging.

The Data Reducer picks one data item (typically a vector or a matrix) and tries to reduce it, by successively subdividing the number of values that remain in the data structure. This transformation also checks that any related data items maintain the same dimensions. For example, in a linear regression model, which models $y = a \cdot x + b$, it is important that the arrays with values for x and y have the same size, and therefore Storm reduces them in the same way and with the corresponding data values.

Parameter Remover. The unobserved variables which must be inferred from the observed data are usually specified as parameters in Storm-IR. This transformer replaces the use of the parameter with a constant value, vector, or matrix. Storm chooses the constant values randomly, from the set of those within the support of the prior distribution and ensures that the dimensions are maintained.

Math-Function Call Remover. Stan and Pyro provide inbuilt mathematical functions (*log*, *exp*, *abs*, etc.) and probability-related functions (*logit*, *tgamma*, *gamma_p*, etc.). Storm replaces such function calls with a value in the expected output range of the function. These kinds of functions require domain knowledge; for example, the output of *gamma_p* is always positive. For overloaded functions, Storm performs type and dimensional analyses on the function arguments to ensure that the expression is valid (such analyses are typically not done by the conventional program reducing approaches).

Distribution Simplifier. This transformation replaces less-often used distributions like *Laplace*, *Weibull*, etc. with more commonly used distributions like *Normal* or *Uniform*. If the program already uses the simpler distributions, it tries to reduce the parameters to standard values. For example, it might reduce $Normal(52.15, 10.2)$ to $Normal(0, 1)$, a standard normal distribution. These transformations are useful when a developer wants to understand the reduced program and manually calculate reference solutions. They can also

help with fault-localization, by pinpointing that the failure is (not) due to less-commonly used distributions.

Inference Argument Reducer. Stan and Pyro implement two kinds of Monte-Carlo sampling algorithms: HMC [195] and NUTS [194]. They allow the user to specify the number of iterations to run, which determines the number of samples to be taken from the posterior distribution for inference. For variational inference algorithms, like ADVI [86] in Stan and SVI [207] in Pyro, the iterations determine the maximum steps the optimization algorithm might use. The Inference Argument Reducer searches for the minimum number of iterations that reproduces the failure. In each round, the Inference Argument Reducer halves the iterations, starting from the initial value, and checks whether the failure is reproduced.

Limits Remover. For every data and parameter variable, some languages (Stan being a prominent example) allow the user to specify a set of lower and/or upper limits of the parameters. The limiting helps the sampling algorithm focus on a subset of the input space and converge faster. But these limits can be ill-specified in practice. This transformer attempts to remove the limits and checks if the program still fails. It is analogous to changing the variable type in conventional programs.

3.4.3 Transformation Orders

An important component of Algorithm 3.1 is the transformation order O, which can affect the quality and speed of reduction on a given benchmark. We evaluated our algorithm on six orders that we briefly outline next.

Random order (**Rnd**) chooses each transformation with uniform probability without replacement. Fixed order (**Fixed**) is an order we manually chose for the experiments based on our experience and understanding of transformations. Size of transformation order (**Size**) applies first the transformations that change more code (e.g., *Loop Remover* and *Conditional Remover*). Cost of analysis order (**CoA**) ranks the transformations such that transformations which do not require any analyses (e.g., *LoopRemover*, and *FunctionStatementRemover*) execute before the transformations which require one or more analyses techniques (e.g., *Arithmetic Simplifier*). Basic-Probabilistic order (**B-P**) applies first all basic transformations, then probabilistic ones. Probabilistic-Basic order (**P-B**) first applies all probabilistic transformations, than basic ones. We discuss the effect of transformation orders in Section 3.6.4.

3.4.4 Transformation rules

We formally define the transformation rules:

- **Loop Remover**:
  for(x *in range*) $b \mapsto$ *skip*

- **Loop Var Remover**:
  for(x *in range*) $b \mapsto b[x/c]$
  where $c \in$ *range*

- **Conditional Remover**:
  if(*expr*) then b else $b' \mapsto b$
  if(*expr*) then b else $b' \mapsto b'$

- **Function Statement Remover**:
  $func(e_1, ...) \mapsto$ *skip*

- **Assignment Remover**:
  $x =$ *expr* $\mapsto$ *skip*
  if $uses(a1) == 0$

- **Sampling Remover**:
  $x :=$ *dist*$(e_1, ...) \mapsto$ *skip*
  if $uses(x) == 0$

- **Observe Remover:**
  `observe`(*dist*$(e_1, ...), x) \mapsto$ *skip*

- **Arithmetic Simplifier**:
  $a\, op\, b \mapsto a$
  $a\, op\, b \mapsto b$
  $a\, op\, b \mapsto c$
  where $op \in \{+, -, *, /, \hat{}\}$ and $c \in \mathbb{Z}$ or $c \in \mathbb{R}$

- **Data Reducer**:
  $D : [d_1, ..., d_N] \mapsto D : [d_1, ..., d_m]$
  where $m = N/2$
  $D : [d_1, ..., d_N] \mapsto D : [d_{m+1}, ..., d_N]$
  where $m = \lfloor N/2 \rfloor$

- **Parameter Remover**:
  $p := dist(p_1, ..p_N) \mapsto skip \wedge$
  $q = expr\, op\, p \mapsto q = expr\, op\, c \wedge$
  $p : type \mapsto skip$
  where $c \in support(dist)$, op $\in \{+, -, *, /, \hat{}\}$, and, $p, q \in Vars$

- **Math-Function Call Remover**:
  $x = func(e_1, ...) \mapsto x = c$
  where $c \in range(func)$

- **Unused Item Remover**:
  $x : [c+] \mapsto skip$ if $uses(x) == 0$
  $x : type \mapsto skip$ if $uses(x) == 0$
  $x : expr \mapsto skip$ if $uses(x) == 0$

- **Distribution Simplifier**:
  $p := dist(e_1, ..., e_N) \mapsto p := dist'(f_1, ..., f_M)$
  where $dist$ and $dist'$ have same support

- **Limits Remover**:
  $x : type\ limits \mapsto x : type$

- **Inference Argument Reducer (Sampling)**:
  $Infer(p_1, p_2, ..., p_N, iters_1) \mapsto Infer(p_1, p_2, ..., p_N, iters_2)$
  where $iters_2 = \lfloor iters_1/2 \rfloor$

3.5 METHODOLOGY

This section presents our methodology for collecting and categorizing program that expose bugs in Stan and Pyro.

3.5.1 Selection of Bugs

Stan. To obtain probabilistic programs that reveal existing bugs for Stan, we studied the bug reports from the existing bug-database for probabilistic programming systems [196]. The database contains 138 probabilistic bugs, divided in four categories. We denote each such

bug with the prefix "stan" followed by the issue identifier. We selected only the bugs with reproducible test cases.

We augment the programs from the bug reports with additional bug-revealing programs from Stan's repository of models [203]. We obtained a set of 367 probabilistic programs from Stan's public repository [203]. We ran the programs across the versions of Stan 2.3, 2.5, 2.6, 2.6.2, 2.7, 2.9, 2.10, 2.14, 2.15 and 2.18 (the latest). We ran each program using three inference algorithms available in Stan: NUTS [194], HMC [195], and ADVI [86]. We identified those programs that produce a failure (compile but either crash, produce numerical errors, or loop infinitely) in one of the earlier versions, but produce the correct result in the latest version. Those programs are representative of those that would reveal true bugs in the PP system in real operation. These programs were considerably larger than the ones obtained from the bug reports in both code (more than 90%) and data (more than 100%) on average.

In total, we tested Storm on 23 bugs from Stan issues and 11 programs taken from Stan's example models repository. The size of test cases range from 5 to 57 lines of code (excluding blank lines and comments). We used Cmdstan to run all the programs, except three that require PyStan. Overall, the programs cover four inference methods: Sampling (NUTS [194] and HMC [195]), Variational (ADVI [86]) and Optimization (also known as MAP [208]), and one simulation method: Fixed Param (FP); additionally, some failures were due to bugs in the Stan compiler code (stan723), and bugs in Diagnostic mode (stan1308), which is used to test computations of gradient and log-probability and flag any issues.

Pyro. For Pyro, we collected bug-revealing programs from two sources. We obtained 6 bug-revealing programs from [196]. We named those programs pf1-pf6. We also converted the programs obtained from Stan's repository of models [203] into Pyro programs and ran them using the three recent versions of Pyro, 0.2.0, 0.2.1, and 0.3.0. We identified 7 programs (dyes, dyes_020, ES, ES_020, GP2, GP2_020, radon) which run without failures in the current released version of Pyro (0.3.0), but crash in the older and buggy versions. The program lines range from 36 to 77. All the programs were run using Stochastic Variational Inference (SVI) algorithm.

3.5.2 Bug Classification

Following the characterization from [196], we classified the bugs as:

Crashing Bugs. These bugs cause compilation-time or run-time failures with error messages in the output, such as *"runtime error: load of value 3, which is not a valid value for type 'bool'"*, *"Segmentation fault: 11"*, *"Domain error in arguments"* etc. Many of these bugs are

due to the out-of-bounds accesses or wrong dimensions of the data structures. We reproduced 5 such bugs in this category for Stan and all 13 Pyro bugs fall in this category.

Numerical Bugs/Infinite Loops. Numerical bugs include special values like `NaN`s or `Inf`s in the output, which usually appear due to missing support for handling boundary conditions. We reproduced 9 Stan bugs from this category with inputs provided by the bug reports. A special class of numerical bugs are those that cause infinite loops during inference. We reproduced 2 Stan infinite loop bugs.

Accuracy/Unexpected Output Issues. For the cases, the execution does not crash but produces some unexpected values. For example, in one case, the computation of effective sample size of a parameter for the NUTS engine in Stan was incorrect due to a bug in the code. We reproduced 5 bugs from this category with inputs provided by Stan's bug reports.

Language/Implementation. These bugs appear while translating the program's source-code. We replicated 8 such bugs for Stan. Finally, we also consider 3 general coding bugs.

3.5.3 Reduction Metrics

To demonstrate the quality of test case minimization, we collect several metrics during experimentation. For code reduction, we consider two metrics that characterize the size of the program:

- Lines of code, without empty lines or comments.
- The count of non-terminal language constructs in Stan's parse tree (e.g., loops, sampling statements, conditionals, or arithmetic operations) for the given test case.

We use count of non-terminals since the probabilistic programs have a high-level of expressiveness, and a single change (smaller than a line) in the code may make significant impact on the accuracy, analyzability, or execution time of the program. For the grammar that we used for Stan, there are 42 such unique constructs. For Pyro, we use the Python grammar, which has 41 unique constructs.

In this work, we use a metric for code reduction known as *Size Reduction Rate (SRR)* defined in [209] as:

$$Score(t_o, t_{red}) = \frac{Size(t_o) - Size(t_{red})}{Size(t_o)}, \tag{3.2}$$

where t_o is the original test case, t_{red} is the reduced test case, $Size(t)$ is the size of the test case using the metrics defined above. To compare data savings $DataRed$, we use the following

metric:

$$DataRed(t_o, t_{red}) = \frac{DataSize(t_o) - DataSize(t_{red})}{DataSize(t_o)} \tag{3.3}$$

where $DataSize(t)$ is number of bytes in the data input for t. Finally, we calculate the ratio of the number of samples/iterations in the reduced test case to that in the original.

Table 3.2: Stan Example-Models Reduced Using Storm

Test	**SRR**	**LoC**	**Data Red.**	**Iters.**
arma11 (VI)	44/85	20/25	49.9%	3/1000
arma11_alt (VI)	46/73	20/23	49.9%	1/1000
dogs_log (VI)	10/98	6/33	100.0%	1/1000
roaches (VI)	11/38	8/18	99.5%	1/1000
roaches_od (VI)	23/74	12/28	99.2%	1/1000
roaches_od_2 (VI)	23/74	14/28	64.8%	1/1000
salm2 (VI)	24/74	16/27	51.1%	1/1000
salm2_2 (VI)	1/74	2/27	100.0%	1/1000
salm (VI)	38/128	22/47	41.0%	1000/1000
stagnant (VI)	11/76	9/26	98.7%	125/1000
survey (VI)	35/65	28/32	36.4%	1/1000
Avg. Savings	68.37%	49.27%	71.86%	103/1000

3.6 EVALUATION

We evaluate experimentally the following research questions:

RQ1 How effective is Storm in reducing test cases?

RQ2 How much benefit do probabilistic transformations provide?

RQ3 How much does program reduction speed up inference?

RQ4 How important is the order of transformation in reduction?

3.6.1 Test Cases Reduced by Storm

Tables 3.2, 3.3, and 3.4 show the performance of Storm for the bugs in Stan examples, Stan issues, and Pyro examples, respectively. Column 1 (**Test**) is the test-case identifier – an

Table 3.3: Stan Github-Issues Reduced Using Storm

Test	**SRR**	**LoC**	**Data Red.**	**Iters.**
stan240 (NUTS)	8/8	9/9	0.0%	1/1000
stan499 (NUTS)	12/17	12/14	68.2%	1/1000
stan543 (NUTS)	18/32	14/18	74.5%	500/1000
stan674 (NUTS)	13/24	8/10	NA	1/1000
stan685 (NUTS)	8/13	6/10	NA	1/1000
stan723 (NUTS)	23/32	16/19	NA	1/1000
stan1053 (FP)	13/21	8/11	NA	1/10000
stan1121 (NUTS)	5/16	8/13	0.0%	1/1000
stan1194 (NUTS)	5/5	5/5	NA	1/1000
stan1200 (Opt)	12/21	11/15	99.7%	1/1000
stan1241 (NUTS)	19/28	8/13	NA	1/1000
stan1308 (Diag)	37/196	11/57	100.0%	1/1000
stan1366 (NUTS)	12/16	6/11	NA	1/1000
stan1435 (Opt)	5/20	8/13	0.0%	1/1000
stan1443 (NUTS)	8/13	7/10	NA	1/1000
stan1474 (NUTS)	14/16	10/10	NA	15/1000
stan1610 (VI)	12/70	11/36	98.8%	125/1000
stan1789 (NUTS)	10/23	9/16	NA	1/1000
stan1974 (NUTS)	5/7	3/6	NA	1/1000
stan2188 (NUTS)	9/9	6/6	NA	1/1000
stan2237 (HMC)	3/63	5/25	100.0%	1/1000
stan2294 (NUTS)	9/9	6/6	NA	1/1000
stan2311 (NUTS)	4/18	6/16	NA	1/1000
Avg. Savings	40.89%	32.32%	60.13%	29/1391

issue number or benchmark name for the example-models, and algorithm[NUTS/ HMC/ Variational(VI)/Optimize(Opt)] or mode[Diagnose(Diag)/Fixed-Param(FP)] used to run the program. Column 2 (**SRR**) presents the ratio of the original number of program constructs to the ones in reduced test cases (Section 3.5.3). Column 3 (**LoC**) presents the ratio of original source lines of code to the reduced source lines of code for the test case. Column 4 (**Data Red.**) presents the percentage of the reduced data points (relative to the original size). The cases which did not have any data are marked as NA. Column 5 (**Iters**) presents the reduction of the argument (the number of samples for MCMC, iterations for variational inference) of the approximate inference algorithms. For **SRR**, **LoC**, and, **Data Red.**, we compute the average savings by adding up the savings for each benchmark and dividing by total benchmarks in the set. For **Iters**, we compute the average of original iterations and reduced iterations separately and report the ratio as savings.

From the data in the three tables, we conclude that Storm was able to reduce all 47 test

Table 3.4: Pyro Example-Models Reduced Using Storm

Test	SRR	LoC	Data Red.	Iters.
pf_1 (VI)	138/209	28/37	99.3%	3/4000
pf_2 (VI)	88/207	23/36	99.3%	1/4000
pf_3 (VI)	153/298	30/46	98.6%	3/4000
pf_4 (VI)	153/285	30/44	92.2%	3/4000
pf_5 (VI)	126/270	27/44	98.8%	62/4000
pf_6 (VI)	153/326	30/50	92.1%	3/4000
dyes (VI)	131/331	29/55	96.3%	1/4000
dyes_020 (VI)	129/330	29/56	96.3%	1/4000
ES (VI)	129/250	29/44	95.8%	1/4000
ES_020 (VI)	129/254	29/44	95.8%	1/4000
GP2 (VI)	131/553	29/77	99.9%	1/4000
GP2_020 (VI)	129/557	29/77	99.9%	1/4000
radon (VI)	129/439	29/61	100.0%	1/4000
Avg. Savings	56.58%	42.04%	97.25%	6/4000

cases across all algorithms in at least one of program constructs, number of lines, data, or the number of samples. Storm was able to reduce over 90% in program size, data, and inference arguments (samples in Monte-Carlo simulation and iterations in Variational inference). Out of total 47 programs, 5 improved in one category, 7 improved in two categories and 29 improved in three categories. Tables 3.2 and 3.4 show that with an exception of salm, all larger probabilistic programs are reduced by Storm in all three categories.

Coverage. Table 3.5 presents how many lines of the PP system the original and reduced program cover on average (as measured with gcov for Stan and coverage.py for Pyro). Column 1 (**Benchmark**) presents the group of benchmarks. Column 2 (**Hit**) presents the average number of lines executed by the original programs. Column 3 (**Total**) presents the total number of lines in the PP system. Column 4 (**Cov**) presents the average original line coverage for the benchmarks. Column 5 (**HitR**) presents the average number of lines executed by the reduced programs. Column 6 (**TotalR**) presents the total lines in the PP system. Column 7 (**CovR**) presents the average line coverage for the reduced programs.

Table 3.5: Coverage of Reduced Programs

Benchmarks	Hit	Total	Cov	HitR	TotalR	CovR
Stan Issues	9796	25713	38.07%	8996	25713	34.97%
Stan Examples	10690	25738	41.53%	9844	25738	38.24%
Pyro Examples	7272	24790	29.31%	7224	24790	28.92%

The results show that 1) the number of lines covered by both the original and the

reduced programs is significant and 2) the coverage of the reduced programs, despite of their significantly smaller size is only slightly lower than the coverage of the original programs.

3.6.2 Impact of Probabilistic Transformations

We next study the impact of the newly proposed probabilistic transformations. To do so, we compare the impact of Storm when using both the probabilistic and basic transformations, to a variant that uses only basic transformations. Table 3.6 presents the summary of the results. For each group of benchmarks, we compute and aggregate three statistics from Section 6.6. We compared the savings of that version of Storm to the original (non-reduced) program. Here, Column 2 (**Code**) presents savings in code constructs – SRR. Column 3 (**Data**) presents savings in data items. Column 4 (**Iters**) presents savings in the number of iterations. Note that the basic transformations cannot not reduce the number of iterations. In all cases, the savings are represented as percentages.

Table 3.6: Comparing Storm and Basic Transformations Only

Benchmarks	**Code**		**Data**		**Iters**	
	Storm	Basic	Storm	Basic	Storm	Basic
Stan Issues	40%	36%	60%	46%	97%	0%
Stan Examples	68%	61%	71%	47%	89%	0%
Pyro Examples	56%	53%	97%	82%	99%	0%

The results show that probabilistic transformations improve reduction of both code and data. The reduction in data with basic transformations is due to the *Dead Variable Remover* transformer: when it is possible to remove some data variables from the model, corresponding data-sets can also be removed from the data file without any effect on the model. Even then, we notice that in three cases, probabilistic transformations can further reduce the data-sets.

Probabilistic transformations contributed significantly to the success of reduction – more than 60% of the successful transformations across the three sets of benchmarks were domain-specific transformations. The Storm algorithm accepted 59% of all the domain-specific transformations, compared to 48% of the basic transformations across all benchmarks.

The reduction of inference arguments is unique to Storm with probabilistic transformations. For all the test cases which use sampling algorithms, Storm was able to significantly reduce the number of samples (to 1 in all cases except stan543 and stan1474). For variational inference, the iterations were reduced to 1 in 8 cases (out of 12) for Stan and 8 cases (out of 13) for Pyro. This shows that often the bugs can be revealed quickly by a small number of iterations and can save debugging time for the developer.

3.6.3 Speedup of Reduced Programs

If the developers needs to rerun these tests in regression testing, they can leverage smaller versions of the programs provided by Storm's transformations. Table 3.7 presents the summary of the run-times (without compilation) of the original and reduced programs, run with recent versions of Stan (2.16.0) and Pyro (0.2.1). We only consider the cases where both original and reduced programs pass the test. Column 2 (**TimeO**) presents the average time of the original program. Column 3 (**TimeR**) presents the average time of the reduced program. Column 4 (**Speedup**) presents the ratio of TimeO by TimeR. Overall, the speedup when running the reduced tests is significant, especially for the larger programs with more data (for Stan and Pyro examples).

Table 3.7: Execution time reduction

Benchmarks	**TimeO**	**TimeR**	**Speedup**
Stan Issues	1.02s	0.25s	4.1x
Stan Examples	12.63s	0.10s	126.3x
Pyro Examples	106.36s	0.85s	125.1x

3.6.4 Impact of Transformation Orders

We evaluated whether the order of transformations in each step of the algorithm has a significant impact on the overall reduction of the test cases. We ran Storm on the test cases using six orders described in Section 3.4.3.

Table 3.8 presents the total execution times of the algorithm for these six orders, on a 12-core machine, using all cores for evaluation. Each time is in the format "minutes:seconds". In all cases, Storm converges to the minimal test case, or a test case with very similar quantitative reduction metrics (which we omitted), even in unfavorable orderings. Recall that Storm's algorithm iterates until reaching a fixed point, and unfavorable ordering will most often simply take more steps to terminate.

Table 3.8: Execution Times for Different Orders.

	Fixed	**Rnd**	**Size**	**CoA**	**B-P**	**P-B**
Stan Issues	41:24	45:04	53:33	41:32	55:58	44:27
Stan Examples	47:33	43:26	42:15	46:15	46:32	52:28
Pyro Examples	2:25	7:14	4:52	1:46	2:05	2:18

For Stan Issues, Fixed and CoA orders show the best results. For Stan Examples, Size order is the fastest. We believe the reason is that these programs are more complex than the Stan issues and have more control structures like loops and conditions. Hence, using the transformations which remove the larger blocks early on reduces the run-time of the programs

and thus the reduction algorithm as well. Finally, for Pyro examples, CoA is the fastest. The time of algorithm for Pyro is significantly smaller than for Stan because it interprets the programs, instead of compiling them, like Stan.

3.7 APPLICATIONS OF STORM'S PROGRAM REDUCTION

In this section, we highlight two additional scenarios in which Storm's program reduction technique can be applicable.

3.7.1 Incremental Debugging

A test case can potentially reveal multiple bugs in the system. But during execution, one failure can hide other bugs. A developer then has to go through a cycle of fixing a bug and re-running the test case to find other bugs in the system. Storm can help automate this cumbersome process.

Consider a simplified program for linear regression in Figure 3.5. The program has two data-sets x and y, each of size 10 (lines 2 and 3). In lines 4-6, the parameters w, b, and p, are assigned prior distributions. In line 7, the linear regression model is defined and conditioned on the data variables using observe statement. Lines 8-10 contain queries for posterior distribution for each parameter. The original program fails when run with Pyro 0.1.2 with the error *"Domain error in arguments"*, which does not clearly indicate the cause of the failure. When we run this program with Storm, it is reduced to the program from Figure 3.6. The data-set y now has only 1 element and the observe statement has a simple distribution and data variable y. Now, it is quite easier to figure out that the value in y (-2.99) is outside the range of support for beta distribution $(0, 1)$. If we look at the original program, we can observe that the values in y were outside of the support of lognormal distribution, $(0, \infty)$.

Even after this issue is resolved (for instance, by changing *lognormal* distribution to *normal*), the program fails with the same error. Using Storm again, we reduce the fixed original program (Figure 3.5) to the snippet in Figure 3.7, which has only one parameter with exponential distribution. The bug here is the negative value in exponential distribution, which expects a positive value. The original program had the same issue at Line 5. Storm takes only 74 seconds to find the error-inducing lines in each round.

```
1  N: 10
2  x: [ 72.97, 34.94, ...]
3  y: [ −2.99, 1.95, 2.77, ...]
4  w:= exponential(37.47)
5  b:= exponential(−31.49)[N]
6  p:= lognormal(55.43,61.35)[N]
7  observe(lognormal(w∗x+b, p), y)
8  posterior(w)
9  posterior(b)
10 posterior(p)
```

Figure 3.5: Original Program

```
1 N:1
2 y: [−2.99]
3 p:= gamma(1.0,1.0)[N]
4 observe(beta(1.0, 1.0), y)
5 posterior(p)
```

Figure 3.6: Minimized Prog. 1

```
1 N:3
2 b:= exponential(−31.49)[N]
3 posterior(b)
```

Figure 3.7: Minimized Prog. 2

3.7.2 Using Coverage as a Criterion

We explore the generality of Storm through a case study where we change the reduction property (ψ) to preserve the line coverage of the PP system (Stan) under test i.e. the coverage of the reduced program is the same as the original program. We used *lcov* to measure the line coverage after each transformation. We also turned off transformations that do not always reduce code and might cause the transformed program to execute a different function (e.g., Distribution Simplifier may replace a distribution with another distribution). Overall, Storm reduced the code constructs by 30.4% and data by 22.5%.

3.8 THE STORM FRAMEWORK AND ITS APPLICATIONS

The Storm-IR language, presented in this chapter, is a part of the Storm framework - a unified platform for testing, transformation, and analysis of probabilistic programs. At its core, Storm consists of four main components: the Storm-IR language, a Translator, a Transformer, and a Static Analysis engine. The key advantage of Storm is the common intermediate language, which allows the programmer to write any kind of analysis/transformation once, and then translate it into any probabilistic programming language of their choice. This rids the programmer from having to deal with the intricacies of individual PPSs.

The Storm framework currently supports seven different probabilistic programming languages: Stan, Pyro, PyMC3, Edward2, TensorFlow-Probability, NumPyro, and PSI. Storm provides various static analyses like dimensional analysis, interval analysis, def-use analysis, and reaching definitions analysis. The analyses in Storm employ domain-specific techniques like observe dependency tracking and utilize the properties of distributions to extend the approaches used for conventional programs. Storm also provides over 20 program and data transformations including both distribution and distribution parameter transformers, a constant to parameter transformer, a data to parameter transformer, a data rescaling

transformer, and a transformer for applying continuity correction to conditionals, among others.

We present research works (other than the program reduction presented in this chapter) that have leveraged the Storm framework for debugging, analyses, and inference of probabilistic programs.

3.8.1 Debugging Convergence Problems in Probabilistic Programs

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 developed SixthSense [210] – an ML-based approach [211, 212, 213] 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.

SixthSense uses the Storm framework to represent probabilistic programs. 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 developed 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 pro- grams. 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 [212] and Code2Seq [213]. SixthSense can guide the debugging of convergence problems, which pinpoints the causes of non-convergence significantly better by Stan's built-in warnings.

3.8.2 Automated Quantized Inference for Probabilistic Programs

We developed AQUA [214, 215, 216] – a new probabilistic inference algorithm that operates on probabilistic programs with continuous posterior distributions. AQUA approximates programs via an efficient quantization of the contin- uous distributions. It represents the distributions of random variables using quantized value intervals (Interval Cube) and

corresponding probability densities (Density Cube). AQUA's analysis transforms Interval and Density Cubes to compute the posterior distribution with bounded error. AQUA used the Storm framework to represent probabilistic programs and to implement all analyses for inference.

We evaluate AQUA on 24 programs from the literature. AQUA solved all of 24 benchmarks in less than 43s (median 1.35s) with a high-level of accuracy. We show that AQUA is more accurate than state-of-the-art approximate algorithms (Stan's NUTS and ADVI) and supports programs that are out of reach of exact inference tools, such as PSI and SPPL.

3.8.3 Studying Robustness of Probabilistic Programs

Robustness is the property of systems, including probabilistic programs, to remain unaffected by data noise. To help users understand both the practical and fundamental properties of probabilistic robustness transformations, we developed the ASTRA framework [217]. ASTRA automatically modifies the program code to apply the robust transformation (and check for its legality) and systematically evaluates different robustness transformations for user-defined input noise models and posterior accuracy metrics. ASTRA then ranks the transformed programs by predictive accuracy.

ASTRA uses Storm-IR to represent probabilistic programs and implements all the robustness program transformations using the Storm framework. Using ASTRA we studied various robustness transformation in literature on a diverse set of benchmarks. Our experimental results indicate that the existing trans- formations are often suitable only for specific noise models, can significantly increase execution time, and have non-trivial interaction with the inference algorithm.

3.9 THREATS TO VALIDITY

Internal. Our Storm implementation may contain bugs, some bugs may have been miscategorized during our selection and reproduction, and we may have made wrong conclusions about some minimized programs. To mitigate the risk of implementation bugs, multiple co-authors conducted a code-review of Storm and test-cases.

External. Storm methodology may not generalize to all PP system. However, there are three aspects which help mitigate the risk. First, we present the evaluation on two commonly used languages with different design, Stan and Pyro. Second, we looked at historical bugs, which may provide a good guide for the kind of bugs that may appear in the future. The maturity of Stan and the development effort in Pyro increase confidence that these bugs are

representative of probabilistic programming in general. Third, our design on Storm minimizes the dependence on the language – most of the transformations are generic and can be applied to other languages.

3.10 RELATED WORK

Test Reduction. C-Reduce [199] reduces test cases for C programs (often generated using CSmith [87] test generator). C-Reduce uses source-to-source transformations customized for C-like programs, but its application to the domain of probabilistic programming is not straightforward. We also show the importance of domain-specific (probabilistic) transformations for successful program reduction, and make a parallel to CSmith and C-Reduce, by showing how Storm can reduce the programs generated by Probfuzz [196]. Other approaches for program reduction include Delta debugging [200], which is one of the earliest known techniques for test reduction. It removes parts of the failing test (code or data) until no single part can be removed without the test passing. Hierarchical Delta Debugging (HDD) [201] applies DD using the structure of the input. HDD generates fewer syntactically invalid programs but provides no guarantee. In contrast to these approaches, Storm produces only syntactically-correct reduced tests.

Perses [202] is a recent language-agnostic framework for reducing programs in conventional programming languages (e.g., C and Java). Like Perses, Storm uses the syntax of the language to guide the reduction process, applies the transformations on the intermediate representation, and generates syntactically valid programs. Storm strengthens the reduction process using various kinds of static analyses including dimensional and type analysis. Storm also uses probabilistic transformations to reduce data and inference parameters, which improves the overall reduction quality beyond the reach of general reduction frameworks.

Zhang et al. [218] proposed a technique for test simplification that is also able to modify portions of a test by replacing expressions with those already existing in the test. Other approaches [219, 220, 221] provide domain specific transformations to produce minimal structured data sets and reduce size of tests. To the best of our knowledge, we are the first to present domain-specific transformations and minimization for the probabilistic programming domain.

Verification and Analysis of Probabilistic Programs. Previous research proposed various techniques for statically analyzing and verifying properties of probabilistic programs, e.g., [148, 149, 150, 151, 152, 153, 154], including analyses that aim to help with debugging, e.g., [157, 222, 223]. In comparison, Storm presents a dynamic analysis that performs a heuristic search for smaller probabilistic program that reveal the same bugs.

Program slicing [224] is a standard technique for removing unnecessary parts of the code. Researchers recently extended slicing to probabilistic setting [225, 226]. Similarly, refactoring is a general set of techniques that modify program source code while preserving the program semantics, yet improve the program's internal structure [227, 228]. In contrast, Storm reduces program while ensuring that only bug manifestation remains, and has the freedom to change program semantics.

3.11 SUMMARY

This chapter presented Storm, a novel approach for reducing probabilistic programs. Storm presented both basic program reduction transformations, driven by program analysis and domain-specific probabilistic techniques to minimize the size and complexity of bug-revealing probabilistic programs. We evaluated Storm on 47 bug-revealing programs from two state-of-the-art PP systems, Stan and Pyro. For Stan, our minimized programs have 49% less code, 67% less data, and 96% fewer iterations. For Pyro, our minimized programs have 58% less code, 96% less data, and 99% fewer iterations.

Chapter 4: EMPIRICAL STUDY OF RANDOMNESS IN REGRESSION TESTS IN MACHINE LEARNING LIBRARIES

4.1 INTRODUCTION

The extensive success of Machine Learning has led to its widespread adoption across several critical domains such as autonomous driving, natural language processing, and medical diagnosis. These domains implement applications that utilize different ML algorithms such as Deep Learning [1], Reinforcement Learning [2], and Probabilistic Programming [3, 4]. Consequently, this has led to the development of a rich ecosystem of Machine Learning libraries and tools that solve tasks at varying levels of specializations. However, bugs in the implementation of these tools can lead to catastrophic consequences for the end users, often amounting to loss of lives and property [9, 10].

