FLEX: Fixing Flaky Tests in Machine Learning Projects by Updating Assertion Bounds

Saikat Dutta  
UIUC  
Urbana, USA  
saikatdl2@illinois.edu

August Shi  
The University of Texas at Austin  
Austin, USA  
august@utexas.edu

Sasa Misailovic  
UIUC  
Urbana, USA  
misailo@illinois.edu

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 [31], Reinforcement Learning [43], or Probabilistic Programming [32, 33]. 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 [18, 19, 54] that compare the metric to an acceptability bound, e.g.,

\[ \text{assert (accuracy} > \alpha) \]

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 [19]. However, by keeping randomness throughout, the tests may become flaky [49] – 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 \( \alpha \) 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 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) 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). 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 \( \alpha \) 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 \( \alpha \) 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 \( \alpha \) 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. 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 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, TensorFlow, TensorFlow-Probability, Pyro, PyMC3, and NumPyro. The dependent projects provide domain specific functionalities and have a wide user base.

FLEX proposes a fix for 28 tests (Section 7.1). 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 paper 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/arc/flex.

### 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_nodes, 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.

Listing 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. It initializes a dataset for the test, which is sampled from a mixture of two Gaussian distributions. 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. The test compares the results of the sampler with the expected ground truth using the Kolmogorov-Smirnov (KS) test, 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).

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...
We leverage methods from Extreme Value Theory (EVT) to compute a bound with high statistical confidence (Section 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 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 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 [4, 10]). 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” [66]. 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)” [66].

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 $\theta$ explicit, we will write $F(x; \theta)$.

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 [64] 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 \(T\):

\[
F_T(y) = P(X - T \leq y \mid X > T) = \frac{F(T + y) - F(T)}{1 - F(T)} \quad (1)
\]

where \(0 \leq y \leq x_F - T\), where \(x_F\) is the rightmost endpoint
of \(F\) or infinity. Prior work [5, 64] 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

\[
G(y; T, \sigma, \xi) = \begin{cases} 
1 - \left(1 + \frac{y - T}{\sigma}\right)^{-1/\xi} & \text{if } \xi \neq 0 \\
1 - \exp(-y/T)^{1/\sigma} & \text{if } \xi = 0
\end{cases} \quad (2)
\]

Here, \(T\), \(\sigma\), and \(\xi\) correspond to location, scale, and shape,
respectively. These parameters can be estimated using Maximum Like-
lihood Estimation (MLE) methods [53]. The shape parameter, \(\xi\),
determines the nature of the tail: light, exponential, or heavy.

Figure 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 \(\sigma = 10\))

Figure 3 presents an example of how different kinds of tail dis-
tributions 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 exponen-
tial) 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
\(S = x_1, \ldots, x_n\), the location, 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 de-
defined as \(P(\theta | S) = \prod_{i=1}^{n} P(\theta | x_i)\) · · · \(\prod_{i=1}^{n} P(\theta | x_n) = \prod_{i=1}^{n} P(\theta | x_i)\),
where \(\theta\) is the set of parameters of GPD distribution. MLE then
obtains the parameter estimates that maximize this likelihood:

\[
\arg\max_{\theta} \prod_{i=1}^{n} P(\theta | x_i) \quad (3)
\]

where \(\lambda\) 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 [77] showed that applying Box-Cox transformation can be useful in presence of heavy tails and can lead to faster convergence. Further, Helsel and Hirsch [36] showed that quantiles (or percentiles) are invariant to monotonic transformations. Hence, $Q_r(f(Y)) = f(Q_r(Y))$, where $Q$ is the quantile function, $r \in (0, 1)$ 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 [58]. However, it is a useful heuristic that can help speed up or even enable finding convergence for a distribution.

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 $T$ of the form assert $X < a$, 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 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 $P(X \leq B) \geq 0.99$.

4.1 FLEX Algorithm

Algorithm 1 describes the main FLEX algorithm. It takes a test $T$, an assertion $A$ in the test, and a confidence level $C$ as input and returns the fixed version(s) of the test $T^\ast$ as output. Intuitively, the algorithm executes $T^\ast$ 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 4.2) takes the samples $\text{Samples}$, assertion $A$, a flag $F$ to enable/disable the Box-Cox transformation (Section 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) and returns the patched test (Line 19).

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 [4] to compare different threshold choices (discussed in Section 3, Goodness-of-fit) and choose the lowest threshold that passes the GPD test [10], meaning it fits adequately to a GPD distribution.

Algorithm 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 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., $p > \text{SIGNIFICANCE\_LEVEL}$), and we obtain a light or exponential distribution (Line 20), then we estimate the bound $B$ by computing the extreme percentile ($Q_\xi$) 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., $0 < \xi < 5$, for some small $\xi$), we can still approximate it using an exponential distribution in some cases. We use the Likelihood Ratio
We describe details on how we implement the main components. It takes as input a test and the number of times to run N. First, Test Runner instrument test T to log the actual and expected values used in the assertion A. For instance, for an assertion of the form \texttt{assert alllclose(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 \texttt{pytest}, a popular library for executing tests in Python projects.

### 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 T, an assertion A within T, and the number of times to run N. First, Test Runner instruments test T to log the actual and expected values used in the assertion A. For instance, for an assertion of the form \texttt{assert alllclose(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 \texttt{pytest}, a popular library for executing tests in Python projects.

Tail Bound Estimator. It implements the algorithm described in Section 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 [23] 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 (\( \gamma \)) of 0.05.

Patch. The Patch module takes a test T, assertion A in the test, all collected samples \texttt{Samples}, fitted GPD D, and the proposed bound B as input and provides one or more fixed version(s) of the test as output. If B is not infinity, it updates the assertion in the test accordingly (Section 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 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 5.1-5.3). FLEX may also choose not to fix a test (Section 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 6.3).

#### 5.1 Using the Statistical Tail Bound (SB)

If we obtain a light or exponential tailed distribution using Algorithm 2, then the distribution has a finite bound. We then simply compute the extreme percentiles (e.g., \( Q_{0.99} \) or \( Q_{0.999} \)), 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 \( 1 - C \).

#### 5.2 Using the Empirical Bound (EB)

If the tail bound estimation algorithm (Algorithm 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 [22] 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., \( Q_{1}(E) \), mean, or median) as the new empirical bound. FLEX also computes the 95% confidence interval (\( [Q_{0.025}(E) - Q_{0.975}(E)] \)) which denotes the variability in the empirical bound – a smaller confidence interval indicates the empirical bound is close to the true bound.

#### 5.3 Re-Running the Test (RR)

The Flaky [25] plugin for pytest allows the developers to automatically re-run the test on failure. To use this plugin, a developer needs
We describe how FLEX updates an assertion when a statistical or assertions comparing absolute values. This category includes whether the relative or absolute difference between two floating-point values is less than a threshold. Some assertions check assert_almost_equal (e.g., (GPD) returned by Algorithm 2 and \(\alpha\) max_runs 1). The plugin runs the test up to \(\alpha\) max_passes times until it passes \(\alpha\) 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 1. FLEX computes the number of re-runs in the following two ways: 1) FLEX computes the empirical failure probability of the test: \(p = \frac{\#\text{failures}}