Testing implementations of ML algorithms is challenging. Many ML algorithms are *inherently random* in nature – multiple executions of the algorithm with same inputs and configurations may often lead to varying results. Moreover, the lack of proper test oracles further complicates the testing scenario. A natural consequence of randomness is *test flakiness*, i.e., when a test passes and fails non-deterministically for the same version of code. Test flakiness undermines the reliability of test results and puts additional burden on developers for investigating test failures (even in absence of bugs). Flakiness makes it difficult for developers to distinguish real test failures due to programming errors (*bugs*) from noisy executions due to randomness.

To minimize test flakiness, developers need to make various non-trivial choices such as choosing the optimal hyper-parameters for the ML algorithm under test [229] and a reasonable assertion bound [230]. However, without a systematic approach, it is hard for a developer to get these settings right. Hence, to mitigate flakiness, they tend to set the *seeds* for the *random number generators* that are used by the code under test. Setting the seeds can make the test execution deterministic and alleviate the developer from dealing with randomness. However, it is unknown whether this is *always* the best approach or if there are alternative ways to mitigate flakiness. Setting seeds can also lead to unintended consequences. For instance, fixing the seed(s) limits the sequence of computations exercised by the code under test. Hence, developers may potentially miss bugs in code under test that are triggered by other sequences, thus reducing the fault-detecting effectiveness of the test [53].

Prior work [53] has shown that algorithmic randomness is a major contributor to flakiness in ML projects. Further, it is known that tests for ML algorithms (prone to flaky failures) are typically more time-consuming than other tests in the suite, often consuming more than

80% of test time [229]. This makes it important to study such tests – and the role of seeds – in greater depth and scale than previous works.

In this chapter, we conduct the first large-scale and systematic study of the usage of seeds (for random number generators) and its implications for testing in Machine Learning projects. We study several research questions and provide insights that can be useful for both developers and researchers: 1) How prevalent are tests that fail non-deterministically without seeds and how often they fail? (Section 4.4), and 2) What are the common characteristics of these failing tests? (Section 4.5).

We conduct an empirical study on a corpus of 114 Python projects from the Machine Learning domain. We develop a tool, XSEED, which automatically installs each project, runs each test a pre-specified number of times (500 in our evaluation) in two modes: *with seeds* and *without seeds*, and generates a report summarizing the failed tests, the different types of failures, and the failure rates. Overall, we find 461 unique tests across 32 projects that fail when seeds are removed but always pass when seeds are present.

We further analyze a subset of 56 tests, discuss and categorize various characteristics such as nature of test oracles, source(s) of randomness, evolution of seeds relative to the tests, and how the seeds are set. We also provide a general set of *recommendations* based on our experience for using (or not using) seeds (more details in Section 4.5):

- Use *fixed* seeds only for tests checking for exact reproducibility of some functionality in their code.
- Randomize and log the seeds in other tests for non-deterministic algorithms to allow both reproducibility of failures and diverse executions.
- Use Test Re-runs on failure instead of setting seeds to mitigate random failures.
- Determine optimal test settings such as hyper-parameters for the algorithm(s) under test and/or assertion bounds to minimize flakiness.

Contributions. We make the following contributions:

- We conduct the first large-scale empirical study of the usage of seeds in tests in 114 Machine Learning projects.
- We analyze the tests that fail without seeds and study important aspects related to the nature of such tests and the root causes for flakiness.
- We provide several insights and implications related to usage of seeds and a general set of recommendations for both developers and researchers.

Our source code and replication package are available at `https://github.com/uiuc-arc/xseed`.

4.2 BACKGROUND

We describe previous research on flaky tests in Machine Learning projects.

4.2.1 Common Test Structure in Machine Learning Projects

```
1 def test_MLAlgo():
2     [[setup code]]
3     trainer = MLAlgo( P_1 = v_1 , P_2 = v_2 , ... , P_k = v_k )
4     trainer.train()
5     metrics = trainer.compute_metrics()
6     for i in range(len(metrics)):
7         assert metrics[i] >= expected[i]
```

Listing 4.1: Common Test Pattern in ML projects

The tests that developers write for testing the correctness of their implementations of stochastic Machine Learning algorithms typically emulate the training (or fitting) process. Listing 4.1 presents the common structure of such tests, previously identified by Dutta et al. [229]. In this test, Line 2 contains setup code that performs basic initialization steps such as loading data-set(s), creating the execution environment, or setting up other configurations (such as seeds). Line 3 initializes the Machine Learning algorithm (`MLAlgo`) using a set of values $(v_1, \ldots, v_k)$ for arguments $(P_1, \ldots, P_k)$ known as *hyper-parameters*. These hyper-parameters influence both the accuracy and performance of the ML Algorithm. Lines 4-5 then perform the training step and compute one or more accuracy or performance metrics. Lines 6-7 check if the computed metrics (`metrics[i]`) are greater or equal to expected values (`expected[i]`). In our study, we also find that most tests exhibit similar structure.

4.3 METHODOLOGY

4.3.1 Selection of projects

For our study we require projects that test various stochastic/non-deterministic algorithm implementations. Such projects are more likely to use seeds during testing to avoid flakiness. Hence, we select projects from the domains of Machine Learning and Probabilistic Programming. For selecting projects in these domains, we follow a similar methodology as Dutta et al. [230]. We select two Machine Learning frameworks: PyTorch [77, 231] and

TensorFlow [78], and four Probabilistic Programming libraries: Pyro [191], NumPyro [232], TensorFlow-Probability [233], and PyMC3 [234]. We search for Python projects that depend on these six main libraries. For this task, we use GitHub's API to search for the dependent projects. We only select projects that can be installed as a Python library (known as "packages") and have at least 10 stars on GitHub. This allows us to eliminate toy projects and select projects that are more likely to have good test suites, relatively more popular, and have an active developer base. To limit our study to a reasonable number of projects, we only select top 100 dependent projects per library for our study.

Using this methodology, we selected 305 unique projects. We use a general installation script to install these Python libraries [230]. This script installs a general set of system-level packages. It processes the project files and creates a list of required dependencies for the project. It then creates a virtual Python environment using Anaconda [235] and installs the project and all its dependencies. However, in many cases the installation may fail due to incomplete specifications in the project files. For each project, we try to install the project using this script and check if we can run the tests successfully using `pytest`. Overall, we were able to run 114 projects.

4.3.2 Running and detecting flaky tests

In this work we aim to study tests that are affected by the seeds set for various random number generators that are used by the code under test. To find such tests, we run the existing tests in each project in two modes: *with seeds* (using the original version of the test) and *without seeds* (by removing all seed-setting statements). We then identify the tests that always pass when run with seeds but fail (at least once) without seeds.

To automate this task, we developed a tool: XSEED. XSEED takes as input the GitHub slug of the project, the number of times (N) to run each test, the number of threads to run in parallel (K), and a timeout (T) for executing the entire test-suite. XSEED then performs the following tasks. XSEED installs the project in a new Conda environment using the setup script described in Section 6.5. It runs each test N times in the project using `pytest` and collects the test execution logs. XSEED parallelizes the runs by using K threads, with each thread running all the tests in the default order. XSEED uses the specified timeout T to limit the maximum allowable running time for the test-suite (i.e., for a single thread).

XSEED then searches for all seed-setting code in the project and replaces them with *pass* (equivalent of *skip* in Python). In particular, XSEED searches for the API calls across various libraries that provide random number generators. Some libraries like PyTorch and TensorFlow provide multiple APIs to set seeds. Such APIs may also change across library

versions. Hence, XSEED searches for invocations to all such APIs in the project. Table 4.1 presents the list of all APIs used by XSEED. XSEED then re-runs the tests using the same settings but the seeds *removed* and collects the execution logs.

XSEED parses the test execution logs from the runs with and without seeds and returns a summary report containing the list of tests that failed and frequency of each kind of failure (e.g., `AssertionError`, `ValueError`) per test per project. We use this report for further analyses.

Table 4.1: Seed Setting APIs

Library	API
Numpy	`random.seed`
TensorFlow	`random.set_seed`
TensorFlow	`set_random_seed`
TensorFlow	`random.set_random_seed`
TensorFlow	`compat.v1.random.set_random_seed`
PyTorch	`manual_seed`
PyTorch	`cuda.manual_seed_all`
PyTorch	`seed`
Random (Python)	`seed`

4.3.3 Analyzing results

We select a subset of tests that always pass with seeds but fail at least once without seeds for manual analysis. For each selected test, we try to identify various characteristics such as the source of randomness, nature of the code under test, and how the seeds are set. We also study historical features of each test such as when were the seed(s) set relative to the test creation date and how often were the test settings (such as seeds or assertions) changed by the developers. For each studied characteristic, we determine appropriate categorizations.

For this analysis, one author independently analyzes each test and determines appropriate categorizations. Then another author double-checks each test and its categorizations to mitigate any inaccuracies. Finally, the authors discuss together and collate all the results. We discuss the results of this analysis in Section 4.6.

4.3.4 Research Questions

We address the following research questions in this work.

RQ1 How many and how often do tests fail without seeds?

RQ2 What kind of assertions are used in the failing tests?

We address RQs 1-2 in Section 4.4.

4.3.5 Experimental Setup

We run all experiments on Azure machines (`Standard_F32s_v2` configuration) with 3.4GHz Xeon CPUs with 32 cores and 64GB memory. For each project, we run XSEED with 20 threads (K) in parallel. We run each test-suite 500 times (N) with a timeout of one hour (T) for each complete test-suite run.

4.4 EMPIRICAL RESULTS

4.4.1 RQ1: Test behavior without seeds

We run the tests in 114 projects both with and without all seed-setting code using XSEED (Section 4.3.2). Out of these, in 30 projects the tests timed-out. We exclude those projects from our study and use the results from the remaining 84 projects. Table 4.2 presents the details of results for a subset of these projects. Each row of this table represents one project. Column **Project** is the name of the project. Column **#Tests** is the number of tests in each project. Column **Total Failures** is the total number of tests that failed (at least once out of 500) in the project with or without seeds. Sub-column **ws** is the number of failed tests when run with seeds. Sub-column **wos** is the number of failed tests when run without seeds. Sub-column **wos-uniq** is the number of tests that always passed with seeds but failed at least once with seeds. Table 4.2 only presents the results of projects with at-least one such failing test. Overall, there are 32 such projects. In 52 projects, we did not find any additional test failures when seeds are removed.

Columns 6-10 present the failure rate statistics for tests in **wos-uniq** category. Column **[0%, 5%)** is the number of tests with failure rate between 0-5 (exclusive)%. Similarly, columns **[5%, 10%)**, **[10%, 50%)**, and **[50%, 100%)** are the number of tests with failure rates of 5-10%,10-50%, and 50-100% respectively. Column **100%** is the number of tests with a failure rate of 100%. The second last row presents the totals per column for 32 projects. The last row presents the summary for all 84 projects that did not time-out. Columns 6-10 in last row present the breakdown of all failing tests without seeds (2226).

Overall, we observe that 1800 tests fail when run with seeds whereas 2226 tests fail when seeds are not used across all 84 projects. In the smaller subset of 32 projects, the number of

Table 4.2: Running Tests With and Without Seeds

Project	#Tests	Total Failures			[0%, 5%)	[5%, 10%)	[10%, 50%)	[50%, 100%)	100%
		ws	wos	wos-uniq					
Accenture/AmpliGraph	86	11	11	8	3	1	1	1	2
quantumlib/Cirq	10101	27	49	22	3	0	11	7	1
GPflow/GPflow	1003	0	13	13	13	0	0	0	0
ziatdinovmax/GPim	7	0	1	1	0	0	0	0	1
google/TensorNetwork	9224	37	186	149	7	1	4	0	137
SeldonIO/alibi-detect	1166	12	34	25	18	2	5	0	0
bambinos/bambi	88	16	33	17	15	2	0	0	0
pytorch/captum	769	0	102	102	23	9	31	30	9
thinkingmachines/christmAIs	32	5	5	1	1	0	0	0	0
autorope/donkeycar	65	3	5	2	2	0	0	0	0
google/dopamine	137	11	13	2	0	0	0	0	2
RaRe-Technologies/gensim	0	12	13	4	2	0	2	0	0
tensorflow/graphics	2200	17	18	1	0	0	0	0	1
learnables/learn2learn	57	6	1	1	1	0	0	0	0
magenta/magenta	354	4	6	2	2	0	0	0	0
Unity-Technologies/ml-agents	36	17	18	1	1	0	0	0	0
uber/orbit	246	0	25	25	0	0	0	0	25
josejimenezluna/pyGPGO	13	0	1	1	0	1	0	0	0
quantopian/pyfolio	80	8	9	1	0	0	1	0	0
exoplanet-dev/pymc3-ext	93	6	11	5	4	1	0	0	0
pymc-devs/pymc4	1334	20	24	4	4	0	0	0	0
jettify/pytorch-optimizer	346	0	25	25	23	1	1	0	0
tensorflow/ranking	502	23	24	2	0	0	0	0	2
refnx/refnx	227	6	16	10	1	0	0	0	9
datamllab/rlcard	208	5	6	2	1	0	0	0	1
YosefLab/scvi-tools	69	4	5	1	1	0	0	0	0
snorkel-team/snorkel	250	22	32	10	2	4	3	1	0
danielegrattarola/spektral	90	7	4	2	1	0	1	0	0
autonomio/talos	8	3	3	1	1	0	0	0	0
explosion/thinc	21	1	4	4	0	0	0	0	4
EpistasisLab/tpot	7	0	1	1	0	0	0	0	1
lmcinnes/umap	139	1	17	16	14	1	1	0	0
Total/Avg (32 projects)	28958	284	715	461	143	23	61	39	195
Overall (84 projects)	43783	1800	2226	461	236	49	82	54	1805

such tests are 284 and 715 respectively. 461 tests fail in these projects when the seeds are removed but always pass with seeds. Out of these tests, 195 tests consistently fail (i.e., 500 failures out of 500 runs) whereas 266 tests are *flaky* (i.e., they non-deterministically pass or fail). Out of the failing tests, 227 of them have a failure rate of less than 50% whereas 39 of them have a failure rate of more than 50% (but less than 100%). These results show that a significant number of tests depend on seeds set in their random number generators to control the randomness during testing and avoid test failures.

Common Failure Types. We observe that the majority of tests fail due to `AssertionError` (394 out of 461), which is expected since most of these tests contain approximate assertions that compare the result(s) of non-deterministic computations against a fixed value or range. When the seeds are removed, such assertions are more likely to fail. Other common failure types include `ValueError`(40) and `FileNotFoundError`(17). We present a detailed analysis of the tests that fail due to assertion errors in Section 4.5.

Tests with 100% failure rates. We observe that 195 tests fail 100% of the time across 12 projects. We investigate a subset of such tests and determine the most common causes:

- *Exact reproducibility*: Some tests check if two successive calls to the same/similar API produce the *exact* same result when starting from the same seed. Such tests are intended to test whether certain computations that depend on random number generators are reproducible.

  Listing 4.2 shows an example of such a test in *google/TensorNetwork*. TensorNetwork [236, 237] is a library that provides implementations of high dimensional data-structures (called Tensor Networks) used in domains like quantum mechanics. TensorNetwork provides a wrapper API for backends like Numpy so that it can handle custom data types defined in TensorNetwork. In Listing 4.2, Line 2 initializes the Numpy backend. Lines 3 and 4 then make two successive calls to the `randn` API to create a random matrix of size 4×4 for the specified data type (`dtype`). Both use the same seed (10). Line 5 checks whether two matrices are close up to a certain precision level. The `randn` API internally calls *Numpy* random number generator. Hence, the test checks whether generating two matrices starting from the same seed and data type are equal. Evidently, this test is likely to almost always fail if the seeds are removed since the probability of producing the same random matrix in successive calls is very low.

- *Testing for exact equality*: Since seeds make the end result deterministic, developers sometimes add assertions checking whether the final result is *exactly* equal to the expected value. However, without seeds, the computations could lead to slightly different results that causes these assertions to fail. In these scenarios, the test may be fixed by using an approximate assertion that checks whether the result is *close enough* to the expected values.

- *Too strict assertion bounds*: Developers may sometimes specify a very strict assertion bound/tolerance level in their tests that only works when specific seeds are used but not for most cases when the data/sequence of computations are non-deterministic. Such tests may be fixed by choosing a *looser* bound or tolerance level.

```
1 def test_randn_seed(dtype):
2     backend = numpy_backend.NumPyBackend()
3     a = backend.randn((4, 4), seed=10, dtype=dtype)
4     b = backend.randn((4, 4), seed=10, dtype=dtype)
5     np.testing.assert_allclose(a, b)
```

Listing 4.2: Example test in *google/TensorNetwork*

Table 4.3: Distribution of Assertions Used in Failing Tests

API Name	Framework	# of Assertions
assert	Python	137
assert_allclose	Numpy	112
assertAlmostEqual	Unittest	94
assertTrue	Unittest	13
assert_almost_equal	Numpy	12
assert_array_almost_equal	Numpy	9
assertEqual/assertNotEqual	Unittest	6
assertGreater/assertGreaterEqual	Unittest	5
assertLess/assertLessEqual	Unittest	5
assertEqual/assertAllEqual	TensorFlow	3
assertAllClose	TensorFlow	2
assertAlmostEqual	TensorFlow	1
assertIn	Unittest	1
assertLen	TensorFlow	1
assert_	Numpy	1
assert_array_equal	Numpy	1

- *Bug*: A test that always fails without seeds may also be indicative of a bug. We identify one such scenario in *google/TensorNetwork* [238] where the test often produces an empty array as the end result whereas the expected result is an array of size 1.

Tests with low failure rates. We observe that several tests have low failure rates: 143 tests have a failure rate of less than 5%, 23 tests have a failure rate of 5-10%. The low failure rates indicate that many of these tests can likely be fixed using minor adjustments such as adjusting some hyper-parameters (like iterations) or by updating assertion bounds by a modest amount.

Insights and Implications. We discover a large number of tests that fail without seeds, many of which have high failure rates (195 tests always fail). This indicates developers often use seeds to suppress highly unstable tests instead of properly *fixing* them. The only exception to this are tests that check for exact reproducibility where setting seeds is necessary.

4.4.2 RQ2: Assertions used in the failing tests

Table 4.3 presents the number of each kind of assertion that are used in the tests that failed without seeds. For this analysis, we only consider the tests that failed due to an assertion failure. Some tests may have more than one type of assertion that fails, hence the total number of assertions is more than the number of failing tests.

```
1 def test_model_loss(self):
2     label_model = LabelModel(cardinality=2, verbose=False)
3     label_model.fit(data, n_epochs=1)
4     init_loss = label_model._loss_mu().item()
5     label_model.fit(data, n_epochs=10)
6     next_loss = label_model._loss_mu().item()
7     self.assertLessEqual(next_loss, init_loss)
```

Listing 4.3: Test Oracle: Comparison against Same Model

Overall, the most common assertion is Python's `assert` that is used across 137 tests. The developers also use assertion APIs provided by other frameworks like *numpy* (5 APIs used across 135 tests), *unittest* (9 APIs used across 123 tests), and *tensorflow* (5 APIs used across 7 tests).

Insights and Implications. We observe that the failing tests use both approximate (such as Numpy's assert_allclose) and exact equality assertions (such as Unittest's assertEqual). This indicates that developers need to transform both kinds of assertions to reduce flaky failures. For instance, they can change exact equality assertions into approximate ones. On the other hand, they can lower the strictness of approximate assertions (e.g., by reducing precision level).

4.5 ANALYSIS

We select a subset of 56 tests across 21 projects that only failed without seeds, for deeper analysis. We describe the various categories for each characteristic that we analyze.

4.5.1 Nature of Test Oracles

We characterize the nature of oracle used in the tests:

Comparing against same model but different state or configuration

This category includes the tests that compare the results of running the same model with and without some changes. For instance, Listing 4.3 shows the test `test_model_loss` in *snorkel-team/snorkel*. This test fits the same model (`LabelModel`) twice on the same data-set `data` and checks if training loss after 10 epochs is less than that after 1 epoch (Line 7). Overall, 7 tests fall in this category.

```
1 def test_tSP_opt_nograd():
2     tsp = tStudentProcess(squaredExponential())
3     tsp.fit(X, y)
4     assert 0.3 < tsp.params['l'] < 0.5
```

Listing 4.4: Test Oracle: Comparison against Fixed Values

Comparing against different model

Tests in this category compare the results of running a model against a different kind of model (baseline). Overall, 5 tests belong to this category.

Comparing against fixed values

These tests compare the results of training or fitting a model against a fixed value or value range. Listing 4.4 shows such a test in *josejimenezluna/pyGPGO* that fits a model (`tStudentProcess`) on a dataset `(X,y)` and checks if the fitted parameter (`l`) fall in the specified range (Line 4). Overall, 44 tests fall in this category.

Insights and Implications. We observe that majority of the failing tests (44) compare against fixed values. This implies that developers often find it difficult to choose these values (also known as assertion bounds) which in turn forces them to use seeds in their tests to avoid flaky failures. Hence, developers should be more careful when choosing assertion bounds or use tools like FLEX to automatically find optimal values.

4.5.2 Introduction and Evolution of Seeds Relative to Tests

We look into commits between when the test was added and when it was last modified and study the evolution of seeds relative to the test. Overall, we find that for 23 tests, seeds were introduced in the same commit as the test. For 16 tests, the seeds were added after the test. In 17 tests, seeds were present before the tests were added. Further, in 20 tests, developers also modified the seeds in later commits.

Insights and Implications. We observe that developers often modify the seeds they set. This indicates that setting seeds may not always be the most reliable way of mitigating flakiness.

4.5.3 Sources of Randomness

We categorize the tests based on the source of randomness. In 19 cases, the randomness is only due to the algorithm under test. In 33 cases, the randomness is only due to generation

of random data. In 4 cases, randomness is due to both.

Insights and Implications. We observe that the nature of the source of randomness does not have a strong correlation with flakiness. Rather, other test settings such as hyper-parameters or assertion bounds have a stronger impact on flakiness.

4.5.4 Seed Setting Location

Developers set seeds in tests in different ways, which effects the execution of tests differently. For instance, developers can set seeds at the *global* level, i.e., before executing all tests. Developers can set the seed at *module* level (or file level), i.e., before running tests in a file, or at *class* level, i.e., at the beginning of test class, or at *function* level, i.e., inside the test method. Out of 56 tests we analyze, developers set seeds at global level in 7 cases, at module level in 3 cases, at class level in 17 cases, and at test level in 29 cases.

Insights and Implications. We observe that developers mostly prefer setting seeds at the test level, which minimizes chances of flaky failures. Setting seeds at class level is useful when the class initialization code involves generating some random data that is shared among the tests in the class. Setting seeds at higher levels (module or global) can potentially introduce implicit order dependencies between tests such that tests only pass for a specific set of orderings but fail for others. Future work may explore this aspect of setting seeds.

4.6 DISCUSSION

4.6.1 Should Seeds be Used in Tests?

Based on our experience with the tests and projects that we study, we develop a set of general recommendations or best practices for using (or not using) seeds for testing.

When to use fixed seeds. Developers should ideally use seeds when testing for *exact reproducibility* of some functionality in their code. For instance, this may include APIs that implement functionality/wrappers related to random number generators (such as Listing 4.2 from *google/TensorNetwork*). Another example is a test in *TensorFlow/ranking* [239] that tests a sorting algorithm that randomly shuffles ties. The test uses two different seeds to check whether the algorithm outputs two sequences with two different orderings of tied elements.

Randomize and Log seeds for variability and reproducibility. During our discussions with developers, some mentioned that they use fixed seeds in their tests for *better reproducibility* of test failures. A better approach might be to *randomize* and *log* the seed. This would ensure that the code under test exhibits different sequences of computations and test

failures can still be reproduced. Interestingly, we find one such example in *pytorch/serve* [240]: `random.seed(datetime.datetime.now())`. However, the test may also randomly fail (not due to a bug). We next discuss a strategy for mitigating this risk.

Use Test Re-run on failure instead of setting seeds. Instead of setting seeds, developers can choose to re-run the test on failure (e.g., using Python's *flaky* plugin [241]). This has a few distinct advantages: 1) CI builds will not be blocked due to intermittent failures, reducing the burden on developers, 2) intermittent failures can still be logged allowing developers to investigate them later, and 3) if test still fails after re-run(s), developers can use it as signal for immediate investigation. The expected cost of re-runs will be low if the test rarely fails [229].

Finding optimal test settings to minimize flakiness. In Chapters 5 and 6, we will introduce techniques that will allow developers to find both optimal hyper-parameters and assertion bounds for their tests. Future research can perhaps look into making these tools more accessible and cost-efficient for developers so that they can easily integrate them into their workflow.

4.6.2 Impact of Seeds on Fault-Detecting Ability

Setting seeds minimizes the randomness in the test and consequently the chance of flaky failures. But it can also limit the ability of the test to detect faults that are only exposed by a subset of potential sequences of random numbers. In this work, we found such an instance in *google/TensorNetwork* [238] project, where the fixed seed was hiding a truncation issue in the code under test . On removing the seed, the test failed 519 out of 1000 times exposing the issue. Dutta et al. [53] also reported a similar observation in *geomstats/geomstats* project where the test fails in 42 out of 1000 runs only when the seed is removed and exposes a bug. These instances show seeds can seriously impact the fault-detecting ability of tests. Hence, developers must be careful when dealing with randomness in tests and also consider alternative strategies.

A more comprehensive analysis of fault-detecting ability of the tests with and without seeds can be done using techniques such as mutation testing or by leveraging historical bugs – similar to the methodologies described in TERA [229] (Section 6). This is however beyond the scope of this work.

4.6.3 Threats to Validity

The projects that we use for our empirical study only contain a subset of all machine learning projects. Hence, our results may not generalize well beyond the projects we study. To mitigate this risk, we start with popular ML and probabilistic programming frameworks and select their dependent projects that are also fairly popular (have at least 10 stars). Using this approach we find a large number of projects where seeds are used and tests that are affected when those seeds are removed. Hence, we believe that our results are representative.

Our analysis of tests may contain potential miss-categorizations. To minimize this risk, two authors of the work jointly analyze the tests and determine the correct characterizations after mutual discussions.

4.7 RELATED WORK

Flaky Tests. Luo et al. [50] conducted the first systematic study on flaky tests. They studied flaky tests in Java open-source projects and discovered common causes and fixes. Researchers have also studied flaky tests specific to Python [51], Android [242], and Embedded Systems [243]. Researchers have developed techniques to detect flaky tests of specific kinds such as ones due to test-order dependencies [244, 245], concurrency [246], unordered collections [247], and asynchronous wait [248, 249]. Dutta et al. [53] conducted the first study of flaky tests in Machine Learning projects. They observed that the major cause of flakiness in this domain is due to algorithmic randomness. They developed FLASH [53] to detect such flaky tests. Researchers have developed various techniques for fixing flaky tests due to test-order dependencies [55], unordered collections [56], asynchronous waits [249], and algorithmic randomness [230].

Prior works on flaky tests in Machine Learning projects have made brief observations regarding usage and influence of seeds on testing [53, 229, 230]. However, their observations have been mostly limited to small number of projects and they do not directly address the problem of setting seeds or its general impact. In this work, we present the first large-scale study on how seeds are used and how they impact testing in this domain.

Testing non-deterministic or approximate software. Many emerging systems in domains such as ML and Probabilistic Programming exhibit non-deterministic behavior. These systems require specialized testing techniques to detect deep faults. Researchers have proposed methods for testing and debugging various non-deterministic systems such as ML frameworks [12, 13, 250, 251, 252, 253], Probabilistic Programming Systems [196, 210, 254, 255], and Approximation Algorithms [39]. On the other hand, Hariri et al. [142] proposed

approximate transformations (e.g., loop perforation) for mutation testing of Java projects. They observed that such mutations can sometimes generate valid approximations (leading to surviving mutants) due to the presence of *approximable code*.

4.8 SUMMARY

We identified 461 tests across 114 projects that are flaky but are hidden due to developer-set seeds. This demonstrates that setting seeds is a common *workaround* used by many developers. Our study motivates the need for alternative strategies for fixing such tests. In the following chapters we will introduce techniques that will allow developers to properly *fix* such tests and mitigate flakiness. We hope our study and insights will motivate developers in writing better tests and researchers in improving the fixing techniques and making them more accessible to developers.

Chapter 5: FIXING FLAKY TESTS IN MACHINE LEARNING LIBRARIES

5.1 INTRODUCTION

Many emerging applications in computer vision, natural language processing, and medical diagnosis are implemented using Machine Learning (ML) algorithms such as Deep Learning [1], Reinforcement Learning [2], or Probabilistic Programming [3, 4]. The recent pervasiveness of ML algorithms has led to the emergence of general-purpose libraries and specialized tools that build on top of these libraries. Many ML algorithms are *inherently random* – each execution of the algorithm may produce a slightly different result. Such randomness has an impact on how to carefully check the implementations of these algorithms, because the tests have to account for the variability of computed results from the code under test.

A common class of tests in existing ML projects are integration tests that check for end-to-end quality of the implementation of an ML algorithm. Such tests typically 1) create a small fixed or randomly generated dataset, 2) train the model on the dataset, 3) perform inference on the trained model, and 4) compute quality metrics and check if they are acceptable. Some common quality metrics include inference accuracy, recall, and error rate. When developers write their tests, they implement property checks using *approximate assertions* [53, 229, 256] that compare the metric to an acceptability bound, e.g.,

$$\texttt{assert}(\texttt{accuracy} > \alpha) \tag{5.1}$$

Developers typically choose the bounds based on intuition and experience with the code under test. These choices are often ad-hoc and not well-understood, especially when the developers are testing implementations of ML algorithms that inherently rely on some degree of randomness. While randomness in implementations of ML algorithms can be controlled through setting seeds in the underlying pseudo-random number generator(s), doing so can make the test less effective as it limits possible executions that can potentially help expose real bugs in the implementation [53]. However, by keeping randomness throughout, the tests may become *flaky* [50] – test executions can fail non-deterministically even when there is no bug in the implementation. The chance of flaky test failures depends on how tight the developer-selected bound α is. *An important question then becomes how to systematically select such bounds so that test flakiness can be minimized to a desirable level.*

Our Work. We present FLEX, the first tool for automatically fixing flaky tests due to algorithmic randomness. FLEX focuses on tests that use approximate assertions to compare the actual and expected quality of ML algorithm results. FLEX transforms the test and

systematically selects appropriate assertion bounds that reduce the chance of flaky failures.

The key challenge is to determine how to estimate appropriate assertion bounds with high statistical confidence. FLEX's solution is based on Extreme Value Theory (EVT). EVT [62, 63, 64] is a branch of statistics, often used in finance and hydrology, that can model extreme events, such as market risks (finance) or occurrence of extreme floods (hydrology). Given an input sample of measurements of some observed variable, EVT models the tail of the distribution, which can then be used to compute the likelihood of extreme values. The advantage of using EVT is that, in the limit, the tail distribution will converge to a specific group of probability distributions.

We use the Peak Over Threshold (POT) [62] method from EVT to estimate the tail distribution of a ML algorithm's result quality. With this method, the tail distribution converges in the limit to an instance of the Generalized Pareto Distribution (GPD) [62]. GPD is parameterized by a *shape* parameter, which determines if the measured quantity has a tail (left or right) that is *exponentially bounded*. An exponentially bounded tail converges quickly to GPD and can be used to estimate an appropriate bound for the variable in the failing assertion. On the other hand, a *heavy tailed* distribution cannot provide a reasonable estimate. In such a case, we either collect more samples (to get a better estimate) or resort to alternative test fixing strategies.

FLEX records the actual values in the assertion (e.g., the variable `accuracy` in the example assertion earlier) from multiple executions. It then uses the recorded values to estimate the GPD as representative of the tail distribution. Since the tail distribution converges to GPD only in the limit, FLEX uses statistical methods to find the sufficient number of samples of the output value that leads to convergence. FLEX then uses the inferred GPD to determine the likelihood of the extreme values and choose an assertion bound α that keeps the chance of the test failure below a pre-specified probability C.

FLEX implements several test fix strategies to reduce flakiness:

- **Update the assertion using a statistical tail bound:** FLEX handles two kinds of assertions. First, for assertions that compare the absolute values (e.g., the variables `accuracy` and α from our earlier example assertion), FLEX collects the samples of the actual value `accuracy`, computes the bound satisfying the confidence level using POT, and updates the constant α with the new bound. Second, for assertions that use bounds for differences between two values, FLEX estimates the tail distribution of the differences and updates the bound based on the tail estimate.
- **Update the assertion using an empirical bound:** FLEX updates the assertion as in previous strategy, but instead of computing GPD, it uses an empirical bound computed

using bootstrap sampling [257]. It is used when the POT method fails to compute the tail distribution or produces a heavy-tailed distribution.

- **Rerun the test to improve confidence:** FLEX does not modify the test body, but marks it using the `@flaky` annotation [241] so that the test is re-run on failure, only declaring true failure if it fails for all re-runs; this annotation then reduces the chance of a flaky failure stopping a build. Currently, developers may use reruns and specify the number of repetitions based on some intuition. Instead, FLEX determines the number either from the estimated GPD (when available) or using the observed failure rate.

Updating the thresholds in the assertions does not change the execution time of the test. However, re-running the test can increase the overall execution time (as a function of the failure probability).

Results. We evaluate FLEX on a corpus of 35 existing flaky tests collected from the latest versions of 21 projects, which use one of six popular Machine Learning and Probabilistic Programming frameworks: PyTorch [231], TensorFlow [78], TensorFlow-Probability [233], Pyro [258], PyMC3 [259], and NumPyro [232]. The dependent projects provide domain specific functionalities and have a wide user base.

FLEX proposes a fix for 28 tests (Section 6.6). It selected the statistical tail bound strategy in 17 cases, empirical bound strategy in 2 cases, and re-run strategy in 9 cases. For the remaining 7 tests, FLEX determines that the current assertion bound is looser than what FLEX suggests. Hence, we do not propose fixes for those cases, as the flaky failures, if they occur, are statistically rare. We sent 19 pull requests, each fixing one test, to the developers. So far, 9 pull requests have been accepted by the developers, 4 are pending, and 6 have been rejected. Of the 6 rejected pull requests, the developers mostly acknowledged the flakiness and chose to fix the problem in their own way custom to the project. These results jointly demonstrate that our approach can reduce the flakiness of tests by proposing appropriate assertion bounds for pre-specified confidence levels.

Contributions. This chapter makes the following contributions:

- We present FLEX, the first technique for automatically fixing tests that are flaky due to algorithmic randomness in ML algorithms.
- We present a novel test fixing algorithm that leverages statistical techniques from Extreme Value Theory to guide several test modification strategies.
- We evaluate FLEX on a corpus of 35 flaky tests, fixing 28 tests while determining that the rest do not need fixes. FLEX is publicly available at `https://github.com/uiuc-arc/flex`.

5.2 EXAMPLE

We present an example flaky test whose assertion is not properly bounded, leading it to pass and fail non-deterministically when run multiple times on the same version of code. The flaky test is named `test_ground_truth_separated_modes`, from *ICB-DCM/pyPESTO*, a library for parameter estimation that provides state-of-art algorithms for optimization and uncertainty analysis of black-box objective functions [260].

Listing 5.1 presents the (simplified) test code. The test first initializes a sampler using the Adaptive Metropolis Sampling algorithm, which is a Markov Chain Monte Carlo (MCMC) method (Line 2). It initializes a dataset for the test, which is sampled from a mixture of two Gaussian distributions (Line 3). The test then defines the objective function that needs to be optimized. In this case, the objective function measures whether the generated MCMC samples resemble the target mixture distribution using a negative log likelihood metric (not shown here). Then, the test uses the MCMC sampler to find a solution to the problem that uses 1000 iterations for sampling (Line 4). The test compares the results of the sampler with the expected ground truth (Line 8) using the Kolmogorov-Smirnov (KS) test [261], a popular statistical procedure used to find the distance between two probability distributions (lower is better). The test checks whether the KS distance/statistic is below 0.1 (Line 9).

```
1 def test_ground_truth_separated_modes():
2     sampler = sample.AdaptiveParallelTemperingSampler(internal_sampler=sample.AdaptiveMetropolisSampler(),
          ↪ n_chains=3)
3     problem = gaussian_mixture_separated_modes_problem()
4     result = sample.sample(problem, n_samples=1000, sampler=sampler, x0=np.array([0.]))
5     samples = result.sample_result.trace_x[0, :, 0]
6     rvs1 = norm.rvs(size=5000, loc=-1., scale=np.sqrt(0.7))
7     rvs2 = norm.rvs(size=5001, loc=100., scale=np.sqrt(0.8))
8     statistic, pval = ks_2samp(np.concatenate([rvs1, rvs2]), samples)
9     -assert statistic < 0.1
9     +assert statistic < 0.2
```

Listing 5.1: Fix for test in *ICB-DCM/pyPESTO*

```
1 def _propose_parameter(self, x: np.ndarray):
2     x_new = np.random.multivariate_normal(x, self._cov)
3     return x_new
```

Listing 5.2: Source of randomness for example flaky test

We found that this flaky test fails 17 out of 500 times we run it on the same version of code. Our inspection found that the computed KS statistic varies due to inherent randomness of the code under test; such variance in computed values is common in machine learning (ML) projects [53]. The source of randomness in this test is in the Adaptive Metropolis Sampling

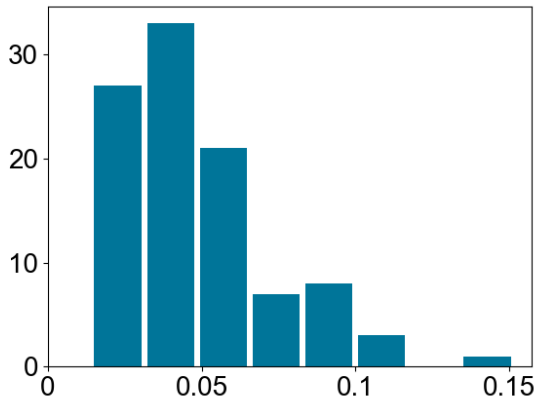


Figure 5.1: Distribution of values from example flaky test

algorithm. The sampling algorithm makes some random choices during its execution such as choosing the next sample (from a distribution) for a parameter that is being estimated. Listing 5.2 shows the corresponding code snippet. Since the sampler runs for a finite number of steps (1000 in this case), the solution may sometimes be further from the ground truth values than what is expected. As a result, the KS statistic can sometimes exceed 0.1, causing the test to fail.

We collected the actual computed values of the KS statistic at the failing assertion (Line 9) from several test executions. Figure 5.1 shows the distribution of the collected samples. Clearly, we see that some values exceed the expected bound (0.1) originally set by the developers. We assume the code under test is implemented correctly, so we would then need to repair the test code, providing a more reliable assertion bound to ensure it fails less often due to randomness.

To compute a better assertion bound, we need to examine the tail of this distribution and also provide statistical confidence in our estimation. A naive strategy here might be to use the observed extreme value as the new bound (0.15 here). However, this strategy does not give statistical confidence that the execution will never result in an even more extreme value. Another workaround might be to set the bound to a large value, say 1.0. However, doing so can lead to the test missing bugs which manifest as accuracy regressions. Ideally, we want to determine a value that is both *large enough* as to minimize the flakiness and *tight enough* as to not miss bugs.

We leverage methods from Extreme Value Theory (EVT) to compute a bound with high statistical confidence (Section 5.3). These methods take as input a set of samples of the observed variable (e.g., `statistic`) and return a curve representing the tail (left/right) of

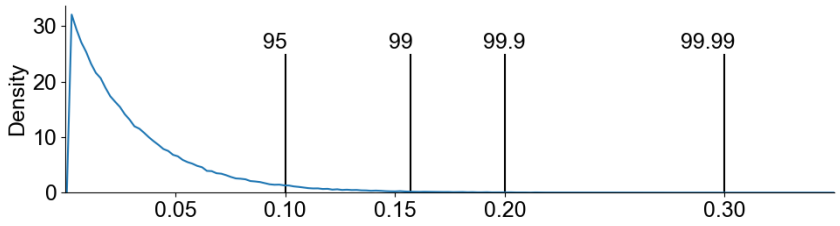


Figure 5.2: Estimated Tail Distribution (Exponential) and corresponding percentile estimates

the distribution. We can then use the tail distribution to estimate the most probable extreme value (max/min) for a pre-specified confidence level. In this example, since we want to find the maximum bound of `statistic`, we need to inspect its right tail. Using EVT method Peak Over Threshold, we are able to fit an exponential distribution to the tail samples (see Figure 5.2). We estimate this distribution using only 100 samples collected from executing this test. To check for goodness of fit and confirm that we do not need more samples, we use a sequence of statistical hypothesis tests (GPD test [262, 263]). Using this distribution, we can ultimately determine that the assertion bound should be 0.2, which ensures the computed values will lead to a passing assertion 99.9 percent of the time (the assertion bound is at the 99.9th percentile for the tail distribution). We do not choose the 99.99th percentile (0.3) in this case, since it seems to be too extreme. We sent a pull request that changes the assertion bound to this value to the developers of this project. The developers accepted and merged this pull request, leaving a message: *"Thanks for this contribution! I think checking the test percentiles is the way to go indeed"* [264]. Further, we also collected 1 million samples for this test and observed that our predicted bound indeed matches this empirical percentile.

An alternative strategy to fix such tests might be to fix the seed in the random number generator(s) (RNG) that are being used, which would make the test execution more deterministic. However, setting the seed can also make the test more brittle: future changes in code under test or the RNG can break the test. Also, it can hide potential bugs since the test will always observe the same set of values from the RNG. In this example, the developers also agreed on this point saying: *"I think checking the test percentiles is the way to go indeed (unless we set the RNG, which we however rather don't want to atm)"* [264].

5.3 BACKGROUND: EXTREME VALUE THEORY

Extreme Value Theory (EVT) encompasses statistical methods that model the probability of extreme events (e.g., those more extreme than any event observed so far). We will next describe EVT and related statistical methods that we use in our approach. We will use the

standard notation from the probability theory: X will denote a random variable, $X_1, \ldots, X_n$ will denote random variables, each representing observed samples of X, and $F(X \leq x)$ (or equivalently $F(x)$) will denote the cumulative distribution function (CDF) of the random variable X. It denotes the probability that the value of X is smaller than a constant x. To make distribution parameters $\boldsymbol{\theta}$ explicit, we will write $F(x; \boldsymbol{\theta})$.

To characterize the probability of extreme events, EVT studies values which are relatively smaller/larger (i.e. belong to the tail region) than the rest of the observations in the sample, and uses them to model the tail (right/left) of the distribution.

Peak Over Threshold (POT). For a random variable X, the POT method [62] takes as input a set of independent and identically distributed (i.i.d.) samples: $X_1, \ldots, X_n$, and outputs a distribution representing the tail of the distribution of X. The POT method uses a user-specified threshold T to select a subset of samples that exceed the threshold. This threshold helps select values from the tail of the distribution. POT represents the tail of arbitrary *continuous* distributions using *exceedance probability*. Given a random variable X, with CDF F_X, we define exceedance probability F_T as the CDF of X above threshold T:

$$F_T(y) = \mathbb{P}(X - T \leq y \mid X > T) = \frac{F(T+y) - F(T)}{1 - F(T)} \tag{5.2}$$

where $0 \leq y \leq x_F - T$, where x_F is the rightmost endpoint of F or infinity. Prior work [62, 265] showed that for a large class of continuous distributions F and large T, F_T can be approximated by a Generalized Pareto Distribution (GPD), i.e., $F_T(y)$ converges in distribution to $G(y)$ as $T \to \infty$, where

$$G(y; T, \sigma, \xi) = \begin{cases} 1 - \left[1 + \xi \frac{y-T}{\sigma}\right]^{-1/\xi} & \text{if } \xi \neq 0 \\ 1 - \exp^{-(y-T)/\sigma} & \text{if } \xi = 0 \end{cases} \tag{5.3}$$

Here, T, σ, and ξ correspond to *location*, *scale*, and *shape*, respectively. These parameters can be estimated using Maximum Likelihood Estimation (MLE) methods [266]. The *shape* parameter, ξ, determines the nature of the tail: light, exponential, or heavy.

Figure 5.3 presents an example of how different kinds of tail distributions behave. The *exponential-tailed distributions* and *light-tailed distributions* (defined as having less probability mass in the tail than exponential) converge very fast and can provide reasonable estimates of the extremes. However, the *heavy-tailed distribution* (defined as having more probability mass in the tail than exponential) converges very slowly. Computing an assertion bound in a high percentile for such a distribution would result in a very extreme value that may be an impractical assertion bound for a test.

Estimating Parameters of GPD. Given a set of observations $\boldsymbol{S} = x_1, \ldots, x_n$, the *location*,

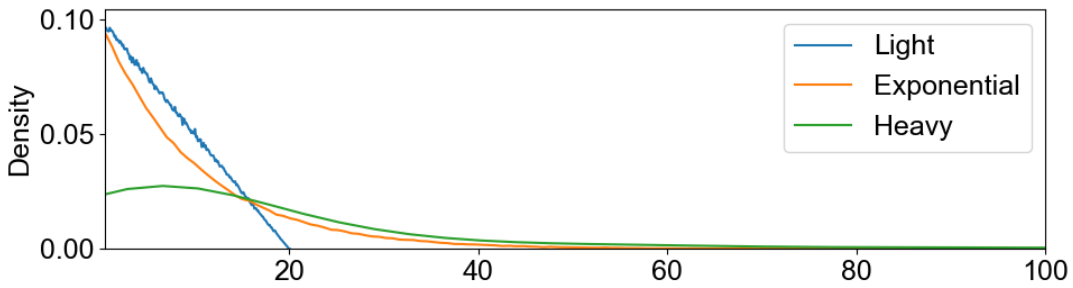


Figure 5.3: Example CDF plots for light, exponential, and heavy tailed GPD distributions with $\xi = -0.5$, $\xi = 0$, and $\xi = 0.5$ respectively ($\mu = 0$ and $\sigma = 10$)

scale, and *shape* parameters of GPD can be estimated using Maximum Likelihood Estimation (MLE) methods. MLE methods compute the point estimate of distribution parameters that maximize the likelihood that distribution produces the observed data. Formally, the likelihood function can be defined as $\mathbb{P}(\boldsymbol{\theta}|\boldsymbol{S}) = \mathbb{P}(\boldsymbol{\theta}|x_1) \cdot \mathbb{P}(\boldsymbol{\theta}|x_2) \cdot \ldots \cdot \mathbb{P}(\boldsymbol{\theta}|x_n) = \Pi_{i=1}^{n} \mathbb{P}(\boldsymbol{\theta}|x_i)$, where $\boldsymbol{\theta}$ is the set of parameters of GPD distribution. MLE then obtains the parameter estimates that maximize this likelihood: $\text{argmax}_{\boldsymbol{\theta}} \prod_{i=1}^{n} \mathbb{P}(\boldsymbol{\theta}|x_i)$. Intuitively, it selects parameter values such that observed data is most probable. As the number of observations grows, the MLE estimates converge in probability to the true values.

Goodness-of-fit vs. Samples Count. According to POT, the tail distribution is guaranteed to converge to the GPD distribution in the limit [62, 64]. However, it is unknown how many samples may be needed for convergence in practice, especially if the distribution has a heavy tail. The choice of threshold T determines a trade-off between goodness of fit and minimum samples required for convergence. Researchers have proposed several heuristics for choosing appropriate thresholds.

In this work, we adopt the methodology proposed by Bader et al. [263] for automated threshold selection using goodness of fit tests. The precise problem can be stated as follows: Given a sequence of samples $X_1, \ldots, X_n$ of size n, we want to determine the lowest threshold T such that the GPD fits the exceedances $Y_i = X_i - T$ adequately. Bader et al. propose using a sequence of goodness of fit tests for the GPD over each candidate threshold in an increasing/decreasing order until the stopping criteria is reached.

For an ordered set of thresholds: $T_1 < ... < T_l$, let there be z_i exceedances, $i \in \{1, \ldots, l\}$, for each threshold. The sequence of null hypotheses can be stated as "H_0^i : *The distribution of z_i exceedances above T_i follows the GPD.*" The alternative hypotheses are "H_1^i : *The distribution of z_i exceedances above T_i does not follow the GPD.*"

We use the non-parametric Anderson-Darling test [262] (as recommended by Bader et al.)

for this hypothesis test. To reduce the chances of choosing the wrong threshold by mistake (also known as False Discovery Rate or FDR), the authors introduce special stopping criteria when evaluating these hypotheses. In particular, we test each threshold, starting from the highest, and stop if the following criteria is satisfied: $\exp\left(\sum_{j=k}^{l} \frac{\log p_j}{j}\right) \leq \frac{\gamma \cdot k}{l}$ where γ is the False Discovery Rate (probability of choosing a wrong threshold), $k \in \{1, \ldots, l\}$ is the index of the current threshold, and p_j is the p-value returned by the jth hypothesis test H_0^j. This technique allows for a principled way to select a reliable threshold and check whether a GPD can be fit. When one or more of the hypothesis tests pass based on the stopping criteria, we say that the samples converged to a GPD and choose the lowest threshold for further analysis. If all the hypothesis tests fail, this means that we may need more samples. We abstract this check using *StopTest* function in our algorithm (Section 5.4.2).

Box-Cox Transformation. Box-Cox transformation [267] is a power transform that can create a monotonic transformation of data (i.e. preserves the original order of values). This transformation is useful in making the data closer to a normal distribution and stabilizing its variance. Normality is a key assumption in many statistical analyses. Hence, applying the Box-Cox transformation can enable a broader range of analyses on the data. The Box-Cox transformation can be described as follows:

$$y_i^{(\lambda)} = \begin{cases} \frac{y_i^{\lambda}-1}{\lambda}, & \lambda \neq 0; \\ \log y_i, & \lambda = 0; \end{cases} \tag{5.4}$$

where λ is a parameter that can be estimated from the samples using MLE methods. It can only be applied when $y_i > 0$, $i \in \{1, \ldots, n\}$.

Teugels and Vanroelen [268] showed that applying Box-Cox transformation can be useful in presence of heavy tails and can lead to faster convergence. Further, Helsel and Hirsch [269] showed that quantiles (or percentiles) are invariant to monotonic transformations. Hence, $Q_\tau(f(Y)) = f(Q_\tau(Y))$, where Q is the quantile function, $\tau \in (0, 1)$ is any given quantile, f is the monotonic transformation, and Y is the set of samples. There is no known guarantee that applying the Box-Cox transformation on data will prove to be always useful for any given statistical analysis [270]. However, it is a useful heuristic that can help speed up or even enable finding convergence for a distribution.

5.4 FLEX

We propose FLEX, a technique for fixing flaky tests caused by inherent algorithmic randomness in ML projects. FLEX assumes that the code under test is implemented correctly

and thus considers tests that fail some of the time to be flaky and in need of repair. Given an assertion A in a test $\mathbb{T}$ of the form `assert` $X < \alpha$, FLEX performs the following steps: 1) Collect and pre-process the samples $X_1, \ldots, X_n$ of actual value X from several test executions, 2) Determine the lowest possible threshold T such that a GPD, G_X, can be fit to $Y_i = X_i - t, i \in \{1, \ldots, n\}$, with a confidence of at least 95% using the Goodness-of-fit approach described in Section 5.3, 3) Estimate the most probable bound B from G_X for X, based on the desired confidence level $C \in (0, 1)$, as provided by the developer, and update the assertion bound to B. For instance, if $C = 0.99$ then we determine B such that $\mathbb{P}(X \leq B) \geq 0.99$.

5.4.1 FLEX Algorithm

Algorithm 6.1 describes the main FLEX algorithm. It takes a test $\mathbb{T}$, an assertion A in the test, and a confidence threshold C as input and returns the fixed version(s) of the test $\mathbb{T}^*$ as output. Intuitively, the algorithm executes $\mathbb{T}$ several times and collects the samples from the values being compared in the assertion until either the tail distribution converges to a light or exponential tail or the number of samples collected exceeds the maximum sampling limit (`MAX_SAMPLES`) and therefore we consider to not converge.

In each iteration of the loop (Lines 7-18), we execute the test N times and collect samples from the assertion (Line 8). We add the new samples to the existing set, *Samples*, and check if the tail distribution converges to a light or exponential tail (Lines 9-10). The estimation algorithm *TailBoundEstimator* (Section 5.4.2) takes the samples *Samples*, assertion A, a flag F to enable/disable the Box-Cox transformation (Section 5.3), and confidence level C as inputs. When a distribution has a light or exponential tail, the distribution has a finite bound and hence can be used to fix the test assertion. On the other hand, if the distribution does not converge or has a heavy tail, we might need more samples to get a better estimate. If we fail to get a bound, then we try to get an estimate by enabling the Box-Cox transformation (Line 12). We choose to check convergence first without transforming because the transformation adds extra overhead. Note that Box-Cox can be applied only on positive data. If all the samples are negative, then we change the sign of the values before the analysis and revert the sign of the results if the analysis succeeds. However, if we have a mix of positive and negative values, we do not apply this transformation.

We continue the loop until the sample size exceeds a user-set limit `MAX_SAMPLES` or if the tail converges to light or exponential distribution (Lines 14-16). Finally, FLEX patches the test using different available fix strategies depending on whether it finds a finite bound or not (Section 5.5) and returns the patched test (Line 19).

Algorithm 5.1: FLEX Algorithm

Input: Test $\mathbb{T}$, Assertion A, Confidence level C
Output: Fixed test $\mathbb{T}^*$

```
1: procedure FLEX(𝕋, A, C)
2:     Conv ← False
3:     D ←⊥
4:     Samples ← ∅
5:     N ← INITIAL_SAMPLE_SIZE
6:     Bound ← ∞
7:     while |Samples| < MAX_SAMPLES do
8:         S ← TestRunner(𝕋, A, N)
9:         Samples ← Samples ∪ S
10:        Conv, D, Bound ← TailBoundEstimator(Samples, A, False, C)
11:        if not Conv or not isLightOrExp(D) then                ▷ Enable transform
12:            Conv, D, Bound ← TailBoundEstimator(Samples, A, True, C)
13:        end if
14:        if Conv and isLightOrExp(D) then
15:            break
16:        end if
17:        N ← NEXT_SAMPLE_SIZE
18:    end while
19:    return Patcher(𝕋, A, Samples, D, Bound)
20: end procedure
```

5.4.2 Estimating the Statistical Tail Bound

Given a set of samples collected from test executions, the tail estimation algorithm applies the Peak Over Thresholds (POT) method to select values from the tail of the distribution (based on a threshold) and check if they converge to a tail distribution (which belongs to the Generalized Pareto Distribution (GPD)). However, selecting an appropriate threshold is non-trivial and can affect convergence. In this work, we use an automatic threshold selection technique [263] to compare different threshold choices (discussed in Section 5.3, *Goodness-of-fit*) and choose the lowest threshold that passes the GPD test [262], meaning it fits adequately to a GPD distribution.

Algorithm 5.2 shows the tail bound estimation algorithm, *TailBoundEstimator*. The algorithm takes as input a set of samples S, an assertion A, a flag F on whether or not to enable the Box-Cox transformation, and a confidence level C for choosing the bound. For the threshold that the POT method needs, we iterate over a set of possible, user-defined thresholds M (Line 5). Any value exceeding a threshold is considered to be part of the tail of the distribution and is used to fit to a distribution that helps compute the bound. For each threshold t, we compute the exceedances (Line 11). We apply the GPD test for convergence and compute the p-value p. We also obtain the shape (Light/Exp/Heavy) and

specification of the distribution D if it converges (Line 15). If the GPD test succeeds (i.e., $p >$ `SIGNIFICANCE_LEVEL`), and we obtain a light or exponential distribution (Line 20), then we estimate the bound B by computing the extreme percentile (Q_C) for the distribution such as 99th or 99.99th (Line 21). If the Box-Cox transformation is enabled, the *ComputePerc* method also transforms the bound back to the original scale of the samples. If we obtain a mildly heavy tail (e.g., $0 < \xi < \varepsilon$, for some small ε), we can still approximate it using an exponential distribution in some cases. We use the Likelihood Ratio Test [271] as a hypothesis test to check if original distribution and the exponential distribution obtained by fitting to the samples are not significantly different. We use the estimate if the hypothesis test passes (Lines 23-27). The *FitWithLRT* function (Line 23) abstracts this test and fitting to exponential distribution. If the stopping criteria (*StopTest*, described in Section 5.3) for the hypothesis tests is satisfied, we break out from the loop (Line 30).

When considering possible thresholds, we iterate through them in descending order, because we would like to select the lowest threshold (which in turn selects more samples from the tail region) to obtain a reliable estimate of the bounds of the distribution. Finally, the algorithm returns the status of convergence *Conv*, the GPD distribution D, and the estimated bound B.

5.4.3 Implementation of FLEX Components

We describe details on how we implement the main components for FLEX. We implement FLEX in Python.

Test Runner. It takes as input a test $\mathbb{T}$, an assertion A within $\mathbb{T}$, and the number of times to run N. First, Test Runner instruments test $\mathbb{T}$ to log the actual and expected values used in the assertion A. For instance, for an assertion of the form $assert\_allclose(a, b)$, it will instrument the assertion to log values a and b before the assertion. Second, it will execute the test N times, parse the values of a and b from the execution logs and return it to the caller. *Test Runner* uses *pytest* [272], a popular library for executing tests in Python projects.

Tail Bound Estimator. It implements the algorithm described in Section 5.4.2 to 1) check whether the tail distribution has converged and 2) to estimate an appropriate bound for the assertion A if the distribution converged and has a light or exponential tail. We use the "Eva" package in R [273] for the GPD test. We use the Box-Cox implementation in scipy to transform (or inverse transform) the samples. We choose the common significance level of 0.05 for the GPD test. For *StopTest* check, we use the false discovery rate (γ) of 0.05.

Patcher. The *Patcher* module takes a test $\mathbb{T}$, assertion A in the test, all collected samples *Samples*, fitted GPD D, and the proposed bound B as input and provides one or more fixed

Algorithm 5.2: Tail Bound Estimation Algorithm

Input: Samples S, Assertion A, Enable Transformation F, Confidence level C
Output: Convergence result *Conv*, GPD distribution D, Bound B

```
1: procedure TailBoundEstimator(S, A, F, C)
2:     if F then
3:         S ← Transform(S)
4:     end if
5:     M ← GetThresholds(S)
6:     D ←⊥
7:     B ← ∞
8:     Conv ← False
9:     P ← ∅
10:    for t ∈ SortedDescending(M) do
11:        exc ← {x − t|x > t, x ∈ S}                          ▷ POT method
12:        if |exc| < MIN_TAIL_SAMPLES then
13:            continue
14:        end if
15:        p ← GPDTest(exc)                                    ▷ Convergence Test
16:        P ← P ∪ p
17:        if p > SIGNIFICANCE_LEVEL then                      ▷ Check if converged
18:            D ← FitGPD(S)
19:            Conv ← True
20:            if isLightOrExp(D) then
21:                B ← ComputePerc(D, C, t, F)                 ▷ Find new bound
22:            else
23:                D′ ← FitWithLRT(D)                          ▷ Approximate to exponential
24:                if D′ ≠⊥ then
25:                    B ← ComputePerc(D′, C, t, F)
26:                    D ← D′
27:                end if
28:            end if
29:        end if
30:        if StopTest(P) then                                 ▷ Stopping criteria for hypothesis test
31:            break
32:        end if
33:    end for
34:    return Conv, D, B
35: end procedure
```

version(s) of the test as output. If B is not ∞, it updates the assertion in the test accordingly (Section 5.5.5) and returns the patched test to the caller. Otherwise, it may also propose other fixes for the test. We discuss each fix strategy in Section 5.5.

5.5 TEST FIXING STRATEGIES

FLEX provides three different strategies for automatically fixing and updating a flaky test depending on whether a finite tail bound can be computed using EVT and the nature of the

assertion (Sections 5.5.1-5.5.3). FLEX may also choose not to fix a test (Section 5.5.4) when it deems that the original bound is already looser than our proposed bound (indicating that failures are statistically rare). When multiple fixes are proposed by FLEX, we first we fix a test using the statistical bound when available. Otherwise, we use the empirical bound for the fix. If the estimated confidence interval for the empirical bound is too high, we choose to re-run the test instead. We may also need to adapt our strategy based on the context, (see Section 5.6.3).

5.5.1 Using the Statistical Tail Bound (**SB**)

If we obtain a light or exponential tailed distribution using Algorithm 5.2, then the distribution has a finite bound. We then simply compute the extreme percentiles (e.g., $Q_{0.99}$ or $Q_{0.9999}$), based on developer specified threshold C, to find a value that is higher (or lower) than the original bound used in the assertion and update the assertion with the new bound. The fixed assertion then has a failure probability of approximately $1 - C$.

5.5.2 Using the Empirical Bound (**EB**)

If the tail bound estimation algorithm (Algorithm 5.2) fails to converge or provide a finite bound (a heavy tail distribution), FLEX estimates an empirical bound from the observed executions. FLEX uses bootstrap sampling [257] to re-sample (with replacement) several times from the available samples and compute the extreme (max/min) from each instance of re-sampled data. As a result, FLEX obtains the set of sample extremes, E, and returns user-specified statistic of this set (e.g., $Q_\tau(E)$, mean, or median) as the new empirical bound. FLEX also computes the 95% confidence interval ($|Q_{0.975}(E) - Q_{0.025}(E)|$) which denotes the variability in the empirical bound – a smaller confidence interval indicates the empirical bound is close to the true bound.

5.5.3 Re-Running the Test (**RR**)

The Flaky [241] plugin for pytest allows the developers to automatically re-run the test on failure. To use this plugin, a developer needs to annotate the test using `@flaky`. This plugin also allows additional parameters: `max_runs` (default 2) and `min_passes` (default 1). The plugin runs the test up to `max_runs` times until it passes `max_passes` times. FLEX can annotate the test based on its observed failure rate during its analysis, i.e. re-using the observed executions at the end of Algorithm 6.1. FLEX computes the number of re-runs

in the following two ways: 1) FLEX computes the empirical failure probability of the test: $p = \frac{\#failures}{\#runs}$. Then it computes the number of re-runs using: $n = \lceil \log(1-C)/\log\ p \rceil$, where C is the developer provided confidence level (as in Algorithm 6.1) for minimum passing probability. 2) If the distribution converges to a heavy tail, we can also compute the probability that a sample exceeds the current bound set in the assertion. For instance, let D be the tail distribution (GPD) returned by Algorithm 5.2 and α be the current bound used in the test. Then, we can compute $P(x \geq \alpha) = 1 - D(\alpha)$, which is the failure probability of the assertion. We can then compute the re-runs similar to the previous case using this probability.

Unlike other approaches, re-running may increase the average running time of the test. Specifically, if the run time of the test is W, the expected run time of the test will be $\sum_{k=1}^{n} p^{k-1} \cdot (1-p) \cdot k \cdot W$.

5.5.4 Not Fixing a Test (**NF**)

In some cases, FLEX may propose a bound that is very close to, or tighter than the original bound, indicating that the assertion bounds are already conservative. This case indicates that test failures, if they occur, are extremely rare events. As such, we report, but do not propose the fix to the developers.

5.5.5 Updating Assertions

We describe how FLEX updates an assertion when a statistical or empirical bound for an assertion can be computed.

Assertions comparing absolute values. This category includes assertions that either compare with a computed value or with a constant. Some examples include the Python `assert` statement: `assert` $[x\ >\ |\ <\ |\ >=\ |\ <=]\ \alpha$, and some other APIs in unittest (e.g., `assertGreater`(x, α) , `assertLess`(x, α)) and numpy (e.g., `assert_array_less`(x, α)). To fix an assertion, FLEX simply replaces α with the bound it computes. Listing 5.3 shows an example of such a fix from the *ICB-DCM/pyPESTO* project.

```
-assert statistic < 0.1
+assert statistic < 0.2
```

Listing 5.3: Fix for test in *ICB-DCM/pyPESTO*

Assertions using tolerance thresholds. Some assertions check whether the relative or absolute difference between two floating-point values is less than a threshold. Some

examples include numpy APIs such as: `assert_almost_equal`$(a, b,$ `decimal` $= C)$, and also `assert_allclose`$(a, b,$ `rtol` $= C_1,$ `atol` $= C_2)$, where C, C_1, and C_2 are the relative and absolute thresholds respectively. In these cases, FLEX collects the values of both a and b from test executions and computes the absolute or relative difference from each execution. FLEX estimates the tail distribution using these differences as samples. It updates the assertion to either use a lower tolerance threshold or reduce the decimal places being compared depending on the kind of assertion. Listing 5.4 shows an example from the *microsoft/hummingbird* project for absolute tolerance fix.

```
-assert_allclose(model.predict(X),torch_model.predict(X),rtol=1e-4,atol=1e-5)
+assert_allclose(model.predict(X),torch_model.predict(X),rtol=1e-4,atol=1e-4)
```

Listing 5.4: Fix for test in **microsoft/hummingbird**

5.6 METHODOLOGY

5.6.1 Projects and Flaky Tests

We follow a similar methodology as Dutta et al. [53] to select machine learning projects for our evaluation. We start with two popular machine learning libraries (PyTorch [231] and TensorFlow [78]) and four probabilistic programming systems (Pyro [258], NumPyro [232], TensorFlow-Probability [233], and PyMC3 [259]) on GitHub. We use GitHub's feature to track the projects dependent[1] on these libraries and also have more than 10 stars, as an indication of popularity.

Some of these core libraries can have hundreds of dependents, so we only select the top 100 dependent projects per library for our study. Table 5.1 shows all the project details. Overall, we select 305 unique projects. We develop a general installation script to install these libraries, which creates a virtual python environment using Anaconda [274], and then it installs the library and all its dependencies in the environment along with some libraries for testing, such as pytest. In Python libraries, developers typically specify all dependencies in the `setup.py` file, which is the main installation module. They can also specify additional dependencies (e.g., for building documentation and testing) in a `requirements.txt` file. However, in some cases, the installation process may not work due to incomplete dependency specifications, missing system dependencies (such as SQL server client or open-mpi library),

[1]We use only dependent "packages" as reported by the GitHub API, which are projects that can be installed as a library to be used by others. We use packages because they are more likely to be actively maintained by developers and have reasonable test suites.

Table 5.1: Details of projects used

Project	Dependent	Filtered
TensorFlow	836	100
PyTorch	906	100
TensorFlow-Prob	283	100
NumPyro	3	3
Pyro	13	13
PyMC3	31	31
Total	2072	347
Unique	1836	305
Successful at Testing	-	144
Projects with Flaky Tests	-	21

or required specialized build/testing systems (such as Bazel [275]). Our installation script installs a general set of system dependencies but relies on pip and pytest to build and test the libraries. Overall, we are able to successfully install and test 144 projects.

Of the resulting 144 projects, we ran their tests using FLEX's Test Runner module, running only the tests with approximate assertions that we support. Initially, we run each test up to 30 times while recording the actual computed values in each assertion using Test Runner's instrumentation. If any assertion's actual values remain exactly the same for all those initial runs, we discard those tests from consideration. For the remaining tests with assertions whose actual values vary, we run those tests 500 times while recording test results (success/failure) from each run. If we detect any failures (and at least some passing runs as well), we mark the test as flaky and use it for our evaluation. Ultimately, we are left with 21 projects with 35 flaky tests as part of our evaluation. Recall, FLEX assumes that the underlying distribution is continuous. We also included 7 tests with discrete distributions, mainly resembling binomial distribution (that can be often approximated well with a continuous distribution).

5.6.2 FLEX Configuration

For evaluation, we configure FLEX to initially collect 100 samples (`INITIAL_SAMPLE_SIZE` in Algorithm 6.1). If more samples are needed, we configure FLEX to collect more samples in batches of 50 (`NEXT_SAMPLE_SIZE` in Algorithm 6.1). We specify FLEX to collect at most 3000 samples before stopping (`MAX_SAMPLES` in Algorithm 6.1).

We set the minimum number of tail samples when testing for convergence to be 50 (`MIN_TAIL_SAMPLES` in Algorithm 5.2). We use `SIGNIFICANCE_LEVEL` of 0.05 for the GPD tests. For the confidence level (C in Algorithm 6.1), we configure FLEX to use 90th, 95th, 99th, 99.9th and 99.99th percentiles.

We run all experiments on Azure VMs (`Standard_F32s_v2` configuration) with 3.4GHz Xeon processor with 32 cores and 64GB memory. While executing the tests, we run 20 threads in parallel as to speed up experiments.

5.6.3 Reporting to Developers

For each fix we obtain from running FLEX on a flaky test, we prepare a pull request to send to the developers. In the process of preparing the pull request, we manually inspect the proposed fix(es) and the surrounding context in the test as to determine if the fix seems reasonable. For example, if the assertion initially checks if some count of values is greater than zero, and the fix is to change that assertion bound to instead be a negative number, then the fix does not make sense in the context of this test. We select one of the other available fixes in such a case (Section 5.5) based on the context.

For each project in our evaluation, we first send a pull request for fixing one test. We initially send just one pull request as to not bother developers immediately with many pull requests if they are not willing to consider such changes. If the developers accept the initial pull request, we send pull requests for fixing the remaining flaky tests. We ensure every pull request we send only addresses one flaky test at a time. As part of a pull request, we provide both the proposed fix and the statistical evidence we gathered by running FLEX on the test. We present to developers information on the number of times the assertion failed out of how many reruns, and we explain how the tail distribution was computed using the actual values from test executions. We suggest the bounds at either 99.9th or 99.99th percentile (depending on the test), but for completeness we also provide the values for the other percentiles (including also 90th, 95th, and 99th percentiles). If the developer chooses one of these bounds, we adjust the pull request accordingly.

5.7 EVALUATION

In this section, we address the following research questions:

RQ1: How many flaky tests can FLEX fix? Which fix strategy does it apply in each case? How many test runs does it need in each case?

RQ2: How do the different fix strategies compare and in what scenarios can each be applied?

RQ3: How do developers respond to the fixes?

5.7.1 Flaky Tests Fixed by FLEX

We run FLEX on the 35 flaky tests found in the latest versions of 21 projects from Section 5.6.1. Table 5.2 presents the results. Each row represents one flaky test. Column **ID** is a shorthand identifier we give to each test for later reference, **Project** presents the name of the project as a GitHub SLUG, **Test** presents the name of the test, **SHA** presents the commit SHA of the project that we ran FLEX on, **#Samples** presents the number of samples FLEX collected for its analysis, **Conv.** presents whether the tail distribution converged (Algorithm 6.1), (✓ means yes, ✗ means no), and **L/E** presents whether the distribution had a light or exponential tail, when it converges (✓ means yes, ✗ means no, - means not applicable).

For the final four columns under **Fixed**, we mark with ✓ the type of fix that FLEX proposed for the test. The column **SB** means the test was fixed using a statistical bound estimated using the light or exponential tail distribution computed using POT. **EB** means the empirical bound strategy is used. **RR** means re-run strategy is used. By default, FLEX prioritizes the fixes SB>EB>RR (Section 5.5), but adjusts the recommendations based on the context of the test (Section 5.6.3). **NF** means that flaky test was not fixed, because FLEX's proposed new assertion bound is tighter than the original (Section 5.5.4). As such, these tests would be considered tolerant enough already, so FLEX's proposed assertion bound fix would not make sense. In sum, FLEX proposes a fix for 28 flaky tests (**SB**, **EB**, **RR**), while 7 remain not fixed (**NF**). We compare the fix strategies in Section 6.6.

Overall, for 17 tests, FLEX requires only 100 samples (the minimum that we collect) for convergence, showing that our analysis is efficient in most cases. Only for 2 tests does FLEX require more than 1000 samples for convergence. We apply the GPD test to check if we have enough samples to reliably estimate the tail distribution. This gives us statistical confidence in our results. Further, by considering different thresholds for selecting the tail values, we ensure that we can select as many samples from the tail of the distribution for the best possible result. For the remaining 7 tests, which FLEX chooses not to fix, the proposed bound was tighter than original bound. The Box-Cox transformation helped in early convergence and bound estimation for 8 cases: T1, T5, T14, T17, T21, T22, T32, and T34.

5.7.2 Comparison of Fix Strategies

Out of 28 fixed tests, FLEX proposes the statistical bound for 17 tests, empirical bound for 2 tests, and the re-running strategy for 9 tests. In cases where FLEX suggests multiple fixes, we manually inspect and select the most appropriate fix based on the context. We next

Table 5.2: Results of running FLEX on 35 flaky tests

ID	GitHub Project	SHA	#Samples	Conv.	L/E	Fix Type			
						SB	EB	RR	NF
T1	microsoft/coax	37c3e6	100	✓	✓	✓	-	-	-
T2	deepchem/deepchem	6a535b	3000	✗	–	-	-	✓	-
T3	deepchem/deepchem	6a535b	100	✓	✓	✓	-	-	-
T4	deepchem/deepchem	6a535b	3000	✗	–	-	-	✓	-
T5	deepchem/deepchem	6a535b	450	✓	✓	✓	-	-	-
T6	fastnlp/fastNLP	22c6e6	150	✓	✓	✓	-	-	-
T7	rlworkgroup/garage	1f1742	150	✓	✓	-	-	✓	-
T8	RaRe-Technologies/gensim	cfc9e9	250	✓	✓	-	-	✓	-
T9	RaRe-Technologies/gensim	cfc9e9	650	✓	✓	-	-	✓	-
T10	RaRe-Technologies/gensim	cfc9e9	400	✓	✓	-	-	✓	-
T11	RaRe-Technologies/gensim	cfc9e9	650	✓	✓	-	-	✓	-
T12	microsoft/hummingbird	9f71c2	3000	✗	–	-	✓	-	-
T13	microsoft/hummingbird	9f71c2	200	✓	✓	✓	-	-	-
T14	microsoft/hummingbird	9f71c2	1050	✓	✓	✓	-	-	-
T15	kornia/kornia	cf8e85	100	✓	✓	-	-	-	✓
T16	magenta/magenta	b4b9af	100	✓	✓	-	-	-	✓
T17	magenta/magenta	b4b9af	400	✓	✓	-	-	-	✓
T18	plasticityai/magnitude	7ac0ba	3000	✓	✗	-	✓	-	-
T19	plasticityai/magnitude	7ac0ba	100	✓	✓	✓	-	-	-
T20	plasticityai/magnitude	7ac0ba	100	✓	✓	✓	-	-	-
T21	IntelLabs/nlp-architect	728e21	100	✓	✓	✓	-	-	-
T22	facebookresearch/parlai	fb5c92	100	✓	✓	✓	-	-	-
T23	pgmpy/pgmpy	413c61	100	✓	✓	✓	-	-	-
T24	pymc-learn/pymc-learn	4f1ee6	100	✓	✓	-	-	-	✓
T25	pymc-learn/pymc-learn	4f1ee6	3000	✗	–	-	-	✓	-
T26	ICB-DCM/pyPESTO	a34608	100	✓	✓	✓	-	-	-
T27	tristandeleu/pytorch-meta	389e35	200	✓	✓	-	-	✓	-
T28	refnx/refnx	34e369	100	✓	✓	✓	-	-	-
T29	stellargraph/stellargraph	1e6120	150	✓	✓	✓	-	-	-
T30	WillianFuks/tfcausalimpact	9fc9e8	100	✓	✓	-	-	-	✓
T31	google/trax	beaca3	100	✓	✓	-	-	-	✓
T32	lmcinnes/umap	05840e	100	✓	✓	✓	-	-	-
T33	lmcinnes/umap	05840e	100	✓	✓	-	-	-	✓
T34	zfit/zfit	a798f9	100	✓	✓	✓	-	-	-
T35	zfit/zfit	a798f9	100	✓	✓	✓	-	-	-
Σ	21		614.29	31	30	17	2	9	7

discuss in which scenarios each fix might work.

We observe that FLEX's statistical tail analysis converges for 31 tests, out of which we obtain a light or exponential tail for 30 tests and a heavy tail for one test. For 4 tests where the analysis does not converge, even applying the Box-Cox transformation does not aid the analysis. These scenarios occur either because either there are very few samples in the tail region or the samples only consist of very few discrete values.

In some cases, the variable in the assertion of a test might have a known hard bound such as *count* or *length* that are lower bounded by zero (e.g., `assert (np.count_nonzero(scores) > 0)` from *deepchem/deepchem*). This assertion sometimes fails when the count is zero. Hence, this case also does not satisfy FLEX's requirement of the samples belonging to a continuous distribution. However, this information is not easily interpretable just from the samples that FLEX's tail analysis collects. In such cases, FLEX may sometimes propose a negative assertion bound (using the inferred tail distribution), which is an impractical fix. Further, updating the assertion to check for ≥ 0 also does not make sense. In these cases, re-running the test is the only reasonable fix that FLEX can propose.

We propose the empirical bound fix strategy when we have a large set of samples and can estimate a bound with high confidence (i.e., small confidence interval). This strategy is useful in scenarios where the tail analysis fails to converge, and the quantity of interest does not have a known hard bound (like the previous example). For instance, in the *microsoft/hummingbird* project, the test `test_tree_regressors_multioutput_regression` contains a flaky assertion: `assert_allclose(model.predict(X), torch_model.predict(X), rtol=1e-05, atol=1e-05)`.
FLEX tracks the maximum absolute difference between the values being compared and obtains a empirical bound of 3.27 ± 0.96. This bound is evidently much higher than the absolute tolerance specified in the test (10^{-5}). FLEX suggested a fix using this bound to the developers. In this case, however, developers found an actual bug in their code which was causing such erroneous executions.

5.7.3 Developer Response to FLEX's Fixes

Using our methodology for sending pull requests to developers (Section 5.6.3), we ultimately sent 19 pull requests for tests for which FLEX proposes a fix. Table 5.3 presents the status of our pull requests per project, representing the 28 tests that FLEX can fix. Column **A** means number of pull requests accepted, **P** means number pending, R means number rejected, and **U** means number unsubmitted (we are waiting initial response from the developer on our first sent pull request). For *pymc-learn/pymc-learn*, we do not send a pull request since the project has been inactive for the last two years. The total number of pull requests (under column **PRs**) matches the number of tests for which we sent fixes.

So far, developers accepted 9 pull requests. 4 pull requests are still pending developer response, and 6 pull requests are rejected. For most of our pull requests, we selected the estimate based on the 99.99th percentile as the new bound of the test. In some cases we use a different percentile after discussion with developers, and we provide the estimates for the

Table 5.3: Pull Requests

Project	Tests	PRs	A	P	R	U
coax-dev/coax[276]	1	1	1	0	0	0
deepchem/deepchem[277]	4	1	0	1	0	3
fastnlp/fastNLP[278]	1	1	1	0	0	0
rlworkgroup/garage[279]	1	1	1	0	0	0
RaRe-Technologies/gensim[280]	4	1	0	0	1	3
microsoft/hummingbird[281, 282, 283]	3	3	1	0	2	0
plasticityai/magnitude[284]	3	1	0	1	0	2
IntelLabs/nlp-architect[285]	1	1	0	0	1	0
facebookresearch/parlai[286]	1	1	1	0	0	0
pgmpy/pgmpy[287]	1	1	1	0	0	0
ICB-DCM/pyPESTO[264]	1	1	1	0	0	0
pymc-learn/pymc-learn	1	0	0	0	0	1
tristandeleu/pytorch-meta[288]	1	1	0	1	0	0
refnx/refnx[289]	1	1	0	0	1	0
stellargraph/stellargraph[290]	1	1	0	1	0	0
lmcinnes/umap[291]	1	1	1	0	0	0
zfit/zfit[292, 293]	2	2	1	0	1	0
Σ21	28	19	9	4	6	9

other percentiles (Section 5.6.3). Listing 5.5 shows an example of a fix for a test in *zfit/zfit*. For this test, FLEX estimates the extreme percentiles as follows: 90th: 10^{-5}, 95th: 10^{-6}, 99th: 10^{-7}, and 99.99th: 10^{-8}. The original bound is 10^{-6} (shown in red). Initially, we submitted the pull request with the 99.99th percentile as the fix (shown in blue). However, the developers suggested they would prefer the 99th percentile (shown in green) to reduce the flakiness to some extent (compared to the current rate) for now and would later like to investigate into why the computed values are so low.

```
- assert scipy.stats.ks_2samp(x, xns).pvalue > 1e-6
+ assert scipy.stats.ks_2samp(x, xns).pvalue > 1e-8
+ assert scipy.stats.ks_2samp(x, xns).pvalue > 1e-7
```

Listing 5.5: Fix for test in *zfit/zfit*

Of the 6 rejected pull requests, the developers accepted different fixes for the tests. For two of our pull requests to *microsoft/hummingbird* [281, 282], the developers reasoned that our proposed bounds were too large and hence indicative of a real bug in their library. Later on, they proposed a global change for fixing several numerical precision issues in their code, which impacted such tests. For *refnx/refnx* [289], the developer preferred setting the seed instead of changing bounds. For *RaRe-Technologies/gensim* [280], after discussing with the developers, we found that the failures were due to a race condition in the code, and we

proposed a different fix that they accepted [294]. Out of remaining two cases, in one case, for *IntelLabs/nlp-architect* [285], the developers rejected our pull request without providing any reason. For *zfit/zfit* [293], the test was already marked *flaky* and the developers chose not to make any changes.

The positive responses from developers confirm that tuning assertion bounds is a reasonable way to fix flaky tests in these ML projects that deal with randomness (e.g., consider the comments mentioned in Section 6.2). The developers from *microsoft/hummingbird*, while accepting one of our initial pull requests, also confirmed that they rely on their intuition to manually set such bounds: *"...For the moment we manually set a 'reasonable' value for the differences, but having a more 'scientific' way of finding them will be great!"*. The developers of *lmcinnes/umap* accepted our pull request and commented *"Thanks – the non-deterministic tests are a little annoying at times. I appreciate the effort you went to to ensure this won't trip accidentally"*. These positive responses show a practical value of FLEX's systematic approach for determining assertion bounds.

5.8 DISCUSSION: COMPARISON WITH CONCENTRATION INEQUALITIES

In probability theory, concentration inequalities provide conservative probabilistic bounds on how much a random variable deviates from a given value (e.g., its mean). Since the goal of FLEX is to estimate the probability that the random variable used in the assertion is within or exceeds a given value, it seems natural to ask the question: *Can we use concentration inequalities instead of extreme value theory in FLEX?*

The key advantages of concentration inequalities are that they are non-parametric, hold under very mild assumptions, and can be applied to a wide class of distributions. Hence, they may be useful in overcoming some limitations of extreme value theory methods such as non-convergence, heavy tails, and slow convergence (requiring large number of samples). However, the conservative nature of concentration inequalities (in theory) may produce bounds that are more extreme than those obtained by extreme value theory. We first introduce the background for concentration inequalities.

5.8.1 Background: Concentration Inequalities

Chebyshev's inequality [295] is a well-known result in probability theory that guarantees that, for a broad class of distributions, no more than a certain proportion of its values will exceed a certain distance from the mean. More formally, let X be a scalar random variable with a finite mean μ and variance σ^2. Then for any real number $k > 0$: $\Pr(|X - \mu| \geq k\sigma) \leq \frac{1}{k^2}$.

Chebyshev's Inequality can be applied to any arbitrary distribution, assuming known mean and variance. However, it often provides a conservative estimate.

Dasgupta's Inequality. Dasgupta [296] proposed a tighter bound if the distribution is known to be Gaussian:

$$\Pr(|X - \mu| \geq k\sigma) \leq \frac{1}{3k^2} \tag{5.5}$$

Dasgupta's Inequality can also be used with estimated mean ($\overline{X}$) and variance (S^2) [296], i.e., computed from available samples. This provides a conservative estimate but is typically tighter than Chebyshev. Since the distribution is Gaussian, it is symmetric about mean. Hence, we can also use the one-sided variant of this inequality (using estimated mean and variance):

$$\Pr(X - \overline{X} \geq kS) \leq \frac{1}{2} \cdot \frac{1}{3k^2} \tag{5.6}$$

Cantelli's inequality with estimated mean and variance. Often the mean and variance of a distribution are unknown. Tolhurst [297] proposed a variant of Cantelli's equation [298], which is the one-sided version of Chebyshev's Inequality. It uses mean and variance estimated from available samples:

$$\Pr(X - \overline{X} \geq kQ_N) \leq \frac{1}{N+1} \left\lfloor \frac{N+1}{g^2+1} \right\rfloor \tag{5.7}$$

where N is number of samples, $g^2 = \frac{Nk^2}{N-1+k^2}$, $Q_N^2 = \left[\frac{N+1}{N}\right] S^2$, and $\lfloor . \rfloor$ is the floor function. Here, $\overline{X}$ and S^2 are the sample mean and variance respectively. This inequality holds for $k > 1$ and $N \geq 2$. This bound converges to Cantelli's inequality as $N \to \infty$.

Sub-Gaussian Distributions. A Sub-Gaussian distribution [299] has a tail that decays (or converges) at least as fast as a Gaussian distribution. Intuitively, the tail of a Sub-Gaussian distribution is dominated by a Gaussian distribution. This is a useful property since the tail properties of Gaussians can be extended to such distributions directly. For instance, if we know that the underlying distribution of a random variable X is sub-gaussian, then we can directly use the Dasgupta inequality (Eq. 5.6) to estimate the tail probabilities.

Kurtosis Test [300] is a statistical test that is used to check whether the distribution is heavy-tailed or light-tailed (sub-gaussian) relative to a Gaussian distribution. We use the Kurtosis test to verify if the samples are drawn from a Sub-Gaussian distribution and apply the Dasgupta Inequality (Eq. 5.6) to obtain the one-sided tail probability. This provides a tighter tail bound than the more general estimate (Eq. 5.7).

5.8.2 Algorithm: Computing Passing Probability of Test

In our recent work, FASER [65], we leveraged concentration inequalities to design an algorithm for estimating the passing probability of test for a given assertion bound. We introduce it next.

To compute the passing probability of test T, we first execute the test several times and obtain the samples of the variable, say X, in the assertion. Let N be the number of samples. Now, we want to compute the probability: $\Pr(X < \theta)$, where θ is a given *assertion bound*. One potential solution is to compute the empirical probability, i.e., the proportion of samples that fall below the assertion bound. However, this may not be a reliable estimate when N is small and when the nature of distribution is not known. Further, collecting a large number of samples is expensive and hence may not always be feasible.

To overcome these challenges, we apply the inequalities described in Section 5.8.1. Algorithm 5.3 describes the steps for computing passing probability. First, we check if the underlying distribution of X is Gaussian (Line 5) using Shapiro-Wilk Test [301]. If the distribution is Gaussian, we use the one-sided Dasgupta inequality (Eq. 5.6) to estimate the passing probability of the test for the given bound (Line 6). Given a bound θ, we compute $\Pr(X < \theta) = 1 - \Pr(X - \overline{X} \geq kS)$ as the passing probability of the test, where $k = (\theta - \overline{X})/S$ (Line 4). Here, $\overline{X}$ is the estimated mean and S is the estimated variance.

Second, if the distribution is not Gaussian, we check if it is Sub-Gaussian (Line 5) using the Kurtosis Test (Section 5.8.1). If the test passes, we apply the one-sided Dasgupta inequality (Eq. 5.6) in the same manner as the previous case (Line 6). This provides a more conservative bound than the actual underlying distribution but it is tighter than the general estimate.

Third, if the distribution is neither Gaussian nor Sub-Gaussian, we apply Cantelli's inequality using estimated mean and variance (Eq. 5.7) to compute the passing probability of the test (Line 8). This inequality gives us a conservative estimate of the actual passing probability – i.e., in the limit, the actual passing probability is guaranteed to be equal or greater than the estimated passing probability.

5.8.3 Estimating Fault-Detection Ability of Test

We use mutation testing to determine the fault-detection effectiveness of the test. We generate mutants and select the subset of mutants that produce distributions of the assertion variable that are *sufficiently different* from the original distribution. These mutants represent *accuracy bugs* in code, i.e., bugs that lead to wrong results (e.g., lower model accuracy). We define such mutants as *effective mutants*. We use the two-sample Kolmogorov-Smirnov

Algorithm 5.3: Passing Probability Algorithm

Require: Samples $\mathcal{D}$, Bound θ
Ensure: Passing probability P_θ
procedure PassingProb($\mathcal{D}$, θ)
  $\overline{X} = mean(\mathcal{D})$
  $S = std(\mathcal{D})$
  $k = \frac{\theta - \overline{X}}{S}$
  if IsGaussian($\mathcal{D}$) or IsSubGaussian($\mathcal{D}$) **then**
    $P_\theta = 1 - \text{DasGuptaIneq}(\mathcal{D}, k)$ ▷ Using Eq. 5.6
  else
    $P_\theta = 1 - \text{CantelliIneqEst}(\mathcal{D}, \overline{X}, S, k)$ ▷ Using Eq. 5.7
  end if
  return P_θ
end procedure

(KS) Test [302, 303] to determine if the original and the mutant distributions are sufficiently different. We use KS test because it is non-parametric and applies to a broad class of distributions.

We define several general (such as operator mutation) and *domain-specific* mutation operators (such as `torch.zeros` and `torch.ones`) to generate mutants of the code under test. For a given test T, we only apply mutations to covered source lines of code. More details about the mutation operators can be found in our work [65].

Let $m_1, \ldots, m_K$ be the generated set of effective mutants. For a given bound, θ, we define the probability that a mutant, m_i is killed as:

$$\Pr_\theta(m_i \text{ is killed}) = \Pr(X_{m_i} > \theta) = \frac{1}{|D_{m_i}|} \sum\nolimits_{x \in D_{m_i}} \mathbb{1}_{[x>\theta]} \tag{5.8}$$

where D_{m_i} is the set of samples obtained and X_{m_i} is the assertion variable when T is executed with mutation m_i.

We define the *mutation score* (MS) of the test, T, for a given θ, as the average mutant kill rate:

$$\text{MS}(T, \theta) = \frac{1}{K} \sum\nolimits_{i=1\ldots K} \Pr_\theta(m_i \text{ is killed}) \tag{5.9}$$

Our definition of mutation score is different than that used in traditional mutation testing where each mutant is *deterministically* either killed (1) or the mutant survives (0). In contrast, for non-deterministic tests, we define whether a mutant is killed as a *probability*, $\Pr_\theta \in [0, 1]$, and the mutation score as the *average probability of killing a mutant*. For deterministic tests, it reduces to standard mutation score metric, i.e., $\Pr_\theta \in \{0, 1\}$.

5.8.4 Results

We apply FASER's algorithm (Algorithm 5.3) to estimate the bounds for the test fixed by FLEX. For this experiment, we select all tests excluding the cases when the samples contained discrete values or when there was a hard lower bound (e.g., count of elements in a list) from Table 5.2. We use FASER to estimate the bounds for the same percentile (e.g., 99 or 99.99) as used by FLEX. For each test, we compare how conservative the bounds obtained by FASER are compared to FLEX by computing *FASER Bound/FLEX Bound*. We also compute the mutation scores for both FASER and FLEX's bound to estimate the effectiveness of the bounds. Table 5.4 presents the results.

In Table 5.4, the column *Comparison* shows the ratio of FASER's bound (column *FASER*) compared to FLEX's bound (column *FLEX*). Rows marked **DNC** indicate tests for which FLEX did not converge or converged to a heavy tail. Rows marked NA indicates tests for which the distribution contained discrete values or when there was a hard lower bound (e.g., count of elements in a list). Column *MS Diff.* shows

Overall, we observe that FASER's bounds are more conservative in 21 tests. In 6 tests, FASER's bounds are an order of magnitude more conservative than FLEX, whereas in 2 tests, FASER's bounds are two orders of magnitude more conservative than FLEX. In all cases, we used only 100 samples to obtain FASER's estimates. Interestingly, for 5 tests where FLEX does not converge or converges to heavy tail, FASER can still be applied to obtain a conservative bound. These results suggest that while concentration inequalities might be less expensive to compute and are also applicable when extreme value theory does not work, they often produce very conservative bounds. Out of 7 tests where FLEX suggests no fix, FASER's analysis would suggest a fix in 6 tests.

Overall, in 7 tests, FASER's bounds regress the mutation scores by more than 5% over FLEX, whereas in 5 tests it exceeds 10%. In two cases (T26, T34), the mutation score difference exceeds 50%!

Interestingly, the magnitude of the bound comparison does not always correlate with the difference in mutation scores. We reason that this is because in addition to the difference in bounds, the mutation scores also depend on the scale of the bounds. For instance, for T19 and T20, FASER's bound is 10 times than FLEX. However, because the values are so small (10e-6 vs 10e-5), the absolute difference between them is also quite small. As a result, the mutant generator is not able to generate many mutants that produce values in this range.

On another hand, for T26, the mutation scores differ by more than 50% when FASER's bound is only 4 times that of FLEX. In this case, the absolute difference is large: 0.75, which provides the mutant generator a higher chance to produce mutants that generate samples in

Table 5.4: Comparison with Concentration Inequalities

ID	GitHub Project	FLEX	FASER	Bound Comparison	ΔMut. Score
T1	coax-dev/coax	8.00E-4	1.00E-5	80	0
T2	deepchem/deepchem	DNC	–	–	–
T3	sleepy-owl/deepchem	1.58	7	4.43	8.39
T4	deepchem/deepchem	DNC	–	–	–
T5	deepchem/deepchem	8.83	0.12	73.58	4.12
T6	fastnlp/fastNLP	120	210	1.75	0
T7	rlworkgroup/garage	NA	–	–	–
T8	RaRe-Technologies/gensim	NA	–	–	–
T9	RaRe-Technologies/gensim	NA	–	–	–
T10	RaRe-Technologies/gensim	NA		–	–
T11	RaRe-Technologies/gensim	NA	–	–	–
T12	microsoft/hummingbird	DNC	–	–	–
T13	microsoft/hummingbird	1.00E-4	3.00E-4	3	0.04
T14	microsoft/hummingbird	5.32	7.1	1.33	0.17
T15	kornia/kornia	1.00E-4	2.00E-4	2	0.05
T16	magenta/magenta	0.016	0.08	5	0.26
T17	magenta/magenta	1.00E-3	4.00E-3	4	0.44
T18	plasticityai/magnitude	DNC	–	–	–
T19	plasticityai/magnitude	1.00E-6	1.00E-5	10	0.01
T20	plasticityai/magnitude	1.00E-6	1.00E-5	10	0.01
T21	IntelLabs/nlp-architect	4.00E-3	2.00E-2	5	13
T22	facebookresearch/parlai	44	182	4.14	0
T23	pgmpy/pgmpy	0.047	0.2	4.26	0.09
T24	pymc-learn/pymc-learn	1	1	1	0
T25	pymc-learn/pymc-learn	DNC	–	–	–
T26	ICB-DCM/pyPESTO	0.236	0.99	4.19	50.68
T27	tristandeleu/pytorch-meta	NA	–	–	–
T28	refnx/refnx	1	1	1	0
T29	stellargraph/stellargraph	1.00E-5	1.00E-5	1	0
T30	WillianFuks/tfcausalimpact	0.05	0.4	8	8.63
T31	google/trax	0.796	-1.40	1.75	28.78
T32	lmcinnes/umap	0.45	0.05	9	0
T33	lmcinnes/umap	0.67	0.15	4.47	44.5
T34	zfit/zfit	1.60E-8	-5	3.12E8	58.59
T35	zfit/zfit	5.21E-4	-10	1.91e4	0

that range.

Additionally, because the mutation algorithm is limited by the mutation operators that we define, it cannot always generate effective mutants – mutants that alter the output distribution significantly. For example, for T22, we observe that FASER's bound is 4 times than FLEX's bound and is greater by over 140 in absolute terms. However, the mutation score difference is 0, even though we generate 344 mutants! Also, the original bound (22) has a mutation score of only 0.8%. This reflects that the mutation operators that we use are not very effective in this scenario and we need to design better operators.

Because the mutation scores are often similar, one could use concentration instead of extreme value theory without a significant loss in fault-detecting effectiveness. However, the impact may be hard to predict apriori. Hence, developers must exercise due diligence if choosing the more conservative bounds. To obtain the best of both worlds, we recommend that future work may compute bounds using both concentration inequalities and extreme value theory in parallel and choose the more precise bound. Since the cost of sampling dominates the overall cost of both approaches, this hybrid approach would help in avoiding the shortcomings of both approaches.

5.9 THREATS TO VALIDITY

The projects we use in our evaluation are only a subset of all machine learning applications. We selected these projects by starting with the most popular machine learning libraries and finding their dependent projects. We believe these projects are representative. We also focus on flaky tests that use approximate assertions, found to be a common type of flaky test from prior work [53]. We detect the flaky tests in these projects through repeated reruns. We use a similar rerun strategy to detect these flaky tests as prior work [53]. The flaky tests we use are then a lower-bound on the total number of flaky tests, as other flaky tests may require even more reruns to observe some failures. Such tests have a higher chance of flakiness and hence are likely the ones that developers would want to focus on.

Since FLEX builds on several statistical methods and heuristics, there is a possibility of estimating incorrect bounds. As a result we may sometimes over-estimate the bound which may cause the tests to miss some bugs. We minimize this risk by using high significance levels both for individual hypothesis tests and for the algorithm for threshold selection. To increase confidence in the the bug finding ability of the fixed test one can use strategies from the literature, e.g., [229]. Like other prior work on repairing tests [304, 305, 306, 307, 308], we assume code under test to be correct, with the implementation matching the intended logic. Ultimately, we send the proposed fixes as pull requests to developers, providing them the statistical evidence of the fixes. We allow the developers, who are more knowledgeable about the code than us, to use the provided evidence to make the final judgment call on how good the proposed fix is.

5.10 RELATED WORK

Flaky Tests. Luo et al. [50] performed the first empirical study on flaky tests, studying open-source projects and determining common root causes for flaky tests. Later work would

build upon Luo et al.'s findings, developing techniques to detect specific flaky tests with root causes found from their study, such as due to test-order dependencies [244, 245], asynchronous waits [248], or unordered collections [247]. However, these prior works focused on flaky tests in traditional software.

Dutta et al. [196] performed an empirical study to find common root causes for flaky tests in ML applications. They found that a common cause for flakiness in this domain is algorithmic randomness (e.g., calls to random number generators), both in the application code and the tests. Leveraging these insights, they developed FLASH [196] to detect such flaky tests using convergence testing. Our work shows how to fix such flaky tests using EVT and statistical hypothesis tests to update approximate assertion bounds.

Flaky Test Repair. Prior work on test repair generally involves updating assertions after code under test has evolved [304, 305, 306, 307, 308]. The assumption is that the code under test is correct and so test assertions need to match the current implementation. We also make this assumption in our work and propose a technique for adjusting assertions that better match the underlying implementation while reducing flakiness. Recently, there has been work on repairing specific types of flaky tests, such as flaky tests due to test-order dependencies [55] or due to unordered collections [56]. The goal of these techniques is to make flaky tests no longer fail due to their flakiness root cause. Lam et al. [57] proposed mitigating flakiness due to asynchronous waits by automatically adjusting wait times as to reduce the chance of tests failing due to waits. We also focus on fixing flaky tests by adjusting assertion bounds, reducing the chance of a flaky test (though not completely eliminating it). We focus on flaky tests with approximate assertions that can fail due to inherent randomness in executing code under test.

TERA [229] aims to reduce the time of testing ML projects by changing the algorithm hyper-parameters, which potentially *increases* the flakiness of tests. TERA is based on Bayesian optimization guided by convergence testing. FLEX instead changes the assertion bounds to *reduce* flakiness (while not impacting execution time) by leveraging distribution estimation from extreme value theory.

Extreme Value Theory (EVT). We rely on EVT [62, 63, 64] to determine tail distributions of the computed values in approximate assertions. While we rely on the Peak Over Threshold (POT) [62] method to apply EVT, there are other popular methods as well. Block Maxima Method (BMM) [63] uses a given block size B (selected by user) to split the given samples into equally sized blocks and then considers the maximum value in each block. According to the Fisher-Tippet theorem [64], this distribution is then guaranteed to converge to a Generalized Extreme Value distribution. The choice of block size B is often not intuitive and can affect the convergence of the distribution. This method is generally better suited for data

with some periodicity, e.g., daily/month weather data/finance data. In our case, the values in the assertions do not exhibit any such periodicity in general, which makes this method less effective. The POT method, on the other hand, considers exceedances over some threshold T (selected by the user). These exceedance values from the samples then converge to a Generalized Pareto Distribution (GPD) [62]. This method is better suited to our use case.

Testing of Programs in Presence of Randomness. Machine learning frameworks like TensorFlow [78] and PyTorch [231] have led to a surge in machine learning based applications. Probabilistic programming has also been gaining in popularity in recent years, leading to the development of numerous probabilistic programming languages [3, 76, 85]. Researchers proposed techniques for testing and debugging probabilistic systems [196, 254, 255], machine learning frameworks [12, 250, 251, 252, 309], and randomized algorithms [39] to complement manual test writing. Researchers have also explored techniques for testing randomized or adaptive software [310, 311, 312, 313] or analyzing robustness of programs [223, 314, 315, 316]. However, the advances in efficient automated test generation for these systems has yet to catch up with the speed of application development while capturing the inherent non-determinism and overcoming the lack of reliable oracles in this domain.

5.11 SUMMARY

We present FLEX, the first tool for automatically fixing tests from machine learning (ML) projects that are flaky due to algorithmic randomness. FLEX analyzes and transforms tests that use approximate assertions to compare actual and expected values that represent the quality of ML results. We leverage statistical methods from Extreme Value Theory to determine the appropriate assertion bounds as to reduce the chance of flaky test failures. We evaluate FLEX on a corpus of 35 tests collected from the latest versions of 21 ML projects. Overall, FLEX identifies and proposes a fix for 28 tests. We sent 19 pull requests, each fixing one test, to the developers. So far, 9 have been accepted by developers. We envision that many future applications will continue to incorporate a degree of randomness. Our goal is to help developers cope with randomness and overcome the lack of reliable testing oracles both in ML and other domains.

Chapter 6: OPTIMIZING EXECUTION TIME OF MACHINE LEARNING TESTS

6.1 INTRODUCTION

The growing popularity of Machine Learning (ML) has led to rapid development of general-purpose libraries and specialized tools that build on top of these libraries. These tools perform various tasks in applications like computer vision, natural language processing, and medical diagnosis by implementing algorithms such as Deep Learning [1], Reinforcement Learning [2], or Probabilistic Programming [3, 4]. However, bugs in the implementations of such tools can make the ML-based applications vulnerable to failures and lead to loss of lives and property [9, 10].

Testing of ML libraries and tools is currently not well-understood, which causes the developers to apply ad-hoc techniques when writing tests. An important trait of many ML algorithms – e.g., Reinforcement Learning [317], Bayesian modelling [318], Seq-to-seq learning [319] – is *inherent randomness*, meaning that each execution of the algorithm may produce a slightly different result. Hence, developers often opt to execute such algorithms for long cycles (more than actually necessary) to ensure their results are *highly likely to be close to expected values in tests*, thereby unnecessarily increasing the cost of testing.

An optimized testing procedure for ML algorithms needs to make careful choices. ML algorithms allow a developer to control their accuracy and run-time through a set of *hyper-parameters* – numerical values that guide model selection or define training strategies. Common examples of hyper-parameters for learning algorithms include the number of training iterations, learning rate, and the number of elements sampled from output distribution.

Listing 6.1 shows a common pattern of tests that check for correctness of an ML algorithm. Such a test typically involves: (1) setup code, e.g., for downloading sample data, initializing test environment on Line 2, (2) initializing a stochastic ML algorithm with one or more hyper-parameters $P_1, \ldots, P_k$ on Line 3, (3) executing the algorithm and computing accuracy metrics on Lines 4-5, and (4) assertions checking if computed metrics are near to or exceed

```
1 def testAlgo():
2     [[setup code]]
3     trainer = MLAlgo( P_1 = v_1 , P_2 = v_2 , ... , P_k = v_k )
4     trainer.train()
5     metrics = trainer.compute_metrics()
6     for i in range(len(metrics)):
7         assert metrics[i] >= expected[i]
```

Listing 6.1: Example test pattern

expected values on Lines 6-7. When the developers do not choose the hyper-parameters carefully, these tests can be slow to execute. We observe that such tests are typically more time-consuming than other tests in the test-suite and consume a significant portion of test time, sometimes even exceeding 80% (Section 6.5).

Choosing optimal hyper-parameters is often non-intuitive and difficult for a developer to get right. In this process, developers also need to ensure that their tests are not too *flaky* – pass and fail non-deterministically for the same version of code – due to randomness of the ML algorithm. For instance, if a developer chooses hyper-parameters too conservatively (e.g., selects a large number of training iterations), the test becomes *less flaky* but is too expensive to run. On the other hand, if the developer chooses hyper-parameters too liberally, the test runs faster but can become *more flaky*. Dutta et al. [53] showed that algorithmic randomness is the major cause of flakiness in the ML domain, further signifying the importance of accounting for randomness in tests.

At present, developers have to make ad-hoc decisions and manually select sub-optimal hyper-parameter values. Naturally, they are more inclined to be conservative since they are focused on eliminating flakiness. An important and intriguing challenge then is to find a way to significantly reduce the running time of such tests without making them more flaky.

Our Work. We propose TERA – the first automated technique for reducing the cost of regression testing in ML projects[2] without making the tests more flaky. TERA rests on a (seemingly counter-intuitive) insight that modestly relaxing the desired passing probability of some tests can result in both faster and highly reliable execution of the test suite. To find the optimized version of the tests, TERA systematically navigates the trade-off space between execution time of the test and its passing probability by tuning the algorithm hyper-parameters.

To determine the degree of flakiness of a test, we define a metric *Test Passing Probability* (**TPP**) – probability that a given test passes for the same code version. For its exploration, TERA exposes TPP to the developer as *a tunable knob* $\alpha \in [0, 1]$. For instance, if the developer specifies $\alpha = 0.99$, TERA will try to find a set of optimal hyper-parameter values which minimize the running time of the test without dropping the probability of passing below 99% (or such that $TPP \geq \alpha$). If successful, the optimization will reduce the regular testing time and the build time (which involves running tests across multiple environments) of the project in the continuous integration systems like Travis-CI [320] and CircleCI [321].

TERA formulates the problem of exploration of the trade-off space between the execution

[2]By "ML projects" we denote ML-related libraries or tools, in which the outcomes are affected by some stochastic component (e.g., in the algorithm or the data-set that the test generates). Hereon, we will use "projects" to refer to such libraries/tools.

time of the test and its passing probability as an instance of *Stochastic Optimization* over the space of algorithm hyper-parameters. Stochastic Optimization [322] encompasses a family of algorithms for optimizing objective functions when randomness is present. The main benefits of Stochastic Optimization are that (1) the optimization can reduce the running time of the test in a black-box fashion (i.e., it does not need to look into the test body); and (2) it automates the selection of algorithm hyper-parameters in a systematic manner. In this work, we use Bayesian Optimization, as an instance of Stochastic Optimization method, to solve the optimization problem.

For a given test, TERA constructs an objective function that executes the test (using a given set of hyper-parameter values) several times and returns the average execution time (the optimization objective) and the test passing probability. However, to construct this objective function, we must address two challenges: (1) How many times to execute the test? and (2) How to estimate the test passing probability? Existing literature on optimization algorithms does not provide mechanisms to automatically develop such stochastic objective functions.

We apply two key techniques to address the challenges above. First, we monitor the values in the assertions of the test while executing the test several times. We then use the samples of the actual values in the assertion (e.g., `metrics[i]` in Listing 6.1, Line 7) to check whether it converges to the target distribution. We use the convergence property to dynamically determine how many times to execute the test. Second, we approximate the passing probability of the test by computing the passing probability of its assertions. We compute this probability by first estimating the distribution using the samples of the actual values and then computing how likely it is to exceed the expected values (e.g., `expected[i]` in Listing 6.1, Line 7). We present more details in Section 6.4.6.

Results. We evaluate TERA on a corpus of 160 tests selected from 15 projects, chosen from four popular ML libraries – PyTorch, TensorFlow, Pyro, PyMC3 – and tools that build on top of them. These tools provide application specific functionalities and have a wide user base, making them an important part of the ML domain. TERA found the optimized configurations for 133 tests. TERA's optimized tests are 2.23x (geo-mean) faster than the original tests, for the passing probability threshold (α) of 0.99 (or 99%). Developers already accepted our optimizations for 24 tests at the time of writing this chapter. We performed two studies on the ability of optimized tests to detect faults:

- On a set of mutated programs, we observed that the mutation scores increase slightly on average. We further inspect some mutants and find two key trends, which we discuss in Section 6.7.1: 1) an optimized test catches a bug missed by original test when faults

introduce small variations in computations which are absorbed in longer cycles (original) but detected in tighter executions (optimized) – increasing true positive rate, and 2) optimized test misses a bug when the error accumulation exceeds a certain threshold (more likely in longer cycles) – increasing false positives. TERA's approach of changing hyper-parameters only affects executions which exhibit such small deviations - which are very rare as demonstrated by our mutation study. Hence, the optimized tests retain most of the fault detection ability of original tests. We also discuss a composite test-execution strategy to mitigate the false positives in Section 6.7.2.

- On a set of 12 historical failures in the project builds, we confirmed that our optimized tests were able to detect the faults in all cases.

These results jointly show that our approach can improve the performance of testing, while retaining the fault detection ability of the optimized tests. We anticipate developers can apply TERA, in addition to existing tests, when a new test of ML algorithm gets added and when such a test fails due to regression errors.

Contributions. This chapter makes the following contributions:

- We frame the problem of reducing the running time of tests in Machine Learning projects as an optimization problem over the space of hyper-parameters used in learning algorithms.
- We propose TERA, an automated approach for optimizing expensive tests by combining Bayesian optimization and statistical testing techniques.
- We evaluate TERA on 160 tests from 15 projects. We show that the optimized versions of the tests run 2.23x times faster while still retaining similar fault detection ability.

We provide a complete replication package containing the source code of TERA and the instructions for reproducing our results at `https://github.com/uiuc-arc/tera`.

6.2 EXAMPLE

Listing 6.2 shows an example test (simplified) for a reinforcement learning algorithm in **ML-Agents** project [46]. **ML-Agents** provides implementations of several training algorithms (like deep reinforcement learning) for training agents in games and simulation environments. We describe the test next.

Lines 2-12 initialize a simple simulation environment (*SimpleEnvironment*) and the training algorithm (**SAC**). Line 13 performs the training step. Lines 14-18 compute the score (rewards)

of the trained agent for the given environment and checks if the scores are above the expected value (0.8).

```
1  def test_2d_sac():
2      env = SimpleEnvironment(...)
3      config = TrainerSettings(
4          trainer_type=TrainerType.SAC,
5          hyperparameters=SACSettings(
6              learning_rate=5.0e-3,
7              batch_size=16,
8              ...
9          ),
10         max_steps=10000
11     )
12     trainer = create_trainer(env, config, ...)
13     trainer.start_learning()
14     processed_rewards = [
15         reward_processor(rewards) for rewards in env.final_rewards.values()
16     ]
17     for reward in processed_rewards:
18         assert reward > 0.8
```

Listing 6.2: Example test from **ml-agents**

```
1  def sample_mini_batch(batch_size, sequence_length):
2      num_seq_to_sample = batch_size // sequence_length
3      mini_batch = AgentBuffer()
4      ...
5      num_sequences_in_buffer = buff_len // sequence_length
6      start_idxes = (
7          np.random.randint(num_sequences_in_buffer, size=num_seq_to_sample)
8          * sequence_length
9      )# Sample random sequence starts
10     for key in self:
11         mb_list = [self[key][i : i + sequence_length] for i in start_idxes]
12         mini_batch[key].set(list(itertools.chain.from_iterable(mb_list)))
13     return mini_batch
```

Listing 6.3: Source of Randomness (Batching)

```
1  def sample_action(dist):
2      ...
3      continuous_action = dist.sample()
4      return AgentAction(continuous_action)
```

Listing 6.4: Source of Randomness (Sampling Action)

We select this test for optimization with TERA because it uses a machine learning algorithm (**SAC**), which is a reinforcement learning algorithm that makes random choices, requires selecting several hyper-parameters, and contains an approximate assertion (Line 18). These hyper-parameters (like `max_steps`) determine the running time of the test. Developers

typically choose these hyper-parameters in an ad-hoc manner, based on their intuition on what seems to be *good enough*. As a result, this test runs longer than what is required (as we show later) to achieve the desired reward (Line 18). The original test takes 90 seconds to run.

The Soft-Actor Critic (SAC) Algorithm [47] is a deep Reinforcement Learning algorithm. Reinforcement learning algorithms aim to maximize the expected reward of an agent solving a given task (such as playing a game).

Sources of Randomness. The SAC algorithm involves several sources of randomness. Listing 6.3, shows the simplified code snippet for a function `sample_mini_batch`, which is used by SAC algorithm [48]. In each iteration, the algorithm uses this function to compute gradients using batches randomly sub-sampled (Lines 6-12) from the agent buffer (which contains traces of the previous steps). Listing 6.4 shows another function `sample_action`, which is used by SAC [49] to sample the next action (which can either be a discrete choice or a continuous value) using a specified distribution (Line 3) for the agent. Due to these random choices of mini-batches and actions at each step, every execution of the test can yield slightly different results (rewards). Hence, if the developers do not choose optimal hyper-parameters[3], the test can sometimes fail and become unacceptably *flaky*.

Optimizing the test. A naive approach to reduce the running time of the test might be to do binary search on the `max_steps` parameter and choose a value for which the test passes. However, this approach is problematic. First, reducing `max_steps` alone, or any other hyper-parameter may not help us find the optimal run-time. We need to simultaneously adjust other hyper-parameters like `batch_size` and `learning_rate`. Second, running the test once is not enough. We need to ensure the test passes with high probability i.e., is not too *flaky*. These problems make manual test optimization hard for developers.

To optimize this test, we select three hyper-parameters: `learning_rate`, `batch_size`, and `max_steps`. To use TERA, we need to first define a valid range of values for each parameter. This, in turn defines the search space for optimization. For `learning_rate`, we select a continuous range $[1e^{-5}, 1.0]$, which is a few orders below and above the original value $(5e^{-3})$. For `batch_size`, we allow a set of discrete choices $\{2, 4, 6, 8, 16, 32, 64, 128, 256\}$, which are typical batch sizes used in machine learning. For `max_steps` we select a discrete interval: $[100, 10000]$ with increments of 100. We select the original value (10000) as the upper bound for this interval since we want to reduce the number of steps and consequently the run-time.

[3]For tests like this, one could argue that a simple way to deal with randomness during testing is to set the seed in the random number generators, which will make the execution more deterministic. The developers can then just execute the test for a much smaller number of steps and reduce the run-time. However, setting the seeds may not always be the right choice: they can be brittle in presence of program changes and can hide bugs [53].

Using the parameter specifications above, TERA automatically adds instrumentation blocks to the code, which perform two steps: (1) replace original parameter values with placeholders: e.g., `max_steps = 10000` => `max_steps = ''<max_steps>''`, and (2) add code to log the actual (`reward`) and expected values (`0.8`) in the assertion. The values in the assertion help TERA reason about how likely is the test to pass over multiple runs.

Next, TERA's Optimizer module constructs a scoring function, that encodes the optimization problem. The scoring function runs the instrumented version of the test with the given parameter values several times and monitors the execution of the test. Then, it inspects the execution trace (the logged assertion values) and determines the probability of passing of the test. This scoring function is called by the Bayesian optimization algorithm, as it systematically explores the search space. For this experiment, we allow the minimum passing probability threshold $\alpha = 0.99$.

TERA reduces the running time of this test to less than 15 seconds. This is over 6x faster than the original test. The optimal configuration TERA found has the following hyper-parameter values: `max_steps` : 2300, `batch_size` : 4, and `learning_rate` : 0.023. To estimate the impact of this optimization on the fault detection capability, we can compute the mutation score post-hoc. In the case of the tests in the **ML-Agents** project, the mutation score after the optimization is slightly above the one of the original tests (62.44% vs 61.16%), indicating that the fault detection capability is not reduced.

6.3 BACKGROUND

In this section, we introduce necessary background related to Bayesian optimization, which is a Stochastic Optimization method, and convergence testing.

6.3.1 Bayesian Optimization

Bayesian Optimization [323, 324] is a popular technique used for global optimization of black-box functions. Given a randomized objective function f, we want to find an input $x \in \mathbb{R}^d$ ($d \in \mathbb{N}$) which minimizes the output of f, subject to a set of constraints on the input space, encoded by the functions $g_1, \ldots, g_K$. Formally:

$$\min_{x \in \mathbb{R}^d} f(x), \text{ s.t. } g_i(x) \geq 0, \text{ for } i = 1, \ldots, K \tag{6.1}$$

Bayesian Optimization algorithms do not make any assumptions about the nature of the objective function and use a prior distribution to model the behavior of the objective function. The user only needs to define the input space of the objective function. The algorithm then

evaluates the function using different inputs and updates the prior to form the posterior distribution using the outputs obtained from function evaluations. The posterior distribution is then used to create an acquisition function. The acquisition function is used to select the inputs for the next round such that it maximizes the chances of finding the optimal parameters.Two common choices of prior/posterior distributions include Gaussian Processes, used in Gaussian Process Regression [325], and Kernel Density Estimators (Non-Parametric), used in Tree-Parzen Estimators [326] (which we use in this work). Examples of acquisition functions include the probability of improvement, expected improvement (used in this work), and knowledge gradient. Bayesian Optimization is advantageous over other methods like random/grid/genetic search [323] when the objective function is computationally expensive.

Researchers have previously employed Bayesian Optimization for problems such as compiler auto-tuning [327], compiler testing [328], finding optimal configurations for software systems [329], and program analysis [330, 331].

6.3.2 Automating Convergence Testing

Given a test which performs stochastic computations, we want to determine how flaky it is. A naive way would be to run the test a large number of times and then check how often it fails. However, this approach is expensive, especially when the test run-time is high or when the test fails rarely.

Convergence testing. Suppose we have an assertion Φ in a test function T. We can determine the probability of passing of the test by computing the probability of passing for this assertion. To compute this probability, we need to reason about the entire distribution of values that the expression in the assertion can evaluate to. Without loss of generality, let us assume we have an assertion of the form: **assert** $x < \gamma$, where x is a variable in the test and γ is a fixed threshold.

We want to estimate the distribution for x so that we can compute the probability of x exceeding γ. We frame this problem as estimating the distribution of an unknown function $\mathcal{F}$, where $\mathcal{F}$ evaluates T, capturing and returning the value of x. We use a sampling-based approach for this problem such that we execute $\mathcal{F}$ several times, obtain a number of samples of x, estimate the distribution from the samples, and compute the probability of passing: $\Pr(\mathcal{F} \leq \gamma)$. This approach involves two main challenges. We need to decide (1) *how many samples* to collect at minimum and (2) whether *we have seen enough samples*.

Several convergence metrics exist in literature [332, 333, 334]. We use the Geweke Diagnostic [333] (similar to [53]) as a heuristic to measure convergence of a set of samples, to solve the second challenge outlined above. Intuitively, the Geweke diagnostic checks whether

the mean of, say, the first 10% of samples is not significantly different from, say, the last 50%. If true, then we can say that the distribution has converged. To measure the difference between the two sub-sets of samples, the Geweke diagnostic computes the Z-score (can be essentially considered as standard deviation), which is computed as the difference between the two sample means divided by the standard errors. Equation 6.2 presents the formula for the Z-score computation for Geweke diagnostic, where a is the early set of samples, b is the later set of samples, $\hat{\lambda}$ is the mean of each set and Var is the variance of each set of samples.

$$z = \frac{\hat{\lambda}_a - \hat{\lambda}_b}{\sqrt{\text{Var}(\lambda_a) + \text{Var}(\lambda_b)}} \tag{6.2}$$

To use the Geweke Diagnostic as the convergence test, the user needs to specify the minimum desired threshold (which we call the *convergence threshold*). The convergence testing procedure keeps collecting samples (from test runs) until the Geweke Diagnostic drops below the user-specified threshold. Naturally, a lower threshold needs more samples for convergence. Dutta et al. [53] showed that using a threshold of 1.0 works well for detecting flakiness in ML projects. We use the same threshold 1.0 in our work.

Choosing the minimum number of samples is non-trivial. Too few samples can lead us to incorrect conclusions whereas too many samples can be too expensive to compute (especially when running the test is expensive). The user can choose appropriate number of samples. For our evaluation we use a minimum of 30 samples (guided by existing studies [335] that recommend this number of samples for statistical significance).

6.4 TERA

6.4.1 Problem Formulation

We formalize the optimization problem TERA aims to solve as follows. Given a test $T : \emptyset \mapsto \{0, 1\}$, TERA transforms the test to an equivalent variant $T' : \boldsymbol{\theta} \mapsto \{0, 1\}$, which is parameterized by an ordered tuple of hyper-parameters $\boldsymbol{\theta}$. Here, $\boldsymbol{\theta} = (P_1, \ldots, P_k)$, where each P_i ($i \in \{1, \ldots, k\}$) is a tunable hyper-parameter we can optimize and k is the number of hyper-parameters identified in T.

Each parameter is either a discrete integer (e.g., number of iterations) or a continuous value (e.g., learning rate). Therefore, $P_i \in U \ \forall i \in \{1, \ldots, k\}$, where $U = \mathbb{Z}$ or $\mathbb{R}$ and $\boldsymbol{\theta} \in U_1 \times U_2 \ldots \times U_k$.

We define a function $TPP : (T', \boldsymbol{\theta}) \mapsto [0, 1]$, which takes a transformed test T' and a tuple of hyper-parameters $\boldsymbol{\theta}$ and returns the probability of *passing* of T' when executed using the selected hyper-parameters $\boldsymbol{\theta}$. Additionally, we also define function $Time : (T', \boldsymbol{\theta}) \mapsto \mathbb{R}^+$,

Algorithm 6.1: TERA Algorithm

Input: Test T, Parameters $\boldsymbol{\theta}$, Min. Passing Probability α
Output: Optimized Test T^*, Parameters $\boldsymbol{\theta}^*$

1: **procedure** TERA($T, \boldsymbol{\theta}$)
2: $\quad$ *Search_Space* $\leftarrow$ *Initialize_Search_Space*(P)
3: $\quad$ $T' \leftarrow$ *TestInstrumentor*($T, \boldsymbol{\theta}$)
4: $\quad$ *Optimizer* $\leftarrow$ *BayesOpt*(*Search_Space*, *Scorer*, T', `MAX_EVALS`, `TIMEOUT`, α)
5: $\quad$ $T^*, \boldsymbol{\theta}^* \leftarrow$ *Optimizer.minimize()*
6: **return** T^*, $\boldsymbol{\theta}^*$
7: **end procedure**

which returns the execution time of the test using the selected parameters.

TERA searches for a hyper-parameter tuple $\boldsymbol{\theta}^* \in U_1 \times U_2 \ldots \times U_k$, which when provided as an input to T' minimizes the execution time of test: *Time* (our objective function), given the constraint that the test passes with at-least probability $TPP(T', \boldsymbol{\theta}^*) \geq \alpha$. Formally:

$$\boldsymbol{\theta}^* = \underset{\boldsymbol{\theta} \in U_1 \times \ldots \times U_k}{\operatorname{argmin}} \; Time(T', \boldsymbol{\theta}) \;\; \text{s.t.} \;\; TPP(T', \boldsymbol{\theta}) \geq \alpha \tag{6.3}$$

We must address several challenges to solve this optimization problem. First, the nature of the objective function (*Time*) is unknown since we may not have sufficient information about the exact functional form of the given test or the code under test. Hence, we need black-box optimization methods for this problem (Section 6.4.5). Second, the optimization space of hyper-parameters is typically large, which makes any analytical or enumerative approaches infeasible. Further, evaluating each configuration can be expensive since the test execution can take several minutes and even multiple executions. Hence, we resort to sampling based approaches to find optimal hyper-parameters. However, instead of randomly sampling from the search space, we use Bayesian techniques to sample more efficiently (Section 6.4.5). Third, since the test execution involves randomness (as shown in Section 6.2), we need to determine how likely the test is to pass with a given tuple of hyper-parameters $\boldsymbol{\theta}$. A naive strategy is to run the test N times and report how often it passes. However, this can lead to imprecise results. We show how we can apply statistical techniques to both determine *how many times to run the test* and precisely *compute the probability of passing* using the assertions in the test (Listing 6.1, Lines 6-7). We filter out hyper-parameters which drop the probability of passing below user-specified threshold α. We provide more details in Section 6.4.6.

6.4.2 System Overview

We describe how we implement the solution for the optimization problem discussed above in TERA. Algorithm 6.1 describes the main algorithm for TERA and how it uses the main components. It takes a test T, a tuple of hyper-parameters $\boldsymbol{\theta}$ used in the test, and the minimum test passing probability α as its inputs. TERA consists of four main components:

- The *Test Identifier* finds tests which run inference algorithms or training algorithms and contain one or more tunable parameters. For each parameter in $\boldsymbol{\theta}$, we define the valid range of values for the parameter and how to sample the values (Line 2).

- The *Test Instrumentor* modifies the given test by creating placeholders for hyper-parameters that TERA needs to optimize and adding instrumentation code for logging the actual and expected values in the test assertion (Line 3).

- The *Optimizer* executes the Bayesian Optimization algorithm. It runs a test with different parameter configurations several times and finds an optimal parameter configuration which minimizes the running time of the test. We initialize the *Optimizer* (Line 4) using the defined search space, the scoring function (Scorer), the instrumented test T', and a few hyper-parameters like maximum number of evaluation of test (`MAX_EVALS`) and time limit for optimization (`TIMEOUT`).

- The *Scorer* implements the entire optimization problem. It takes the instrumented test T', a tuple of hyper-parameters $\boldsymbol{\theta}$, and the minimum passing probability α as input. Then it runs the test (with the parameter configuration) several times, records the actual and expected values in the assertion, and computes the probability of the assertion passing. If the probability of passing is greater or equal to the user specified threshold (α), then the Scorer returns the average run time of the tests as output. Otherwise, the Scorer returns infinity (∞). The *Scorer* is passed as the scoring function by the optimization algorithm.

6.4.3 Test Identifier

To run TERA, we first need to identify parameters in the test which directly affect the running time of the test and the accuracy of the result. For instance, to run an inference algorithm like Stochastic Variational Inference (SVI) in *Pyro*, the developer needs to set the number of iterations to run and the learning rate of the Adam Optimizer (which is a variant of Stochastic Gradient Descent). Similarly, to run a reinforcement learning algorithm in *ML-Agents*, the developer needs to choose hyper-parameters like the number of iterations,

batch size, and learning rate. We identify such parameters manually in the test and use TERA to tune them. For instance, we look for parameters that match the following patterns: *samples*, *iterations*, *epochs*, *batch size*, *learning rate*, *num passes*, and *chains*.

We also need to identify assertions in the test which TERA can use to determine the reliability of the test results. In particular, we look for assertions which perform approximate comparisons between expected and actual values. This notion of approximate assertions is similar to the ones used in previous works [53, 256]. One example of such an assertion is the Python assert statement of the form: `assert` $a < | > | <= | >= b$. Other examples include numpy APIs like `assert_allclose` and `assert_almost_equal`, and unittest APIs like `assertLess` and `assertGreater`. One difference with previous works is that we also consider assertions that check for *exact equality*. However, we limit it to cases where the assertions check some property of a trained model or inference.

Overall, we typically spend 1-2 hours per project on average to identify tests with suitable hyper-parameters and assertions as described above. We anticipate that the developers who have familiarity with their projects will identify such tests much faster.

6.4.4 Test Instrumentor

For the given test T in a project, and hyper-parameters $\boldsymbol{\theta}$, the Test Instrumentor performs two tasks. First, it replaces the original values of each parameter with a placeholder, which will be used by TERA to set new parameter values and run the test. Second, it adds statements to record the actual and expected values in the test assertions (specified by the user). This step ensures TERA can later reproduce the executions, by simply reading the values from the logs and reason about the distribution of values. The Test Instrumentor can handle logging any scalar, vector, or tensor objects. We use Python's *AST* library [336] to implement the Test Instrumentor.

6.4.5 Optimizer

TERA uses Bayesian Optimization to optimize the running time of the tests. In this work, we use *Tree Parzen Estimators* (TPE) Algorithm [326], which is a variant of Bayesian Optimization [324]. To use this algorithm, we need to provide:

- **Legal Parameter Values:** First, we need to define the space of legal parameter values that the optimization algorithm uses to sample the parameter values. To use the TPE algorithm, we need to specify a distribution for each parameter that will be used for

sampling. We use three kind of parameters spaces in this work: (1) Continuous bounded interval, e.g., $x \in [1e^{-5}, 1.0]$, (2) Discrete bounded interval, e.g., $x \in [100, 1000]$, (3) Discrete choices, e.g., $x \in \{1, 2, 3, 4, 5\}$. For the parameters with continuous bounded interval (e.g., learning rate), we use a *log-uniform* distribution so that it samples values of different orders. For parameters with discrete bounded intervals (e.g., iterations) or discrete choices (e.g., batch size), we use a *uniform* distribution for sampling.

We manually define the bounds of the distribution based on the kind of parameter. For instance, for parameters like *iterations* and *number of samples*, we choose the upper bound to be the default value of the parameter and lower bound to 100 (or 1 if default is less than 100). For parameters like *learning rate*, we choose the lower and upper bounds as $1e^{-5}$ and 1.0 respectively.

- **Objective Function:** Second, we need to define an objective function, which takes as input a set of new parameter values proposed in an iteration by the algorithm and returns a score which intuitively evaluates the goodness of a given set of parameter values. Since we are concerned with reducing the run time of the tests, we could just return the execution time of the test as the score. However, it is insufficient to run the test once. We must also ensure that the test passes with *high probability*. Otherwise, we might obtain a faster, but highly flaky test.

6.4.6 Scorer

The Scorer module encodes the optimization problem. Algorithm 6.2 describes the Scorer algorithm. First, it replaces the parameter placeholders in the instrumented test T' with the actual values in $\boldsymbol{\theta}$ and creates a concrete version of the test T_C (Line 2). Next, it initializes the set of samples S to an empty set (Line 4). Then it iteratively runs the test, collects samples, and computes the score (Lines 5–17). We next describe how the Scorer decides *how many times to run the test* and *how to compute the probability of passing*.

Collecting samples from the distribution. We need to determine whether a version of the test is too flaky. Machine Learning algorithms do not come with formal specifications of accuracy which makes it hard to determine the correctness of any implementation. Hence, we use the assertions in the test as specifications of correctness. The Scorer collects the actual and expected values of the assertion from each run of the test (Lines 7–11). Then it applies the convergence test (Lines 12–15) to determine whether we have enough samples of actual values to reason correctly about the distribution i.e. whether the distribution has converged. If the convergence test fails, we continue running the test more times until the distribution

Algorithm 6.2: Scorer Algorithm

Input: Instrumented test T', Parameters $\boldsymbol{\theta}$, Min. Passing Probability α
Output: Score C

```
1: procedure SCORER(T', θ, α)
2:     T_C ← setParameters(T', θ)
3:     i ← 0
4:     S ← ∅
5:     while i < MAX_ITERS do
6:         b ← 0
7:         while b < BATCH_SIZE do
8:             sample ← ExecuteTest(T_C)
9:             S ← S ∪ {sample}
10:            b ← b + 1
11:        end while
12:        score ← ConvergenceScore(S)
13:        if score < CONV_THRESHOLD then
14:            break
15:        end if
16:        i ← i + BATCH_SIZE
17:    end while
18:    TPP ← ComputeProbPass(S)
19:    if TPP ≥ α then
20:        return AvgRunTime(S)
21:    end if
22: return ∞
23: end procedure
```

converges. We compute the probability of passing of the test (Line 18). If this computed probability is above or equal to α, then we compute and return the average running time of the test executions (Line 20), otherwise we return ∞ (Line 22).

Computing the probability of passing. Given a set of samples, we need to determine the probability that the assertion passes. To compute the probability of passing, we perform the following steps. First, we fit a distribution to the set of samples. Since, we may not know the exact shape of the distribution, we try and fit a number of distributions and choose the one with the best fit (maximum likelihood). In our experiments we used the following distributions: *normal*, *exponential*, *gamma*, *pareto*, *student-t*, *lognorm*, *log uniform*, *log normal*, and *truncated normal*. In contrast, Dutta et al. [53] used empirical distribution to fit the samples; however, empirical distributions are not suited for computing the tails of a distribution. Next, we compute the probability that a sample from the fitted distribution is within the assertion threshold. For instance, for an assertion of the form: `assert` $x < \gamma$, we obtain a distribution $\mathcal{D}$ fitted on the samples of x. Then we compute the cumulative distribution frequency: $\texttt{CDF}(\mathcal{D}, \gamma)$, which is also the probability of passing of the test : $\Pr(x < \gamma)$. For equality assertions, we only compute the percentage of times the actual and

Table 6.1: Project Details

Project	Description	#Tests	%Time
autokeras [337]	ML architecture tuning	2	2.89%
bambi [338]	Bayesian Modelling	2	13.50%
cleverhans [339]	Adverserial Attacks for ML models	5	75.72%
fairseq [340]	Seq-to-Seq Modelling	2	0.58%
gensim [341]	Topic Modelling Library	10	25.88%
gpytorch [342]	Gaussian Process Modelling	9	44.99%
imbalanced-learn (im.-learn) [343]	Learning over Imbalanced Datasets	2	1.89%
ml-agents [46]	Training ML agents	14	68.04%
numpyro [232]	Probabilistic Programming	13	24.04%
parlai [344]	Dialog AI modelling	29	5.53%
pyGPGO [345]	Bayesian Optimization	3	85.00%
pymc3 [234]	Probabilistic Programming	18	14.85%
pymc-learn [346]	Probabilistic Machine Learning	8	5.82%
pyro [347]	Probabilistic Programming	22	26.36%
sbi [348]	Simulation Based Inference	21	84.28%
Total/Avg		160	31.96%

expected values match exactly to derive the probability of passing of the test.

6.5 METHODOLOGY

Selection of projects. For this work, we focus on two probabilistic programming systems: Pyro [76, 258] and PyMC3 [234, 259], and two machine learning frameworks: PyTorch [231] and TensorFlow [78]. We look for tests in these projects as well as their dependent projects using GitHub's API. Among the dependent projects [4], we select projects with at least 10 stars and manually inspect them to search for tests. Since TensorFlow and PyTorch have a very large set of dependents, we only inspect the top 30 dependent projects (based on stars) for each. For PyMC3 and Pyro, we inspect 8 and 5 dependent projects respectively. Overall, we end up with 71 unique projects. Out of these, we exclude 14 projects which are not maintained, require special build systems (e.g., bazel) to build/run tests, or need special hardware (e.g., Raspberry Pi).

For each remaining project, we install the project locally and run its test-suite using *pytest* [272] to obtain the test run-times. We then sort the tests based on run-time in decreasing order and inspect them to check if they fit our criteria (Section 6.4.3). We filter out tests which run for less than five seconds since they are already inexpensive. We also exclude tests which run for more than 15 minutes. These involve tests which are typically run on GPUs on Continuous Integration servers and run considerably slower when run on CPUs

[4]We use the dependent "packages" as reported by the GitHub API, which are projects that can compile into reusable libraries. Packages are more likely to be actively maintained by developers and have reasonable test suites.

(which we use for our evaluation). We exclude tests if their parameters have a low value (e.g., 1-2 iterations). We exclude a project if it has no such expensive tests. We excluded 15 projects based on this criteria. Finally, among the remaining projects, we find several suitable tests in Pyro and PyMC3. Among the dependent projects, we found tests in 3 PyMC3 dependents, 3 Pyro dependents, 4 PyTorch dependents, and 3 TensorFlow dependents. Overall, we find 160 tests in these projects. In these tests, we find 17 unique parameters. The top five parameters (and their occurrences) are *learning-rate* (74), *batch_size* (49), *num_samples* (46), *num_epochs* (31), and *num_steps* (28).

Table 6.1 shows the details for these projects. Column **Project** presents the base name of the project. Column **Description** presents the main utility of the project. Column **#Tests** presents the number of tests we find in each project. Column **%Time** shows the portion of the total test-suite run-time consumed by the selected tests. *We observe that these tests consume more than 31% of the run-time of the whole test-suite. Hence, optimizing these tests can significantly reduce the run-time of the test-suites.*

Mutation Testing. Mutation Testing [349, 350] is an approach for evaluating the effectiveness of a test suite using artificial injected faults. Mutation testing approaches apply simple mutation operators on source code, e.g. changing arithmetic operators, mutating constants, mutating expressions, etc. We use mutation testing analysis to compare the effectiveness of the original and the optimized versions of the tests.

For each project, we select the subset of tests that we are able to optimize using TERA. We compute the line coverage of this subset of tests (original version) and generate mutant versions of code by applying mutation operators only on the covered lines. We run both the original and optimized test suite on these mutants and compute the mutation score:

$$Mutation\ score = \frac{Mutants\ Killed}{Total\ No.\ of\ Mutants}\% \tag{6.4}$$

To account for the randomness in the analysis (some mutants may be killed/survive by chance), we run the analysis on each project 20 times, and then report the average and standard deviation of the mutation scores.

Extracting historically failed tests. For a given project, we obtain the set of most recent 200 failed builds on Travis CI. For this step we use the GitHub Actions API [351] to fetch the builds. We use the Travis API [320] when the former is not available for a project. Since TERA's optimization mainly targets algorithm parameters, we focus only on builds which contain one or more assertion errors and filter out builds failing due to configuration errors or syntactic errors (like type error or out of bound errors). For this step, we developed a simple Python script to parse the build logs and search for assertion errors. Next, among

the builds with assertion errors, we manually look for tests that failed and contain one or more tunable parameters for ML algorithms. For each such test, we find the failing version of the code from the build information and try to reproduce the failure locally. If this step works, we also find the most recent version of code before the build where the test passes. Finally, we run TERA to optimize these tests using the passing version of code and check if the optimized test reproduces the failure in the failing version of code. Overall, we find 12 such tests.

Experimental Setup. For all our experiments, we used 32 core machines with 3.7 GHz Intel processors and 64 GB memory on Azure. For PyMC3, we used machines with larger memory (144 GB) since its tests are more memory-intensive. For the main algorithm (Section 3.3.2), we set the maximum evaluation at 5000 and timeout for the search process at 100 minutes every round. We choose 99% as the minimum probability of passing (Section 6.4.6) in all cases. For the convergence test (Section 6.3.2), we choose a threshold of 1.0 for the Geweke Diagnostic metric, maximum iterations as 500, initial batch size of 30, and update batch size of 30. We implemented TERA entirely using Python. We used the HyperOpt python package [352, 353] for Bayesian Optimization.

6.6 EVALUATION

We answer the following research questions:

RQ1 How much does TERA reduce the run-time of the tests?

RQ2 What is the impact of TERA on the fault detection capability of the tests?

RQ3 How does TERA's optimization impact the reproduction of historically failed builds?

RQ4 What is the run-time of TERA?

RQ1: Run-time Reduction Obtained by TERA

We apply TERA on 15 projects selected using the methodology from Section 6.5. For each project, we find tests that have one or more tunable parameters. We identified 160 such tests (most run an inference algorithm or a training algorithm).

Table 6.2 presents the results for the amount of run-time reduction TERA obtains for the selected tests. Column **Project** presents the project name. Column **#Tests** presents the number of tests we considered. Column **Mean Speedup** presents the geometric mean speedup TERA obtained. Column **Max Speedup** presents the maximum speedup TERA

Table 6.2: Run-time improvements of tests obtained by TERA

Project	#Tests	Mean Speedup	Max Speedup	Original Run-time	Optimized Run-time
autokeras	2	1.08x	1.16x	33.40s	30.66s
bambi	2	1.39x	1.95x	56.64s	44.27s
cleverhans	5	1.30x	1.40x	26.74s	20.10s
fairseq	2	1.22x	1.23x	3.97s	3.24s
gensim	10	1.35x	4.52x	162.89s	132.81s
gpytorch	9	1.97x	3.38x	38.45s	17.25s
im.-learn	2	1.43x	1.99x	10.22s	5.93s
ml-agents	14	2.21x	6.17x	811.60s	354.58s
numpyro	13	1.41x	6.82x	279.49s	178.85s
parlai	29	1.10x	2.42x	269.19s	212.71s
pyGPGO	3	3.23x	5.19x	262.87s	54.37s
pymc-learn	8	1.98x	5.08x	494.56s	254.25s
pymc3	18	2.13x	12.78x	469.89s	224.14s
pyro	22	9.94x	93.65x	3039.84s	495.94s
sbi	21	3.22x	7.50x	2221.73s	769.90s
Total/Avg	160	2.23x	93.65x	545.43s	186.60s

obtained for any test. Column **Original Run-time** presents the total running time of the original version of the tests. Column **Optimized Run-time** presents the total running time of the optimized version of the tests. The last row presents the total number of tests, overall geometric mean speedup, overall maximum speedup, average running time of the original tests, and average run time of the optimized tests.

From Table 6.2, we observe that TERA significantly reduces the run-time of the tests in a majority of cases. Overall, TERA obtains an average reduction of 2.23x across all projects. For Pyro, TERA obtains the highest average reduction (9.94x), with a maximum of 93.65x (reducing from 322s to 3s) for one test. Out of 160 tests, TERA was able to optimize 133 tests, with more than 10% speedup in 119 cases and more than 50% speedup in 79 cases. The results show TERA can significantly reduce the running time of the tests while still ensuring that the tests pass with high probability.

Tests that TERA optimized. Among the tests that were optimized by TERA, parameters like number of sampling iterations in inference algorithms (like MCMC) and the maximum number of training iterations were mostly reduced. It is commonly known that these parameters directly influence how long the algorithms (and consequently the tests) will run. However, reducing these parameters alone is not sufficient.

We observe that in most cases, TERA finds the configuration with the maximum speedup if it adjusts one or more associated parameters as well. One such parameter is learning rate of optimizers (e.g., Adam [354], Adagrad [355]). The learning rate controls how fast the

inference/training updates the weights based on computed gradients in each round. A higher learning rate can often overshoot the optimal point whereas a lower learning rate can make convergence very slow. Another example parameter is batch size. Smaller batch sizes enable faster training through parallelization but can return non-optimal solutions. Large batch sizes can lead to optimal solutions at the cost of slow convergence. These trade-offs hence also influence how long the test needs to run to match expected results. TERA enables the developers to effectively navigate this trade-off space while still ensuring the test passes with high probability.

In our evaluation we find multiple cases where developers significantly over-estimate the number of iterations/samples required for obtaining the desired results in the tests. For instance, in a test for variational inference [356] in Pyro, developers run variational inference on a simple model using a simple loss function: Radial Basis Function (RBF) [357] of size 1. However, the developers initially specified 25000 iterations (learning rate: 2e-4) for inference (which takes 322s to run), which is much more than what is needed for inference to converge. TERA finds that running 100 iterations with learning rate 0.09 is enough for the model to converge and pass the desired accuracy in the test and takes only about 3s. We observe similar patterns in other projects as well, indicating that developers are often too conservative.

Tests that TERA did not optimize. Among the tests for which the speedup was less than 10% (mostly in projects like *parlai*, *autokeras*, *bambi*, and *imbalanced-learn*) we observed that in some cases, TERA did optimize the parameters to some extent, but that alone did not reduce the running time of the tests by much. There can be several reasons behind this. For instance, in some cases, other parts of the test like initialization and setup contribute to the majority of the test run-time. In a few other cases, there are other parameters which affect the running time more, but were not exposed in the test itself. In some tests, the parameters were already at their optimum value (like *parlai*), hence making too many adjustments causes the tests to fail more often than the allowable threshold.

Developer Responses. Due to limited time, we randomly sampled projects which had high speedups and spread the pull requests among them. We intended to have over 20% of the tests sampled from the test population. Overall, we selected 37 tests across 7 projects and sent Pull Requests to their developers. So far, 24 tests have been accepted and merged into the projects, 9 rejected, and 4 are still pending developer responses. For the cases that were rejected, the developers thought that the gains (in testing time) were not significant enough for them to accept our changes. Developer responses reflect that they are often open to accepting changes in hyper-parameters in the tests, if they can provide significant gains.

Table 6.3: Mutation testing scores

Project	#Mutants	Original	Optimized
autokeras	274	50.00% (±0.00)	50.00% (±0.00)
bambi	770	60.14% (±3.25)	62.55% (±5.36)
cleverhans	185	62.19% (±0.12)	64.16% (±0.69)
fairseq	3374	16.38% (±2.13)	16.39% (±2.13)
gensim	1075	27.88% (±5.72)	26.53% (±5.53)
gpytorch	555	61.25% (±0.32)	63.32% (±0.68)
im.-learn	457	34.57% (±0.00)	35.23% (±0.00)
ml-agents	724	61.16% (±0.11)	62.44% (±0.03)
numpyro	566	60.60% (±0.00)	61.11% (±1.73)
parlai	335	59.51% (±0.39)	58.81% (±0.00)
pyGPGO	102	67.65% (±0.00)	68.63% (±0.00)
pymc-learn	91	68.68% (±2.49)	74.18% (±2.44)
pymc3	750	48.40% (±0.00)	58.01% (±0.03)
pyro	443	47.40% (±0.00)	48.31% (±0.00)
sbi	346	66.46% (±1.43)	67.65% (±1.46)
Average		52.82%	54.49%

RQ2: Fault Detection Ability of Optimized Tests

Modifying the tests written by developers can impact the capability of the tests in catching regressions in code. In this research question, we study the impact of TERA's optimizations on the fault detection ability of the tests. We describe our approach and the results next.

For each project, we generate several buggy versions (mutants) of the code using the methodology outlined in Section 6.5. We use the Mutmut tool [358] for mutation testing of our projects. To control the cost of mutation testing, we apply mutations only on code related to main inference or training algorithms. We leave out code for utility functions since they are usually almost equally shared across most tests. For projects with longer running times (*sbi*, *pyro*, and *pymc-learn*) we choose the top 50% of the most optimized tests.

Table 6.3 shows the average (and standard deviation) of mutation scores across 20 runs for each project. It also shows the number of mutants generated per project (Column **#Mutants**). We observe that the average mutation scores remain the same or improve slightly in 13 out of 15 projects. We also perform Student's t-test [359] to check the hypothesis that mutation score of optimized test-suite is smaller than original. Interestingly, it rejects the hypothesis in 14 cases, including gensim which has high variance. The improvement in mutation scores reflects that optimizing the tests can make them tighter and help them detect more regression bugs, which would otherwise be hidden when running for longer cycles. **PyMC3** is an extreme case, in which the mutation score improves by almost 10%. Our investigation found that **PyMC3** developers often set a very high number of sampling

iterations (>5000) in the tests. Thus, small variations in computations (caused by faults in the system) can often remain hidden during long cycles of test execution. However, the end-user will experience regressions in performance when using the tool to solve real-world tasks.

The mutation scores regress by about 1-2% in 2 projects (gensim and parlai). This is not unexpected, since we allow the tests to have a minimum probability of passing of 99% during optimization. As a result, the tests might run for fewer cycles than necessary to catch subtle regression bugs. For example, some faults only surface up when the error propagation exceeds the expected threshold leading to a failure. The developer however can opt for a higher passing threshold if required. We discuss more about such examples from our mutation study in Section 6.7.1.

Overall, we observe that the mutation scores are roughly around 52-54%, which indicates many mutants survive (i.e. not killed). This behavior can potentially be attributed to the probabilistic nature of the ML algorithms. This means that some mutations can generate valid approximations of the software which still meet the desired accuracy specifications (i.e. tests in our case), as observed previously by Hariri et al. [142] in general approximate software.

RQ3: Reproducing Historical Failures

In this research question, we evaluate if we apply TERA to the historical versions of tests in these projects, do they still fail when the original versions failed in historical builds.

We obtain 12 failing tests across 6 projects using the methodology outlined in Section 6.5. For each test, we run TERA to optimize the test (using the passing version). Finally, we report how many of the optimized tests reproduce the failure in the failing code version.

Table 6.4 shows the result for this experiment. Column **Passing SHA** shows the commit hash of the version of the code where the test passes. Column **Failing SHA** shows the commit hash of the version of code where the original test fails. Column **#Tests** shows the number of tests which failed in the failing version and we optimized using TERA. Column **#Reproduced** shows the number of tests which were optimized and reproduced the failure in the failing version of code. We observe that in all cases we are able to optimize the original version of the test using TERA. The optimized tests also reproduce the failure in the failing version of code in all cases. This demonstrates TERA's optimized tests can reproduce real failures.

Table 6.4: Reproducing historical failures

Project	Passing SHA	Failing SHA	#Tests	#Reproduced
gensim	0027fb5	3db9406	3	3
ml-agents	82ea74f	3f4b2b5	1	1
numpyro	71532cc	b5d548b5	1	1
pyro	f9dee1e2	7f84f19	2	2
pyGPGO	1c718d	c21120	1	1
sbi	86d9b07	c8aec2f	1	1
sbi	1534cff	fa705c0	3	3
Total			12	12

RQ4: Efficiency of TERA

We analyze the amount of time TERA's optimization algorithm takes to find optimal parameters.

Table 6.5: Running times for optimization

Project	#Tests	Avg. Time	Med. Time	Avg. #Iters	Avg. #Params	Avg. Runs	Avg. Test Run-time
autokeras	2	2m4s	2m4s	6	1.0	30.00	16.70s
bambi	2	2h53m40s	2h53m40s	39	3.0	30.00	28.32s
cleverhans	5	43s	48s	3	1.0	91.50	5.35s
fairseq	2	25s	25s	12	1.0	30.00	1.98s
gensim	10	1m17s	57s	5	1.0	30.00	16.29s
gpytorch	9	5m43s	3m27s	60	2.0	30.00	4.27s
im.-learn	2	3m38s	3m38s	47	1.5	30.00	5.11s
ml-agents	14	44m36s	36m21s	103	2.9	30.73	57.97s
numpyro	13	1h0m34s	51m29s	99	1.7	30.00	21.50s
parlai	29	29m4s	15m49s	94	3.0	34.03	9.28s
pyGPGO	3	8m26s	3m42s	2	1.0	30.00	87.62s
pymc-learn	8	2h36m1s	3h30m38s	25	1.6	34.29	61.82s
pymc3	18	3h16m18s	3h28m10s	46	3.1	30.00	26.10s
pyro	22	17m2s	5m25s	21	2.0	45.63	138.17s
sbi	21	1h10m53s	1h3m7s	84	1.9	51.69	105.80s

Table 6.5 presents the measurements. Column **Avg. Time** shows the average time taken by TERA for a complete run of the optimization algorithm (Section 3.3.2) – i.e., until it either exhausts evaluating all configurations in the search space or reaches a terminating condition such as exceeding maximum function evaluations (`MAX_EVALS`) or exceeding the allotted time limit (`TIMEOUT`). Column **Med. Time** shows the median optimization time. Column **Avg. #Iters.** shows the average number of iterations taken by the optimization algorithm. Column **Avg. #Params** shows the average number of parameters per test. Column **Avg. Runs** shows the average number of test runs in a single optimization round.

Column **Avg. Test Run-time** shows the average run-time of the original test.

For 10 projects, the average optimization time is less than an hour. For 5 projects (sbi, bambi, pymc-learn, numpyro, and pymc3), the average optimization time is more than an hour. **PyMC3** and **Bambi** have a higher average number of parameters, and we chose a wide range of legal values for each parameter. A developer with more domain experience could select a smaller range of values. **Numpyro** also requires high number of iterations due to its large search space. **Sbi**'s tests have a high running time (>100 seconds) and the flakiness of the tests increases TERA's iterations. Similarly, **pymc-learn** also has tests with high run-time (>60 seconds).

6.7 DISCUSSION

6.7.1 Fault Detection Ability of Optimized Tests

Modifying the tests written by developers can have an adverse impact on the fault detection ability of tests [360]. We can characterize the fault detecting effectiveness of a test using the following metrics: True Positive (TP) – failing on real faults, False Positive (FP) – failing when no faults, True Negative (TN) – not failing on no faults, and False Negative (FN) – not failing on real faults. TERA improves or retains the TP rate of the test in most cases, as shown by our mutation study from RQ2, and can increase the FN rate in some cases (where the mutation score drops). Since we use a minimum passing threshold of 99%, the FP rate may increase slightly (similarly TN rate would reduce by a small amount). Even though the optimized tests may regress in some of these factors, the gap in effectiveness is very small in practice and would only miss faults which require very rare executions to manifest as test failures. These observations along with our study on reproducing historical bugs from RQ3 show that TERA's optimized tests are highly reliable. We discuss a strategy to alleviate some of these adverse effects in Section 6.7.2.

We manually inspect some mutants from our mutation study for cases: 1) when our optimized test catches a bug missed by original test and 2) when our optimized test misses a bug caught by original test. We identified two trends: for the first case, we observe that small variations in computations (due to faults in the system) can remain undetected during long cycles of execution in the original test, whereas they are detected by the optimized version which has a tighter execution since it runs for fewer cycles. For the second case, we observe that some faults only manifest as failures when the error accumulated exceeds a certain threshold – these are detected by the original test which runs for sufficient cycles but are sometimes missed by the optimized test.

6.7.2 Composite Test Running Strategies

We can mitigate some of the adverse affects of optimization by using composite running strategies. For instance, we can *re-run on failure*: first run the optimized test (which is correct with probability α), and if it fails, run the original test, which succeeds with typically higher probability ($> \alpha$). This composite execution is, on average, faster than executing the original test, and can still retain the effectiveness of the original test. In this case, the expected run-time of the test will be: $\alpha \cdot T_{opt} + (1-\alpha) \cdot (T_{opt} + T_{orig})$, where T_{opt} and T_{orig} are the run-times of the optimized and original versions of the test respectively. Since we typically restrict $1-\alpha$ to a small value (e.g., less than 1%), the second term does not increase the run-time significantly. For instance, for Pyro this would only increase the total run-time of optimized tests from 495.94s to only 526.34s. Overall, using this composite strategy increases the running time of our optimized tests across all projects by only 3% on average. This strategy can help reduce flaky failures (false positives) and increases the chances of detecting genuine faults (true positives).

6.7.3 Comparison of Different Search Methods

In this work, we use Bayesian optimization for efficiently searching for optimal hyper-parameters (Section 6.4.2). Researchers also commonly use other search methods such as random search or binary search. We compare our main optimization results against a version of TERA which uses these two alternatives instead of Bayesian optimization.

Random search is a method which uniformly samples from the search space of hyper-parameters to find optimal results. For random search method, we use the same configuration for TERA as the main evaluation (see Section 6.5). Binary search method evaluates the middle element in the value interval for a given parameter (such as iterations) and proceeds with either half of the interval depending on whether the objective function evaluates to true (choose lower half) or false (choose upper half). The search continues until the interval is reduced to a single element. Since binary search cannot optimize multiple parameters simultaneously, our implementation optimizes one parameter at a time (keeping others fixed), then uses the optimal value found when optimizing the next parameter. We only choose parameters with a bounded discrete interval such as *iterations* and *number of samples* for optimization using binary search. We ignore parameters such as *learning rate*, since optimizing such parameters in isolation has no direct effect on test's run time.

Table 6.6 presents the results for this experiment. First column shows the name of the project. For each search method, its two sub-columns show the average speedup (geometric

Table 6.6: Comparison of different search methods

Project	BayesOpt		Random		Binary	
	$\mathbf{Spd}_{\text{test}}$	$\mathbf{T}_{\text{TERA}}$	$\mathbf{Spd}_{\text{test}}$	$\mathbf{T}_{\text{TERA}}$	$\mathbf{Spd}_{\text{test}}$	$\mathbf{T}_{\text{TERA}}$
autokeras	1.08x	2m4s	1.07x	2m18s	1.00x	19s
bambi	1.39x	2h53m40s	1.32x	3h34m16s	1.19x	1h11m9s
cleverhans	1.30x	43s	1.17x	1m20s	1.25x	1m23s
fairseq	1.22x	25s	1.10x	4m22s	1.01x	43s
gensim	1.35x	1m17s	1.26x	2m55s	1.27x	53s
gpytorch	1.97x	5m43s	1.64x	9m5s	1.60x	1m8s
im.-learn	1.43x	3m38s	1.29x	7m5s	1.07x	1m0s
ml-agents	2.21x	44m36s	2.11x	1h10m4s	2.09x	8m23s
numpyro	1.41x	1h0m34s	1.40x	37m20s	1.38x	10m32s
parlai	1.10x	29m4s	1.06x	33m8s	1.05x	36s
pyGPGO	3.23x	8m26s	3.20x	24m38s	3.17x	10m52s
pymc-learn	1.98x	2h36m1s	1.50x	3h10m21s	1.46x	1h55m15s
pymc3	2.13x	3h16m18s	2.06x	3h38m9s	1.92x	1h53m27s
pyro	9.94x	17m2s	7.70x	1h27m16s	2.71x	29m4s
sbi	3.22x	1h10m53s	1.72x	53m49s	1.60x	11m21s
Avg	2.23x	58m27s	1.89x	1h12m20s	1.59x	26m59s

Here, $\mathbf{Spd}_{\text{test}}$ is the Avg. Speedup (Geo-mean) of the optimized tests and $\mathbf{T}_{\text{TERA}}$ is the Avg. Time (Arithmetic Mean) that TERA takes per project and per search method.

mean) of the optimized test and average time (arithmetic mean) for running the optimization algorithm, respectively.

We observe that Bayesian optimization outperforms both random search and binary search methods. Although random search reduces the execution time of tests, it finds a less optimal parameter setting than Bayesian optimization for all projects. This is not surprising since random search, unlike Bayesian optimization, does not learn from results obtained in earlier rounds to adapt the search process. Compared to Bayesian optimization, random search takes more time to finish in 13 projects and less time in 2 projects. We observe that binary search is less effective overall and provides lower speedups in all projects than Bayesian optimization and in 13 projects than random search. It is faster than other methods since it evaluates fewer parameter values or combinations of parameter values. We conclude that binary search may only be suitable for tests with few parameters with discrete bounded intervals.

6.7.4 Gains of Optimization

We anticipate that the cost of running TERA can be easily amortized through the daily savings developers will get in build/test time on CI servers. Table 6.7 shows the number of builds developers currently trigger per day (**Builds/day**) and the savings developers would

Table 6.7: Savings on build/test time per day using TERA

Project	**Builds/day**	**Savings/day**
autokeras	1	2s
bambi	1	12s
cleverhans	1	6s
fairseq	2	1s
gensim	1	30s
gpytorch	1	21s
im.-learn	1	4s
ml-agents	11	1h23m47s
numpyro	5	8m23s
parlai	18	16m56s
pyGPGO	1	3m28s
pymc-learn	1	4m0s
pymc3	1	4m5s
pyro	1	42m23s
sbi	1	24m11s

get if they use TERA's optimized tests instead of original tests for their builds (**Savings/day**). We compute **Savings/day** as: *Builds/day* $\times$ (*Original Run-Time* $-$ *Optimized Run-Time*). The *Original* and *Optimized* run-times can be obtained from Table 6.2.

We observe that TERA can provide large savings for the developers in many projects – more than 80 mins/day for *ml-agents* and more than 40 mins/day for *pyro*. These gains are further enhanced with increasing builds per day. Finally, we expect that developers will run TERA offline (e.g., outside of normal working hours) without impacting their time.

6.7.5 Threats to Validity

In this work, we study only a subset of projects in the ML domain, so our results and observations may not generalize to all projects. To mitigate this threat, we focus on four widely used machine learning frameworks, and their top starred dependent projects, which indicates they have a large user-base and are popular.

We may have missed some tests, in the studied projects, which use ML algorithms and have tunable parameters. To account for this risk, multiple student co-authors independently studied these projects and their test-suites to find tests which fit our criteria. As a result, we obtain a substantial number of such tests.

Optimization is a hard problem which makes it difficult to find the best solution for a given problem. As such, it is possible to further enhance the reduction in run-time.

Changing the developer-set parameters in a test can make the test less reliable (or more *flaky*). We mitigate this risk in two ways. First, we set the minimum passing probability

to 99% during optimization, which ensures test quality does not regress too much. Second, we perform mutation testing of the test suites and show the optimized suite is commonly as good as the original suite. Further, since mutations may not represent real errors, we also show that optimized tests can reproduce real historical failures.

6.8 RELATED WORK

Test Reduction. There is significant research on reducing the test size in terms of the lines of code while preserving the test coverage [361, 362, 363] and reproducing the same bugs [199, 200]. Most of these approaches assume that the test and the program-under-test are deterministic (either natively, or with fixed seeds) and the lines of code are the proxy for the execution time. In contrast, TERA reduces the execution time of tests for machine learning algorithms. TERA finds the optimal parameters (such as learning rates or the numbers of iterations) that lead to reduced execution times of the tests with minor impact on the test's fault-detection ability. To the best of our knowledge, the only existing approach for reducing the parameters of machine learning algorithms along with the program code for the purpose of testing is Storm [254] (for probabilistic programming languages). However, Storm's reduction of parameters uses a simple binary search, and does not consider the scenario of optimizing test run-times.

AutoML Methods. Automated Machine Learning (or AutoML) is a novel approach for automated construction of an end-to-end ML pipeline, using limited computational budget [364, 365]. AutoML methods deal with data preparation, feature engineering, model generation, and model evaluation. The model generation step involves selecting from a set of suitable ML architectures (Architecture Optimization) and choosing optimal model specific hyper-parameters (Hyper-Parameter Optimization) [366, 367]. In these steps, AutoML methods typically aim to optimize the accuracy of the model on a data-set. Unlike AutoML, TERA targets the dual problem of reducing the running time of a test executing a *fixed* ML architecture while *also* preserving the desired passing probability of the test.

Hyper-parameter Tuning For Machine Learning. Bergstra et al. [326] explored various strategies to optimize hyper-parameters for neural networks. They showed that Gaussian Process based Bayesian Optimization methods and the newly proposed Tree Parzen Estimator Algorithm perform better than manual or random search based methods on several difficult data-sets. Snoek et al. [368] proposed a new algorithm based on a Gaussian process based surrogate model for Bayesian Optimization for Machine Learning Algorithms. Their approach also accounts for the cost for each configuration of the learning algorithm during optimization by considering the expected improvement per second in the acquisition function. Maclaurin

et al. [369] introduced a gradient based hyper-parameter optimization technique. Unlike those use-cases, the objective of TERA is to improve performance of software testing, which it accomplishes by maintaining a desired level of reliability (i.e. minimum probability of passing) using statistical machinery while reducing the running time of the test.

Flaky Tests. Flaky tests have emerged as an important problem in software testing – several studies characterized and classified such tests in real-world projects [11, 50, 51, 370, 371, 372, 373], and are considered an important class of bugs in industry [371, 372]. Researchers also developed automated tools to detect [53, 244, 245, 247, 374, 375], and fix flaky tests caused due to test-order dependency [55] and under-determined specifications [376].

Prior work has studied the causes and fixes for flaky tests in open-source software [50, 370]. They studied flaky tests in traditional software, finding that common causes for flaky tests include async wait, concurrency, and test-order dependencies. Romano et al. [373] studied flaky UI tests in web and Android projects- their causes, manifestations, and fixes. In this work, we optimize tests in ML projects while modestly relaxing the desired passing probability of the test. While reducing parameters like number of iterations could potentially make the tests more flaky, TERA ensures that the probability of passing does not degrade beyond an acceptable threshold chosen by the developer.

Lam et al. [249] proposed a technique to handle flakiness in tests due to asynchronous calls. They show that the running time of such tests can be reduced significantly while still retaining similar test failure rate (or flakiness). They run each test only a fixed number of times to determine how often it fails and use simple binary search for finding optimal timeout times. TERA, on the other hand, handles tests which are flaky due to algorithmic randomness. TERA uses convergence tests to determine how many times to run the tests and Bayesian optimization to search for optimal hyper-parameters.

Dutta et al. [230] proposed a method to fix flaky tests in Machine Learning projects by only updating assertion bounds in tests. In contrast, in this work we look at the dual problem of speeding up the test by tuning the hyper-parameters whist preserving a high passing probability. While both hyper-parameters and assertion thresholds can affect the flakiness of a test, only hyper-parameters influence the execution time of the test.

Testing of Systems Dealing with Randomness. Robust machine learning frameworks like TensorFlow [78] and PyTorch [231] have paved the way for rapid development of machine learning based solutions. In recent times, there has also been a surge in interest in probabilistic programming in both academic and industrial research communities. This has led to the development of numerous probabilistic programming languages over the years [3, 67, 69, 71, 72, 73, 75, 76, 85, 124, 188, 190, 193]. Researchers proposed techniques for testing and verifying probabilistic systems [196, 255], machine learning frameworks [12, 250, 251, 252,

309], and randomized algorithms [39] to complement manual test writing. These techniques complement manual test development, but the advances in efficient automated test generation for these systems is yet to catch up with the speed of application development, while capturing the inherent nondeterminism and overcoming the lack of reliable oracles in this domain.

6.9 SUMMARY

We presented TERA, an approach to help developers optimize the running time of the tests which involve stochastic computations. TERA combines techniques from Bayesian Optimization and statistical convergence testing to effectively reduce the running time of the tests while guarding their reliability. Using TERA we obtained more than 2.23x average speedup in 160 tests across 15 projects in Machine Learning domain. We anticipate developers will use TERA for the following main tasks: (1) optimize existing expensive tests, (2) optimize parameters of newly added tests, (3) update hyper-parameters after test modification.

Chapter 7: CONCLUSIONS AND FUTURE WORK

7.1 CONCLUSIONS

Randomness has become an unavoidable aspect of most modern software systems. Modern systems often may contain components that make non-deterministic choices (such as stochastic algorithms) or may interact with an environment that is inherently non-deterministic (such as robot navigating a field). Machine Learning-based systems are a prime example of such systems. The pervasiveness of Machine Learning-based systems in society makes it imperative to ensure their correctness and reliability. However, the randomness in such systems dictates that we rethink the traditional approaches to testing and debugging.

This dissertation presented the first steps towards developing a principled approach to testing and debugging Machine Learning-based systems by leveraging mathematical foundations provided by statistics and probability theory. Through this dissertation, we have shown that we can develop test generation techniques that are effective at detecting bugs, debugging techniques that are efficient in localizing bugs, and techniques that improve reliability and efficiency of regression tests – all whilst accounting for randomness in the system under test. Furthermore, the overwhelming positive response from developers of major ML libraries through pull request acceptances, issue resolutions, and discussions on GitHub have shown that the techniques developed in this dissertation are practical and useful.

Many emerging domains such as robotics, autonomous driving, and augmented/virtual reality also exhibit non-determinism either through the use of stochastic algorithms or by interacting with an inherently non-deterministic environment or agent. The work presented in this dissertation can serve as a guiding light for future testing approaches for such domains. The principles and techniques developed in this dissertation could be instrumental in ensuring the correctness and reliability of future systems that deal with non-determinism.

7.2 FUTURE WORK

As the influence and impact of Machine Learning-based systems continue to gain momentum in society, the principal themes of this dissertation will only grow in importance and applicability. The recently introduced White House's AI Bill of Rights underscores the importance of "extensive testing" in developing Machine Learning-based systems to avoid undue harm to society. Further, there is growing interest in this area both from industry and federal funding programs (e.g., NSF DASS and NSF SHF) acknowledging the importance of

testing ML-based systems.

Next, we describe our vision on how the techniques presented in this dissertation can be leveraged and extended to solve the challenges in testing and debugging components of the Machine Learning stack (e.g., deep learning compilers) as well as systems in emerging domains (e.g., autonomous cars and robots).

Effective Testing and Debugging of Deep Learning Compilers. Deep Learning (DL) compilers automate the compilation and optimization of DL models for heterogenous hardware such as high-end GPUs, FPGAs, and edge devices. Some popular examples include TensorFlow XLA, Apache TVM, TensorFlow Lite, and Facebook Glow. DL Compilers efficiently map DL computations, specified in a high-level language like Python, to a lower-level hardware-specific language. DL Compilers often come with optimizations such as auto-scheduling algorithms [377] that automatically generate high-performance code for a DL model. However, due to the huge search space, they often employ random selection and parallel evaluation of optimization choices. As a result, their results may vary across executions, making them non-deterministic. Bugs in such optimizations are often hard to catch due to the presence of randomness and the large space of potential DL models we need to test them with. Hence, we need a comprehensive and cost-efficient testing approach for DL compilers to detect such hard-to-find bugs, while accounting for underlying randomness. The techniques presented in this dissertation on testing probabilistic programming systems can be adapted for this task (e.g., differential testing, template-based model generation). Additionally, we can combine statistical techniques presented in chapters 5 and 6 to reason about randomness.

Rethinking Regression Testing for ML libraries. While writing regression tests for ML libraries, developers manually choose various configurations (like algorithm hyper-parameters and assertion bounds), often leading to problems like flakiness or reduced fault-detection effectiveness. We envision a test generation approach that, given a test specification, can automatically make these choices while jointly optimizing for one or more goals (such as maximizing fault-detection effectiveness or coverage). Future work can combine insights from this dissertation on regression tests in ML libraries with novel approaches to determine optimal test configurations from the large combinatorial space of choices.

Testing Deep Probabilistic Programs. Deep probabilistic programming [83, 378] is a promising programming paradigm that enables composing deep learning models with probabilistic programs. Such programming systems allow users to effectively combine the benefits of both domains, such as scalability to billions of parameters, AI accelerators, interpretability, and reasoning about uncertainty. While the domain is still in its infancy,

it will require novel testing approaches to enable robust development of the programming platforms. We need empirical studies to investigate and identify the unique challenges in this domain and the common programming errors. We can then combine these insights with the principled testing techniques presented in this dissertation (e.g., ProbFuzz/Storm) to develop automated testing approaches.

Improving Reliability of Hybrid Systems with Machine Learning Components. With the proliferation of powerful Machine Learning models, various real-world systems in emerging domains such as autonomous driving, AR/VR, and robotics have started integrating ML-based components. Testing such hybrid systems presents challenges that are beyond current testing approaches. First, a testing approach should account for how the ML-based components interact with other ML- and non-ML based components, especially in the presence of non-determinism. Second, the business logic of the ML-based components is often learned from data and hence lacks interpretability. Therefore, we require a test oracle that can reason about the correctness of the component in the context of the entire system. To tackle these challenges, we need to develop a testing approach that 1) reasons about component- and system-level non-determinism by modeling their behavior using statistical methods, 2) captures component interactions using probabilistic models, 3) generates test inputs that trigger diverse component behaviors and their interactions, and 4) checks for correctness by adapting system-level specifications to the hybrid setting.

The realm of computing is experiencing a major shift as AI and ML find increasing applications, including code generation. However, coding continues to be considerably challenging, and the importance of code correctness remains unchanged, regardless of whether it originates from humans or machines. Testing and debugging will retain their significance as long as code remains crucial.

REFERENCES

[1] I. Goodfellow, Y. Bengio, A. Courville, and Y. Bengio, *Deep learning*, 2016.

[2] L. P. Kaelbling, M. L. Littman, and A. W. Moore, "Reinforcement learning: A survey," *Journal of artificial intelligence research*, 1996.

[3] N. D. Goodman, V. K. Mansinghka, D. M. Roy, K. Bonawitz, and J. B. Tenenbaum, "Church: a language for generative models," in *Conference on Uncertainty in Artificial Intelligence*, 2008.

[4] A. D. Gordon, T. A. Henzinger, A. V. Nori, and S. K. Rajamani, "Probabilistic programming," *Future of Software Engineering Proceedings*, 2014.

[5] B. Huval, T. Wang, S. Tandon, J. Kiske, W. Song, J. Pazhayampallil, M. Andriluka, P. Rajpurkar, T. Migimatsu, R. Cheng-Yue et al., "An empirical evaluation of deep learning on highway driving," *arXiv preprint arXiv:1504.01716*, 2015.

[6] K. Pei, Y. Cao, J. Yang, and S. Jana, "Deepxplore: Automated whitebox testing of deep learning systems," in *Proceedings of the 26th Symposium on Operating Systems Principles*, 2017.

[7] G. Litjens, T. Kooi, B. E. Bejnordi, A. A. A. Setio, F. Ciompi, M. Ghafoorian, J. A. Van Der Laak, B. Van Ginneken, and C. I. Sánchez, "A survey on deep learning in medical image analysis," *Medical image analysis*, 2017.

[8] J. B. Heaton, N. G. Polson, and J. H. Witte, "Deep learning for finance: deep portfolios," *Applied Stochastic Models in Business and Industry*, 2017.

[9] "Understanding the fatal tesla accident on autopilot and the nhtsa probe," *electrek*, 2016, https://electrek.co/2016/07/01/understanding-fatal-tesla-accident-autopilot-nhtsa-probe.

[10] "A google self-driving car caused a crash for the first time," *The Verge*, 2016, https://www.theverge.com/2016/2/29/11134344/google-self-driving-car-crash-report.

[11] J. M. Zhang, M. Harman, L. Ma, and Y. Liu, "Machine learning testing: Survey, landscapes and horizons," *IEEE Transactions on Software Engineering*, 2020.

[12] H. V. Pham, T. Lutellier, W. Qi, and L. Tan, "Cradle: cross-backend validation to detect and localize bugs in deep learning libraries," in *International Conference on Software Engineering*, 2019.

[13] Z. Wang, M. Yan, J. Chen, S. Liu, and D. Zhang, "Deep learning library testing via effective model generation," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020.

[14] J. Kim, R. Feldt, and S. Yoo, "Guiding deep learning system testing using surprise adequacy," in *IEEE/ACM 41st International Conference on Software Engineering*, 2019.

[15] E. Breck, N. Polyzotis, S. Roy, S. E. Whang, and M. A. Zinkevich, "Data validation for machine learning," in *Conference on Machine Learning and Systems*, 2019.

[16] N. Hynes, D. Sculley, and M. Terry, "The data linter: Lightweight, automated sanity checking for ml data sets," in *NIPS MLSys Workshop*, 2017.

[17] J. Wang, G. Dong, J. Sun, X. Wang, and P. Zhang, "Adversarial sample detection for deep neural network through model mutation testing," in *IEEE/ACM 41st International Conference on Software Engineering*, 2019.

[18] J. H. Metzen, T. Genewein, V. Fischer, and B. Bischoff, "On detecting adversarial perturbations," *ArXiv*, 2017.

[19] J. Guo, Y. Jiang, Y. Zhao, Q. Chen, and J. Sun, "Dlfuzz: Differential fuzzing testing of deep learning systems," in *Proceedings of the 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2018.

[20] Z. Sun, J. M. Zhang, M. Harman, M. Papadakis, and L. Zhang, "Automatic testing and improvement of machine translation," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020.

[21] M. T. Ribeiro, T. Wu, C. Guestrin, and S. Singh, "Beyond accuracy: Behavioral testing of nlp models with checklist," *arXiv preprint arXiv:2005.04118*, 2020.

[22] D. Gopinath, C. S. Pasareanu, K. Wang, M. Zhang, and S. Khurshid, "Symbolic execution for attribution and attack synthesis in neural networks," *IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings*, 2019.

[23] Y. Sun, M. Wu, W. Ruan, X. Huang, M. Kwiatkowska, and D. Kroening, "Concolic testing for deep neural networks," *33rd IEEE/ACM International Conference on Automated Software Engineering*, 2018.

[24] M. T. Ribeiro and S. Lundberg, "Adaptive testing and debugging of nlp models," in *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 2022.

[25] M. J. Islam, G. Nguyen, R. Pan, and H. Rajan, "A comprehensive study on deep learning bug characteristics," in *Proceedings of the 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019.

[26] N. Humbatova, G. Jahangirova, G. Bavota, V. Riccio, A. Stocco, and P. Tonella, "Taxonomy of real faults in deep learning systems," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020.

[27] C. Murphy, K. Shen, and G. Kaiser, "Automatic system testing of programs without test oracles," in *Proceedings of the eighteenth international symposium on Software testing and analysis*, 2009.

[28] C. Murphy, K. Shen, and G. Kaiser, "Using jml runtime assertion checking to automate metamorphic testing in applications without test oracles," in *International Conference on Software Testing Verification and Validation*, 2009.

[29] N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, A. H. Teller, and E. Teller, "Equation of state calculations by fast computing machines," *The journal of chemical physics*, 1953.

[30] W. K. Hastings, "Monte carlo sampling methods using markov chains and their applications," 1970.

[31] C. P. Robert, G. Casella, and G. Casella, *Monte Carlo statistical methods*, 1999.

[32] D. A. McAllester, "Pac-bayesian model averaging," in *Proceedings of the twelfth annual conference on Computational learning theory*, 1999.

[33] O. Sebbouh, R. M. Gower, and A. Defazio, "Almost sure convergence rates for stochastic gradient descent and stochastic heavy ball," in *Conference on Learning Theory*, 2021.

[34] E. Hazan, A. Agarwal, and S. Kale, "Logarithmic regret algorithms for online convex optimization," *Machine Learning*, 2007.

[35] K. Sen, M. Viswanathan, and G. Agha, "Statistical model checking of black-box probabilistic systems," in *Computer Aided Verification: 16th International Conference, CAV 2004, Boston, MA, USA, July 13-17, 2004. Proceedings 16*, 2004.

[36] S. Sankaranarayanan and G. Fainekos, "Falsification of temporal properties of hybrid systems using the cross-entropy method," in *Proceedings of the 15th ACM international conference on Hybrid Systems: Computation and Control*, 2012.

[37] W. Zhang, C. Sun, and S. Lu, "Conmem: detecting severe concurrency bugs through an effect-oriented approach," *ACM Sigplan Notices*, 2010.

[38] M. Gligoric, L. Zhang, C. Pereira, and G. Pokam, "Selective mutation testing for concurrent code," in *Proceedings of the international symposium on software testing and analysis*, 2013.

[39] K. Joshi, V. Fernando, and S. Misailovic, "Statistical algorithmic profiling for randomized approximate programs," in *IEEE/ACM 41st International Conference on Software Engineering*, 2019.

[40] A. Banerjee, S. Chattopadhyay, and A. Roychoudhury, "On testing embedded software," in *Advances in Computers*, 2016.

[41] S. Siegl, K.-S. Hielscher, R. German, and C. Berger, "Formal specification and systematic model-driven testing of embedded automotive systems," in *Design, Automation & Test in Europe*, 2011.

[42] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu, "Finding and reproducing heisenbugs in concurrent programs." in *OSDI*, 2008.

[43] S. Misailovic, D. Kim, and M. Rinard, "Parallelizing sequential programs with statistical accuracy tests," *ACM Transactions on Embedded Computing Systems*, 2013.

[44] M. Gligoric, V. Jagannath, and D. Marinov, "Mutmut: Efficient exploration for mutation testing of multithreaded code," in *Third international conference on software testing, verification and validation*, 2010.

[45] S. Yoo and M. Harman, "Regression testing minimization, selection and prioritization: a survey," *Software Testing, Verification and Reliability*, 2012.

[46] "Ml-agents," 2021, https://github.com/Unity-Technologies/ml-agents.

[47] T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine, "Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor," in *ICML*, 2018.

[48] "Ml-agents minibatch," 2021, https://github.com/Unity-Technologies/ml-agents/blob/0e573a1865d0800ad5cd6649b9bdec99327028a1/ml-agents/mlagents/trainers/sac/trainer.py#L251.

[49] "Ml-agents sample action," 2021, https://github.com/Unity-Technologies/ml-agents/blob/master/ml-agents/mlagents/trainers/torch/action_model.py#L70.

[50] Q. Luo, F. Hariri, L. Eloussi, and D. Marinov, "An empirical analysis of flaky tests," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2014.

[51] M. Gruber, S. Lukasczyk, F. Kroiß, and G. Fraser, "An empirical study of flaky tests in python," in *14th IEEE Conference on Software Testing, Verification and Validation*, 2021.

[52] O. Parry, G. M. Kapfhammer, M. Hilton, and P. McMinn, "A survey of flaky tests," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 2021.

[53] S. Dutta, A. Shi, R. Choudhary, Z. Zhang, A. Jain, and S. Misailovic, "Detecting flaky tests in probabilistic and machine learning applications," in *29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2020.

[54] S. Dutta, A. Arunachalam, and S. Misailovic, "To seed or not to seed? an empirical analysis of usage of seeds for testing in machine learning projects," in *2022 IEEE Conference on Software Testing, Verification and Validation*, 2022.

[55] A. Shi, W. Lam, R. Oei, T. Xie, and D. Marinov, "ifixflakies: A framework for automatically fixing order-dependent flaky tests," in *Proceedings of the 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019.

[56] P. Zhang, Y. Jiang, A. Wei, V. Stodden, D. Marinov, and A. Shi, "Domain-specific fixes for flaky tests with wrong assumptions on underdetermined specifications," in *IEEE/ACM 43rd International Conference on Software Engineering*, 2021.

[57] W. Lam, K. Muşlu, H. Sajnani, and S. Thummalapenta, "A study on the lifecycle of flaky tests," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020.

[58] E. Vicario, "Static analysis and dynamic steering of time-dependent systems," *IEEE transactions on software engineering*, 2001.

[59] S. Bernardi and J. Campos, "Computation of performance bounds for real-time systems using time petri nets," *IEEE Transactions on Industrial Informatics*, 2009.

[60] V. Chandola, A. Banerjee, and V. Kumar, "Anomaly detection: A survey," *ACM computing surveys (CSUR)*, 2009.

[61] N. Ye and Q. Chen, "An anomaly detection technique based on a chi-square statistic for detecting intrusions into information systems," *Quality and reliability engineering international*, 2001.

[62] J. Pickands III et al., "Statistical inference using extreme order statistics," *Annals of statistics*, 1975.

[63] L. De Haan and A. Ferreira, *Extreme value theory: an introduction*, 2007.

[64] M. Fréchet, "Sur la loi de probabilité de l'écart maximum," *Ann. Soc. Math. Polon.*, 1927.

[65] S. Xia, S. Dutta, D. Marinov, , S. Misailovic, and L. Zhang, "Balancing effectiveness and flakiness of non-deterministic machine learning tests," in *IEEE/ACM 45th International Conference on Software Engineering*, 2023.

[66] A. Gelman, D. Lee, and J. Guo, "Stan a probabilistic programming language for bayesian inference and optimization," *Journal of Educational and Behavioral Statistics*, 2015.

[67] W. R. Gilks, A. Thomas, and D. J. Spiegelhalter, "A language and program for complex bayesian modelling," *The Statistician*, 1994.

[68] J. Borgström, A. D. Gordon, M. Greenberg, J. Margetson, and J. Van Gael, "Measure transformer semantics for bayesian machine learning," in *Proceedings of the 20th European conference on Programming languages and systems*, 2011.

[69] G. Claret, S. K. Rajamani, A. V. Nori, A. D. Gordon, and J. Borgström, "Bayesian inference using data flow analysis," in *Proceedings of the 9th Joint Meeting on Foundations of Software Engineering*, 2013.

[70] S. Hershey, J. Bernstein, B. Bradley, A. Schweitzer, N. Stein, T. Weber, and B. Vigoda, "Accelerating inference: towards a full language, compiler and hardware stack," *arXiv preprint arXiv:1212.2991*, 2012.

[71] A. V. Nori, C.-K. Hur, S. K. Rajamani, and S. Samuel, "R2: An efficient mcmc sampler for probabilistic programs," in *AAAI Conference on Artificial Intelligence*, 2014.

[72] F. Wood, J. W. van de Meent, and V. Mansinghka, "A new approach to probabilistic programming inference," in *Proceedings of the 17th International conference on Artificial Intelligence and Statistics*, 2014.

[73] V. Mansinghka, D. Selsam, and Y. Perov, "Venture: a higher-order probabilistic programming platform with programmable inference," *arXiv preprint 1404.0099*, 2014.

[74] "Edward," 2018, http://edwardlib.org.

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

[76] "Pyro," 2018, http://pyro.ai.

[77] "Pytorch," 2018, http://pytorch.org.

[78] "Tensorflow," 2018, https://www.tensorflow.org.

[79] R. B. Grosse, S. Ancha, and D. M. Roy, "Measuring the reliability of mcmc inference with bidirectional monte carlo," in *Advances in Neural Information Processing Systems*, 2016.

[80] D. Selsam, P. Liang, and D. L. Dill, "Developing bug-free machine learning systems with formal mathematics," *arXiv preprint arXiv:1706.08605*, 2017.

[81] G. F. Cooper, "The computational complexity of probabilistic inference using bayesian belief networks," *Artificial intelligence*, 1990.

[82] E. Atkinson and M. Carbin, "Towards correct-by-construction probabilistic inference," in *NIPS Workshop on Machine Learning Systems*, 2016.

[83] D. Tran, M. D. Hoffman, R. A. Saurous, E. Brevdo, K. Murphy, and D. M. Blei, "Deep probabilistic programming," in *International Conference on Learning Representations*, 2017.

[84] "Stan webpage," 2018, http://mc-stan.org.

[85] B. Carpenter, A. Gelman, M. Hoffman, D. Lee, B. Goodrich, M. Betancourt, M. A. Brubaker, J. Guo, P. Li, A. Riddell et al., "Stan: A probabilistic programming language," *Journal of Statistical Software*, 2016.

[86] A. Kucukelbir, R. Ranganath, A. Gelman, and D. M. Blei, "Automatic variational inference in stan," in *Proceedings of the 28th International Conference on Neural Information Processing Systems*, 2015.

[87] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and understanding bugs in c compilers," in *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, 2011.

[88] Q. Zhang, C. Sun, and Z. Su, "Skeletal program enumeration for rigorous compiler testing," in *Proceedings of the 38th ACM SIGPLAN conference on programming language design and implementation*, 2017.

[89] V. Le, C. Sun, and Z. Su, "Finding deep compiler bugs via guided stochastic program mutation," *ACM SIGPLAN Notices*, 2015.

[90] C. Sun, V. Le, and Z. Su, "Finding and analyzing compiler warning defects," in *Proceedings of the 38th International Conference on Software Engineering*, 2016.

[91] A. Balestrat, "CCG - random C Code Generator," 2018, https://github.com/Mrktn/ccg.

[92] Y. Chen, A. Groce, C. Zhang, W.-K. Wong, X. Fern, E. Eide, and J. Regehr, "Taming compiler fuzzers," in *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*, 2013.

[93] Y. Chen, T. Su, C. Sun, Z. Su, and J. Zhao, "Coverage-directed differential testing of JVM implementations," in *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2016.

[94] V. Le, M. Afshari, and Z. Su, "Compiler validation via equivalence modulo inputs," *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2014.

[95] V. Le, C. Sun, and Z. Su, "Randomized stress-testing of link-time optimizers," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, 2015.

[96] C. Lidbury, A. Lascu, N. Chong, and A. F. Donaldson, "Many-core compiler fuzzing," in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2015.

[97] A. Bachmann, C. Bird, F. Rahman, P. Devanbu, and A. Bernstein, "The missing links: bugs and bug-fix commits," in *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, 2010.

[98] M. Rath, J. Rendall, J. L. Guo, J. Cleland-Huang, and P. Mäder, "Traceability in the wild: automatically augmenting incomplete trace links," in *Proceedings of the 40th International Conference on Software Engineering*, 2018.

[99] A. Di Franco, H. Guo, and C. Rubio-González, "A comprehensive study of real-world numerical bug characteristics," in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, 2017.

[100] "Edward commit 972a9d9," 2017, https://github.com/blei-lab/edward/commit/972a9d9.

[101] "Pyro issue:351," 2017, https://github.com/uber/pyro/commit/b94f06a.

[102] "Pyro commit 0f1d27e," 2017, https://github.com/uber/pyro/commit/0f1d27e.

[103] 2017, https://github.com/blei-lab/edward/commit/79f4193.

[104] "Pyro commit 6e26f5e," 2017, https://github.com/uber/pyro/commit/6e26f5e.

[105] "Edward commit fe01657," 2016, https://github.com/blei-lab/edward/commit/fe01657.

[106] "Stan commit 7a98bd2," 2016, https://github.com/stan-dev/stan/commit/7a98bd2.

[107] "Edward commit 43d8a39," 2016, https://github.com/blei-lab/edward/commit/43d8a39.

[108] "Pyro issue 303," 2017, https://github.com/uber/pyro/issues/303.

[109] "Edward commit 002a27e," 2016, https://github.com/blei-lab/edward/commit/002a27e.

[110] "Pyro commit f5a51fe," 2017, https://github.com/uber/pyro/commit/f5a51f.

[111] "Pyro commit 8c14f36," 2017, https://github.com/uber/pyro/commit/8c14f36.

[112] "Stan commit 40c8224," 2016, https://github.com/stan-dev/stan/commit/40c8224.

[113] "Stan commit b8d5086," 2013, https://github.com/stan-dev/stan/commit/b8d5086.

[114] "Edward commit 3616d41," 2016, https://github.com/blei-lab/edward/commit/3616d41.

[115] "Pyro commit 7b6cf58," 2017, https://github.com/uber/pyro/commit/7b6cf58.

[116] "Stan commit 99289c85," 2015, https://github.com/stan-dev/stan/commit/99289c85.

[117] "Stan commit 2fc94d4," 2017, https://github.com/stan-dev/stan/commit/2fc94d4.

[118] "Stan commit ae423b2," 2017, https://github.com/stan-dev/stan/commit/ae423b2.

[119] "Edward bug 347," 2016, https://github.com/blei-lab/edward/commit/10118db.

[120] D. Liew, D. Schemmel, C. Cadar, A. F. Donaldson, R. Zähl, and K. Wehrle, "Floating-point symbolic execution: A case study in n-version programming," in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, 2017.

[121] C. Rubio-González, C. Nguyen, H. D. Nguyen, J. Demmel, W. Kahan, K. Sen, D. H. Bailey, C. Iancu, and D. Hough, "Precimonious: Tuning assistant for floating-point precision," in *SC*, 2013.

[122] "Stan issue #2178," 2017, https://github.com/stan-dev/stan/issues/2178.

[123] D. Jackson and C. A. Damon, "Elements of style: Analyzing a software design feature with a counterexample detector," *IEEE Transactions on software engineering*, 1996.

[124] T. Gehr, S. Misailovic, and M. Vechev, "PSI: Exact symbolic inference for probabilistic programs," in *Computer Aided Verification*, 2016.

[125] "Symmetric mean absolute percentage error. armstrong," 1985, http://www.forecastingprinciples.com/files/LRF-Ch13b.pdf.

[126] "Edward commit 3616d41," 2016, https://github.com/blei-lab/edward/commit/3616d41.

[127] "Pyro commit 7b6cf58," 2017, https://github.com/uber/pyro/commit/7b6cf58.

[128] "Pyro commit 3671b06," 2017, https://github.com/uber/pyro/commit/3671b06.

[129] "Stan commit 04fcb74," 2013, https://github.com/stan-dev/stan/commit/04fcb74.

[130] "Stan commit 45a82fd," 2015, https://github.com/stan-dev/stan/commit/45a82fd.

[131] "Edward commit 79f4193," 2017, https://github.com/blei-lab/edward/commit/79f4193.

[132] "Stan commit 5224850," 2015, https://github.com/stan-dev/stan/commit/5224850.

[133] B. Carpenter, "Stan issue 603," 2017, https://github.com/stan-dev/stan/issues/603.

[134] "Stan language manual. appendix d." 2018, http://mc-stan.org/users/documentation/index.html.

[135] "Stan language manual. chapter 3.1." 2018, http://mc-stan.org/users/documentation/index.html.

[136] "Stan best practices," 2018, https://github.com/stan-dev/stan/wiki/Stan-Best-Practices.

[137] "Stan pedantic mode," 2018, https://github.com/stan-dev/stan/wiki/Stan-Parser-Pedantic-Mode.

[138] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. Engler, "A few billion lines of code later: using static analysis to find bugs in the real world," *CACM*, 2010.

[139] M. Beller, R. Bholanath, S. McIntosh, and A. Zaidman, "Analyzing the state of static analysis: A large-scale evaluation in open source software," in *IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering*, 2016.

[140] T. Barik, J. Smith, K. Lubick, E. Holmes, J. Feng, E. Murphy-Hill, and C. Parnin, "Do developers read compiler error messages?" in *Proceedings of the 39th International Conference on Software Engineering*, 2017.

[141] S. Pearson, J. Campos, R. Just, G. Fraser, R. Abreu, M. D. Ernst, D. Pang, and B. Keller, "Evaluating and improving fault localization," in *Proceedings of the 39th International Conference on Software Engineering*, 2017.

[142] F. Hariri, A. Shi, O. Legunsen, M. Gligoric, S. Khurshid, and S. Misailovic, "Approximate transformations as mutation operators," in *Conference on Software Testing, Validation and Verification*, 2018.

[143] "Pyro issue 875," 2018, https://github.com/uber/pyro/issues/875.

[144] "Edward bug: Sgld produces nan in output," 2018, https://github.com/blei-lab/edward/issues/859.

[145] "Stan issue: Stan throws error with empty array," 2018, https://github.com/stan-dev/pystan/issues/437.

[146] "Stan commit 5845db97," 2016, https://github.com/stan-dev/stan/commit/5845db9.

[147] "Edward commit c9afcb1f," 2017, https://github.com/blei-lab/edward/commit/c9afcb1.

[148] D. Monniaux, "Abstract interpretation of probabilistic semantics," in *SAS*, 2000.

[149] P. Mardziel, S. Magill, M. Hicks, and M. Srivatsa, "Dynamic enforcement of knowledge-based security policies using probabilistic abstract interpretation." *Journal of Computer Security*, 2013.

[150] J. Geldenhuys, M. B. Dwyer, and W. Visser, "Probabilistic symbolic execution," in *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, 2012.

[151] A. Filieri, C. S. Păsăreanu, and W. Visser, "Reliability analysis in symbolic pathfinder," in *Proceedings of the 2013 International Conference on Software Engineering*, 2013.

[152] K. Luckow, C. S. Păsăreanu, M. B. Dwyer, A. Filieri, and W. Visser, "Exact and approximate probabilistic symbolic execution for nondeterministic programs," in *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, 2014.

[153] M. Borges, A. Filieri, M. d'Amorim, and C. S. Păsăreanu, "Iterative distribution-aware sampling for probabilistic symbolic execution," in *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering*, 2015.

[154] N. Jansen, C. Dehnert, B. L. Kaminski, J. Katoen, and L. Westhofen, "Bounded model checking for probabilistic programs," *Automated Technology for Verification and Analysis: 14th International Symposium*, 2016.

[155] S. Sankaranarayanan, A. Chakarov, and S. Gulwani, "Static analysis for probabilistic programs: inferring whole program properties from finitely many paths," *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2013.

[156] C. Morgan, A. McIver, and K. Seidel, "Probabilistic predicate transformers," *ACM Transactions on Programming Languages and Systems*, 1996.

[157] C. Nandi, D. Grossman, A. Sampson, T. Mytkowicz, and K. S. McKinley, "Debugging probabilistic programs," in *Proceedings of the 1st ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, 2017.

[158] B. Daniel, D. Dig, K. Garcia, and D. Marinov, "Automated testing of refactoring engines," in *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, 2007.

[159] V. Jagannath, Y. Y. Lee, B. Daniel, and D. Marinov, "Reducing the costs of bounded-exhaustive testing," in *Fundamental Approaches to Software Engineering*, 2009.

[160] M. Gligoric, T. Gvero, V. Jagannath, S. Khurshid, V. Kuncak, and D. Marinov, "Test generation through programming in udita," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, 2010.

[161] C. Holler, K. Herzig, and A. Zeller, "Fuzzing with code fragments," in *USENIX Security*, 2012.

[162] P. Godefroid, A. Kiezun, and M. Y. Levin, "Grammar-based whitebox fuzzing," in *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2008.

[163] R. Majumdar and R.-G. Xu, "Directed test generation using symbolic grammars," in *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, 2007.

[164] E. G. Sirer and B. N. Bershad, "Using production grammars in software testing," in *Proceedings of the 2Nd Conference on Domain-specific Languages*, 1999.

[165] P. M. Maurer, "Generating test data with enhanced context-free grammars," *IEEE Software*, 1990.

[166] P. Godefroid, M. Y. Levin, D. A. Molnar et al., "Automated whitebox fuzz testing." in *NDSS*, 2008.

[167] M. Sutton, A. Greene, and P. Amini, *Fuzzing: Brute Force Vulnerability Discovery*, 2007.

[168] P. Purdom, "A sentence generator for testing parsers," *BIT Numerical Mathematics*, 1972.

[169] B. P. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of unix utilities," *CACM*, 1990.

[170] G. Shu, Y. Hsu, and D. Lee, "Detecting communication protocol security flaws by formal fuzz testing and machine learning," in *Formal Techniques for Networked and Distributed Systems – FORTE 2008*, 2008.

[171] P. Oehlert, "Violating assumptions with fuzzing," *IEEE Security and Privacy*, 2005.

[172] W. M. McKeeman, "Differential testing for software," *DEC Digital Technical Journal*, 1998.

[173] A. Groce, G. Holzmann, and R. Joshi, "Randomized differential testing as a prelude to formal verification," in *29th International Conference on Software Engineering*, 2007.

[174] T. Xie, K. Taneja, S. Kale, and D. Marinov, "Towards a framework for differential unit testing of object-oriented programs," in *Proceedings of the Second International Workshop on Automation of Software Test*, 2007.

[175] S. Srisakaokul, Z. Wu, A. Astorga, O. Alebiosu, and T. Xie, "Multiple-implementation testing of supervised learning software," in *AAAI-18 Workshop on Engineering Dependable and Secure Machine Learning Systems*, 2018.

[176] S. R. Choudhary, H. Versee, and A. Orso, "Webdiff: Automated identification of cross-browser issues in web applications," in *IEEE International Conference on Software Maintenance*, 2010.

[177] N. Li, J. Hwang, and T. Xie, "Multiple-implementation testing for xacml implementations," in *Proceedings of the 2008 Workshop on Testing, Analysis, and Verification of Web Services and Applications*, 2008.

[178] K. Taneja, N. Li, M. R. Marri, T. Xie, and N. Tillmann, "MiTV: Multiple-implementation testing of user-input validators for web applications," in *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, 2010.

[179] C. Murphy and G. E. Kaiser, "Improving the dependability of machine learning applications," CS Department, Columbia University, Tech. Rep., 2010.

[180] T. Y. Chen, S. C. Cheung, and S. M. Yiu, "Metamorphic testing: a new approach for generating next test cases," CS Department, Hong Kong University of Science and Technology, Tech. Rep., 1998.

[181] Z. Q. Zhou, D. Huang, T. Tse, Z. Yang, H. Huang, and T. Chen, "Metamorphic testing and its applications," in *Proceedings of the 8th International Symposium on Future Software Technology*, 2004.

[182] T. Y. Chen, J. Feng, and T. Tse, "Metamorphic testing of programs on partial differential equations: a case study," in *Computer Software and Applications Conference, 2002. COMPSAC 2002. Proceedings. 26th Annual International*, 2002.

[183] A. Gotlieb and B. Botella, "Automated metamorphic testing," in *Computer Software and Applications Conference, 2003. COMPSAC 2003. Proceedings. 27th Annual International*, 2003.

[184] C. Murphy, G. E. Kaiser, L. Hu, and L. Wu, "Properties of machine learning applications for use in metamorphic testing," in *International Conference on Software Engineering and Knowledge Engineering*, 2008.

[185] H. Liu, F.-C. Kuo, D. Towey, and T. Y. Chen, "How effectively does metamorphic testing alleviate the oracle problem?" *IEEE Transactions on Software Engineering*, 2014.

[186] X. Xie, J. W. Ho, C. Murphy, G. Kaiser, B. Xu, and T. Y. Chen, "Testing and validating machine learning classifiers by metamorphic testing," *Journal of Systems and Software*, 2011.

[187] S. Segura, G. Fraser, A. B. Sanchez, and A. Ruiz-Cortés, "A survey on metamorphic testing," *IEEE Transactions on software engineering*, 2016.

[188] A. Pfeffer, "Ibal: a probabilistic rational programming language," in *Proceedings of the 17th international joint conference on Artificial intelligence-Volume 1*, 2001.

[189] A. V. Kozlov and D. Koller, "Nonuniform dynamic discretization in hybrid networks," in *Proceedings of the Thirteenth conference on Uncertainty in artificial intelligence*, 1997.

[190] N. Goodman and A. Stuhlmüller, "The Design and Implementation of Probabilistic Programming Languages," 2014, http://dippl.org.

[191] E. Bingham, J. P. Chen, M. Jankowiak, F. Obermeyer, N. Pradhan, T. Karaletsos, R. Singh, P. Szerlip, P. Horsfall, and N. D. Goodman, "Pyro: Deep Universal Probabilistic Programming," *Journal of Machine Learning Research*, 2018.

[192] J. Ai, N. S. Arora, T. Jiang, M. Tingley et al., "Hackppl: a universal probabilistic programming language," in *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, 2019.

[193] T. Minka, J. Winn, J. Guiver, S. Webster, Y. Zaykov, B. Yangel, A. Spengler, and J. Bronskill, "Infer.NET 2.5," 2013, microsoft Research Cambridge. http://research.microsoft.com/infernet.

[194] M. Hoffman and A. Gelman, "The no-u-turn sampler: adaptively setting path lengths in hamiltonian monte carlo." *Journal of Machine Learning Research*, 2014.

[195] R. Neal et al., "Mcmc using hamiltonian dynamics," *Handbook of Markov Chain Monte Carlo*, 2011.

[196] S. Dutta, O. Legunsen, Z. Huang, and S. Misailovic, "Testing probabilistic programming systems," in *Proceedings of the 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2018.

[197] J. Katoen, "Probabilistic programming: A true verification challenge," in *Automated Technology for Verification and Analysis: 13th International Symposium*, 2015.

[198] "Stan issue reporting instruction," 2018, http://mc-stan.org/users/issues/index.html.

[199] J. Regehr, Y. Chen, P. Cuoq, E. Eide, C. Ellison, and X. Yang, "Test-case reduction for c compiler bugs," in *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*, 2012.

[200] A. Zeller and R. Hildebrandt, "Simplifying and isolating failure-inducing input," *IEEE Transactions on software engineering*, 2002.

[201] G. Misherghi and Z. Su, "Hdd: hierarchical delta debugging," in *Proceedings of the 28th international conference on Software engineering*, 2006.

[202] C. Sun, Y. Li, Q. Zhang, T. Gu, and Z. Su, "Perses: Syntax-guided program reduction," in *Proceedings of the 28th international conference on Software engineering*, 2018.

[203] "Stan example models," 2018, https://github.com/stan-dev/example-models.

[204] "Stan Issue #1610," 2015, https://github.com/stan-dev/stan/issues/1610.

[205] D. Blei, A. Ng, and M. Jordan, "Latent dirichlet allocation," *Journal of Machine Learning Research*, 2003.

[206] T. Parr, *The definitive ANTLR 4 reference*, 2013.

[207] D. P. Kingma and M. Welling, "Auto-encoding variational bayes," in *International Conference on Learning Representations*, 2014.

[208] J. Nocedal and S. Wright, "Numerical optimization, series in operations research and financial engineering," *Springer*, 2006.

[209] M. Alipour, A. Shi, R. Gopinath, D. Marinov, and A. Groce, "Evaluating non-adequate test-case reduction," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, 2016.

[210] S. Dutta, Z. Huang, and S. Misailovic, "Sixthsense: Debugging convergence problems in probabilistic programs via program representation learning," in *International Conference on Fundamental Approaches to Software Engineering*, 2022.

[211] S. Dutta, D. Garbervetsky, S. K. Lahiri, and M. Schäfer, "Inspectjs: Leveraging code similarity and user-feedback for effective taint specification inference for javascript," in *IEEE/ACM 44th International Conference on Software Engineering: Software Engineering in Practice*, 2022.

[212] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "Code2vec: Learning distributed representations of code," *Proceedings of the ACM on Programming Languages*, 2019.

[213] U. Alon, S. Brody, O. Levy, and E. Yahav, "Code2seq: Generating sequences from structured representations of code," in *International Conference on Learning Representations*, 2018.

[214] Z. Huang, S. Dutta, and S. Misailovic, "Aqua: Automated quantized inference for probabilistic programs," in *International Symposium on Automated Technology for Verification and Analysis (ATVA)*, 2021.

[215] Z. Huang, S. Dutta, and S. Misailovic, "Automated quantized inference for probabilistic programs with aqua," *Innovations in Systems and Software Engineering*, 2022.

[216] Z. Zhou, Z. Huang, and S. Misailovic, "Aquasense: Automated sensitivity analysis of probabilistic programs via quantized inference," in *International Symposium on Automated Technology for Verification and Analysis (ATVA)*, 2023.

[217] Z. Huang, S. Dutta, and S. Misailovic, "Astra: Understanding the practical impact of robustness for probabilistic programs," in *39th Conference on Uncertainty in Artificial Intelligence*, 2023.

[218] S. Zhang, "Practical semantic test simplification," in *35th International Conference on Software Engineering*, 2013.

[219] S. Herfert, J. Patra, and M. Pradel, "Automatically reducing tree-structured test inputs," in *32nd IEEE/ACM International Conference on Automated Software Engineering*, 2017.

[220] B. Li, M. Grechanik, and D. Poshyvanyk, "Sanitizing and minimizing databases for software application test outsourcing," in *IEEE Seventh International Conference on Software Testing, Verification and Validation*, 2014.

[221] A. Vahabzadeh, A. Stocco, and A. Mesbah, "Fine-grained test minimization," in *Proceedings of the 40th International Conference on Software Engineering*, 2018.

[222] V. Ngo and A. Legay, "Formal verification of probabilistic systemc models with statistical model checking," *Journal of Software: Evolution and Process*, 2018.

[223] Z. Huang, Z. Wang, and S. Misailovic, "Psense: Automatic sensitivity analysis for probabilistic programs," in *Automated Technology for Verification and Analysis: 16th International Symposium*, 2018.

[224] M. Weiser, "Program slicing," in *IEEE Transactions on Software Engineering*, 1981.

[225] C.-K. Hur, A. V. Nori, S. K. Rajamani, and S. Samuel, "Slicing probabilistic programs," in *ACM SIGPLAN Notices*, 2014.

[226] T. Amtoft and A. Banerjee, "A theory of slicing for probabilistic control flow graphs," in *Foundations of Software Science and Computation Structures: 19th International Conference*, 2016.

[227] T. Mens and T. Tourwé, "A survey of software refactoring," *IEEE Transactions on software engineering*, 2004.

[228] M. Fowler, *Refactoring: improving the design of existing code*, 2018.

[229] S. Dutta, J. Selvam, A. Jain, and S. Misailovic, "Tera: Optimizing stochastic regression tests in machine learning projects," in *30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2021.

[230] S. Dutta, A. Shi, and S. Misailovic, "Flex: Fixing flaky tests in machine learning projects by updating assertion bounds," in *29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021.

[231] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga et al., "PyTorch: An imperative style, high-performance deep learning library," in *Advances in neural information processing systems*, 2019.

[232] "Numpyro," 2020, https://github.com/pyro-ppl/numpyro.

[233] J. V. Dillon, I. Langmore, D. Tran, E. Brevdo, S. Vasudevan, D. Moore, B. Patton, A. Alemi, M. Hoffman, and R. A. Saurous, "Tensorflow distributions," *arXiv preprint arXiv:1711.10604*, 2017.

[234] "Pymc3," 2020, https://github.com/pymc-devs/pymc3.

[235] "Conda package management system," 2017, https://docs.conda.io.

[236] C. Roberts, A. Milsted, M. Ganahl, A. Zalcman, B. Fontaine, Y. Zou, J. Hidary, G. Vidal, and S. Leichenauer, "Tensornetwork: A library for physics and machine learning," 2019.

[237] "Tensornetwork," 2021, https://github.com/google/TensorNetwork.

[238] "Tensornetwork issue 943," 2021, https://github.com/google/TensorNetwork/issues/943.

[239] "Tensorflow ranking example," 2021, https://github.com/tensorflow/ranking/blob/019a7db68d83959b8774bc77bfb6905180504216/tensorflow_ranking/python/utils_test.py#L157.

[240] "Pytorch/serve example," 2021, https://github.com/pytorch/serve/blob/ccaba6f66a1f4a00a594b1371e42d5749be4e35d/test/benchmark/tests/conftest.py#L120.

[241] "Flaky test plugin," 2019, https://github.com/box/flaky.

[242] S. Thorve, C. Sreshtha, and N. Meng, "An empirical study of flaky tests in android apps," in *IEEE International Conference on Software Maintenance and Evolution*, 2018.

[243] P. E. Strandberg, T. J. Ostrand, E. J. Weyuker, W. Afzal, and D. Sundmark, "Intermittently failing tests in the embedded systems domain," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2020.

[244] W. Lam, R. Oei, A. Shi, D. Marinov, and T. Xie, "iDFlakies: A framework for detecting and partially classifying flaky tests," in *12th ieee conference on software testing, validation and verification*, 2019.

[245] A. Gambi, J. Bell, and A. Zeller, "Practical test dependency detection," in *IEEE 11th International Conference on Software Testing, Verification and Validation*, 2018.

[246] Z. Dong, A. Tiwari, X. L. Yu, and A. Roychoudhury, "Flaky test detection in android via event order exploration," 2021.

[247] A. Shi, A. Gyori, O. Legunsen, and D. Marinov, "Detecting assumptions on deterministic implementations of non-deterministic specifications," in *IEEE International Conference on Software Testing, Verification and Validation*, 2016.

[248] W. Lam, P. Godefroid, S. Nath, A. Santhiar, and S. Thummalapenta, "Root causing flaky tests in a large-scale industrial setting," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2019.

[249] W. Lam, K. Muşlu, H. Sajnani, and S. Thummalapenta, "A study on the lifecycle of flaky tests," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020.

[250] Y. Zhang, L. Ren, L. Chen, Y. Xiong, S.-C. Cheung, and T. Xie, "Detecting numerical bugs in neural network architectures," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020.

[251] Q. Guo, X. Xie, Y. Li, X. Zhang, Y. Liu, X. Li, and C. Shen, "Audee: Automated testing for deep learning frameworks," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, 2020.

[252] Q. Hu, L. Ma, X. Xie, B. Yu, Y. Liu, and J. Zhao, "Deepmutation++: A mutation testing framework for deep learning systems," in *34th IEEE/ACM International Conference on Automated Software Engineering*, 2019.

[253] A. Wei, Y. Deng, C. Yang, and L. Zhang, "Free lunch for testing: Fuzzing deep-learning libraries from open source," in *Proceedings of the 44th International Conference on Software Engineering*, 2022.

[254] S. Dutta, W. Zhang, Z. Huang, and S. Misailovic, "Storm: Program reduction for testing and debugging probabilistic programming systems," in *ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019.

[255] Y. R. S. Llerena, M. Böhme, M. Brünink, G. Su, and D. S. Rosenblum, "Verifying the long-run behavior of probabilistic system models in the presence of uncertainty," in *26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2018.

[256] M. Nejadgholi and J. Yang, "A study of oracle approximations in testing deep learning libraries," in *34th IEEE/ACM International Conference on Automated Software Engineering*, 2019.

[257] B. Efron and R. J. Tibshirani, *An introduction to the bootstrap*, 1994.

[258] E. Bingham, J. P. Chen, M. Jankowiak, F. Obermeyer, N. Pradhan, T. Karaletsos, R. Singh, P. Szerlip, P. Horsfall, and N. D. Goodman, "Pyro: Deep universal probabilistic programming," *The Journal of Machine Learning Research*, 2019.

[259] J. Salvatier, T. V. Wiecki, and C. Fonnesbeck, "Probabilistic programming in python using pymc3," *PeerJ Computer Science*, 2016.

[260] "Pypesto project," 2020, https://github.com/ICB-DCM/pyPESTO.

[261] D. Öztuna, A. H. Elhan, and E. Tüccar, "Investigation of four different normality tests in terms of type 1 error rate and power under different distributions," *Turkish Journal of Medical Sciences*, 2006.

[262] V. Choulakian and M. A. Stephens, "Goodness-of-fit tests for the generalized pareto distribution," *Technometrics*, 2001.

[263] B. Bader, J. Yan, X. Zhang et al., "Automated threshold selection for extreme value analysis via ordered goodness-of-fit tests with adjustment for false discovery rate," *The Annals of Applied Statistics*, 2018.

[264] "Pypesto pull request:570," 2021, https://github.com/ICB-DCM/pyPESTO/pull/570.

[265] A. A. Balkema and L. De Haan, "Limit distributions for order statistics. i," *Theory of Probability & Its Applications*, 1978.

[266] I. J. Myung, "Tutorial on maximum likelihood estimation," *Journal of mathematical Psychology*, 2003.

[267] G. E. Box and D. R. Cox, "An analysis of transformations," *Journal of the Royal Statistical Society: Series B (Methodological)*, 1964.

[268] J. L. Teugels and G. Vanroelen, "Box-cox transformations and heavy-tailed distributions," *Journal of Applied Probability*, 2004.

[269] D. R. Helsel and R. M. Hirsch, *Statistical methods in water resources*, 1992.

[270] F. B. Oppong and S. Y. Agbedra, "Assessing univariate and multivariate normality. a guide for non-statisticians," *Math. Theory Modeling*, 2016.

[271] C. Gourieroux, A. Holly, and A. Monfort, "Likelihood ratio test, wald test, and kuhn-tucker test in linear models with inequality constraints on the regression parameters," *Econometrica: journal of the Econometric Society*, 1982.

[272] "Pytest," 2020, https://docs.pytest.org/en/stable.

[273] "Eva package in r," 2021, https://rdrr.io/cran/eva/man/eva.html.

[274] "Anaconda python," 2020, https://anaconda.org/.

[275] "Bazel," 2021, https://bazel.build/.

[276] "Coax pull request:13," 2020, https://github.com/microsoft/coax/pull/13.

[277] "Deepchem pull request:2408," 2020, https://github.com/deepchem/deepchem/pull/2408.

[278] "Fastnlp pull request:352," 2020, https://github.com/fastnlp/fastNLP/pull/352.

[279] "Garage pull request:2242," 2020, https://github.com/rlworkgroup/garage/pull/2242.

[280] "Gensim pull request:3050," 2021, https://github.com/RaRe-Technologies/gensim/pull/3050.

[281] "Hummingbird pull request:450," 2021, https://github.com/microsoft/hummingbird/pull/450.

[282] "Hummingbird pull request:451," 2021, https://github.com/microsoft/hummingbird/pull/451.

[283] "Hummingbird pull request:449," 2021, https://github.com/microsoft/hummingbird/pull/449.

[284] "Magnitude pull request:84," 2020, https://github.com/plasticityai/magnitude/pull/84.

[285] "Nlp-architect pull request:207," 2021, https://github.com/IntelLabs/nlp-architect/pull/207.

[286] "Parlai pull request:3467," 2021, https://github.com/facebookresearch/ParlAI/pull/3467.

[287] "Pgmpy pull request:1380," 2020, https://github.com/pgmpy/pgmpy/pull/1380.

[288] "Pytorch-meta pull request:117," 2021, https://github.com/tristandeleu/pytorch-meta/pull/117.

[289] "Refnx pull request:540," 2020, https://github.com/refnx/refnx/pull/540.

[290] "Stellargraph pull request:1880," 2020, https://github.com/stellargraph/stellargraph/pull/1880.

[291] "Umap pull request:600," 2020, https://github.com/lmcinnes/umap/pull/600.

[292] "Zfit pull request:288," 2021, https://github.com/zfit/zfit/pull/288.

[293] "Zfit pull request:290," 2021, https://github.com/zfit/zfit/pull/290.

[294] "Gensim pull request:3059," 2020, https://github.com/RaRe-Technologies/gensim/pull/3059.

[295] P. L. Chebyshev, "Des valeurs moyennes," *J. Math. Pures Appl*, 1867.

[296] A. DasGupta, "Best constants in chebyshev inequalities with various applications," *Metrika*, 2000.

[297] T. N. Tolhurst, "Model-free tests and evidence of bubbles in real markets," Ph.D. dissertation, University of California, Davis, 2020.

[298] F. P. Cantelli, *Intorno ad un teorema fondamentale della teoria del rischio.* Tip. degli operai, 1910.

[299] V. V. Buldygin and Y. V. Kozachenko, "Sub-gaussian random variables," *Ukrainian Mathematical Journal*, 1980.

[300] F. J. Anscombe and W. J. Glynn, "Distribution of the kurtosis statistic b 2 for normal samples," *Biometrika*, 1983.

[301] S. S. Shapiro and M. B. Wilk, "An analysis of variance test for normality (complete samples)," *Biometrika*, 1965.

[302] F. J. Massey Jr, "The kolmogorov-smirnov test for goodness of fit," *Journal of the American statistical Association*, 1951.

[303] V. W. Berger and Y. Zhou, "Kolmogorov–smirnov test: Overview," *Wiley statsref: Statistics reference online*, 2014.

[304] B. Daniel, V. Jagannath, D. Dig, and D. Marinov, "ReAssert: Suggesting repairs for broken unit tests," in *IEEE/ACM International Conference on Automated Software Engineering*, 2009.

[305] B. Daniel, T. Gvero, and D. Marinov, "On test repair using symbolic execution," in *Proceedings of the 19th international symposium on Software testing and analysis*, 2010.

[306] X. Li, M. d'Amorim, and A. Orso, "Intent-preserving test repair," in *ICST*, 2019.

[307] M. Mirzaaghaei, F. Pastore, and M. Pezzè, "Supporting test suite evolution through test case adaptation," in *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, 2012.

[308] G. Yang, S. Khurshid, and M. Kim, "Specification-based test repair using a lightweight formal method," in *Formal Methods: 18th International Symposium*, 2012.

[309] A. Dwarakanath, M. Ahuja, S. Sikand, R. M. Rao, R. J. C. Bose, N. Dubash, and S. Podder, "Identifying implementation bugs in machine learning based image classifiers using metamorphic testing," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2018.

[310] K. Sen, M. Viswanathan, and G. Agha, "On statistical model checking of stochastic systems," in *International Conference on Computer Aided Verification*, 2005.

[311] A. Arcuri and L. Briand, "A practical guide for using statistical tests to assess randomized algorithms in software engineering," in *International Conference on Software Engineering*, 2011.

[312] A. Arcuri and L. Briand, "A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering," *Software Testing, Verification and Reliability*, 2014.

[313] C. Mandrioli and M. Maggio, "Testing self-adaptive software with probabilistic guarantees on performance metrics," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020.

[314] B. Nongpoh, R. Ray, S. Dutta, and A. Banerjee, "Autosense: A framework for automated sensitivity analysis of program data," *IEEE Transactions on Software Engineering*, 2017.

[315] P. Wang, H. Fu, K. Chatterjee, Y. Deng, and M. Xu, "Proving expected sensitivity of probabilistic programs with randomized variable-dependent termination time," *Proceedings of the ACM on Programming Languages*, 2019.

[316] T.-W. Weng, H. Zhang, P.-Y. Chen, J. Yi, D. Su, Y. Gao, C.-J. Hsieh, and L. Daniel, "Evaluating the robustness of neural networks: An extreme value theory approach," in *International Conference on Learning Representations*, 2018.

[317] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*, 2018.

[318] P. Congdon, *Applied bayesian modelling*, 2014.

[319] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," *arXiv preprint arXiv:1409.3215*, 2014.

[320] "Travis-ci," 2021, https://travis-ci.org.

[321] "Circleci," 2021, https://circleci.com.

[322] J. Schneider and S. Kirkpatrick, *Stochastic optimization*, 2007.

[323] M. Pelikan, D. E. Goldberg, E. Cantú-Paz et al., "Boa: The bayesian optimization algorithm," in *Proceedings of the genetic and evolutionary computation conference GECCO-99*, 1999.

[324] J. Mockus, *Bayesian approach to global optimization: theory and applications*, 2012.

[325] J. Quiñonero-Candela and C. E. Rasmussen, "A unifying view of sparse approximate gaussian process regression," *Journal of Machine Learning Research*, 2005.

[326] J. S. Bergstra, R. Bardenet, Y. Bengio, and B. Kégl, "Algorithms for hyper-parameter optimization," in *Advances in neural information processing systems*, 2011.

[327] J. Chen, N. Xu, P. Chen, and H. Zhang, "Efficient compiler autotuning via bayesian optimization," in *International Conference on Software Engineering*, 2021.

[328] V. Le, C. Sun, and Z. Su, "Finding deep compiler bugs via guided stochastic program mutation," *ACM SIGPLAN Notices*, 2015.

[329] V. Nair, Z. Yu, T. Menzies, N. Siegmund, and S. Apel, "Finding faster configurations using flash," *IEEE Transactions on Software Engineering*, 2018.

[330] K. Heo, H. Oh, H. Yang, and K. Yi, "Adaptive static analysis via learning with bayesian optimization," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 2018.

[331] H. Oh, H. Yang, and K. Yi, "Learning a strategy for adapting a program analysis via bayesian optimisation," *ACM SIGPLAN Notices*, 2015.

[332] A. Gelman, H. S. Stern, J. B. Carlin, D. B. Dunson, A. Vehtari, and D. B. Rubin, *Bayesian data analysis*, 2013.

[333] J. Geweke, *Evaluating the accuracy of sampling-based approaches to the calculation of posterior moments*, 1991.

[334] A. E. Raftery and S. M. Lewis, "The number of iterations, convergence diagnostics and generic metropolis algorithms," *Practical Markov Chain Monte Carlo*, 1995.

[335] J. A. Rice, *Mathematical statistics and data analysis*, 2006.

[336] "Python ast package," 2021, https://docs.python.org/3/library/ast.html.

[337] "Autokeras," 2021, https://github.com/keras-team/autokeras.

[338] "Bambi," 2021, https://github.com/bambinos/bambi.

[339] "Cleverhans," 2021, https://github.com/tensorflow/cleverhans.

[340] "Fairseq," 2021, https://github.com/pytorch/fairseq.

[341] "Gensim," 2021, https://github.com/RaRe-Technologies/gensim.

[342] "Gpytorch," 2021, https://github.com/cornellius-gp/gpytorch.

[343] "imbalanced-learn," 2021, https://github.com/scikit-learn-contrib/imbalanced-learn.

[344] "Parlai," 2021, https://github.com/facebookresearch/ParlAI.

[345] "Pygpgo," 2021, https://github.com/josejimenezluna/pyGPGO.

[346] "Pymc-learn," 2021, https://github.com/pymc-learn/pymc-learn.

[347] "Pyro," 2021, https://github.com/pyro-ppl/pyro.

[348] "Sbi," 2021, https://github.com/mackelab/sbi.

[349] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE transactions on software engineering*, 2010.

[350] M. Papadakis, M. Kintis, J. Zhang, Y. Jia, Y. Le Traon, and M. Harman, "Mutation testing advances: an analysis and survey," in *Advances in Computers*, 2019.

[351] "Github actions," 2020, https://github.com/features/actions.

[352] "Hyperopt: Hyperparameter optimization," 2021, https://github.com/hyperopt/hyperopt.

[353] J. Bergstra, D. Yamins, and D. Cox, "Making a science of model search: Hyperparameter optimization in hundreds of dimensions for vision architectures," in *International conference on machine learning*, 2013.

[354] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[355] R. Ward, X. Wu, and L. Bottou, "Adagrad stepsizes: sharp convergence over nonconvex landscapes," in *The Journal of Machine Learning Research*, 2020.

[356] "Pyro test for variational inference," 2021, https://github.com/pyro-ppl/pyro/blob/25368f56c984506a46e412a9017c0d8fa43fd0c6/tests/infer/test_inference.py#L288.

[357] "Pyro test using rbd kernel," 2021, https://github.com/pyro-ppl/pyro/blob/25368f56c984506a46e412a9017c0d8fa43fd0c6/tests/infer/test_inference.py#L292.

[358] "Mutmut: Python mutation tester," 2020, https://github.com/boxed/mutmut.

[359] C. A. Boneau, "The effects of violations of assumptions underlying the t test." *Psychological bulletin*, 1960.

[360] A. Shi, A. Gyori, S. Mahmood, P. Zhao, and D. Marinov, "Evaluating test-suite reduction in real software evolution," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2018.

[361] S. Tallam and N. Gupta, "A concept analysis inspired greedy algorithm for test suite minimization," *ACM SIGSOFT Software Engineering Notes*, 2005.

[362] J. A. Jones and M. J. Harrold, "Test-suite reduction and prioritization for modified condition/decision coverage," *IEEE Transactions of Software Engineering*, 2003.

[363] A. Groce, J. Holmes, and K. Kellar, "One test to rule them all," in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2017.

[364] X. He, K. Zhao, and X. Chu, "Automl: A survey of the state-of-the-art," *Knowledge-Based Systems*, 2021.

[365] Q. Yao, M. Wang, Y. Chen, W. Dai, H. Yi-Qi, L. Yu-Feng, T. Wei-Wei, Y. Qiang, and Y. Yang, "Taking human out of learning applications: A survey on automated machine learning," *arXiv preprint arXiv:1810.13306*, 2018.

[366] B. Zoph and Q. V. Le, "Neural architecture search with reinforcement learning," *arXiv preprint arXiv:1611.01578*, 2016.

[367] T. Elsken, J. H. Metzen, F. Hutter et al., "Neural architecture search: A survey." *Journal of Machine Learning Research*, 2019.

[368] J. Snoek, H. Larochelle, and R. P. Adams, "Practical bayesian optimization of machine learning algorithms," in *Advances in neural information processing systems*, 2012.

[369] D. Maclaurin, D. Duvenaud, and R. Adams, "Gradient-based hyperparameter optimization through reversible learning," in *International conference on machine learning*, 2015.

[370] F. Palomba and A. Zaidman, "Does refactoring of test smells induce fixing flaky tests?" in *IEEE international conference on software maintenance and evolution*, 2017.

[371] M. Harman and P. O'Hearn, "From start-ups to scale-ups: Opportunities and open problems for static and dynamic program analysis," in *18th International Working Conference on Source Code Analysis and Manipulation*, 2018.

[372] W. Lam, P. Godefroid, S. Nath, A. Santhiar, and S. Thummalapenta, "Root causing flaky tests in a large-scale industrial setting," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2019.

[373] A. Romano, Z. Song, S. Grandhi, W. Yang, and W. Wang, "An empirical analysis of ui-based flaky tests," in *International Conference on Software Engineering*, 2021.

[374] J. Bell, O. Legunsen, M. Hilton, L. Eloussi, T. Yung, and D. Marinov, "DeFlaker: Automatically detecting flaky tests," in *International Conference on Software Engineering*, 2018.

[375] A. Wei, P. Yi, T. Xie, D. Marinov, and W. Lam, "Probabilistic and systematic coverage of consecutive test-method pairs for detecting order-dependent flaky tests," in *Tools and Algorithms for the Construction and Analysis of Systems*, 2021.

[376] P. Zhang, Y. Jiang, A. Wei, V. Stodden, D. Marinov, and A. Shi, "Domain-specific fixes for flaky tests with wrong assumptions on underdetermined specifications," in *IEEE/ACM 43rd International Conference on Software Engineering*, 2021.

[377] L. Zheng, C. Jia, M. Sun, Z. Wu, C. H. Yu, A. Haj-Ali, Y. Wang, J. Yang, D. Zhuo, K. Sen et al., "Ansor: Generating high-performance tensor programs for deep learning," in *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*, 2020.

[378] G. Baudart et al., "Deep probabilistic programming languages: A qualitative study," *arXiv preprint arXiv:1804.06458*, 2018.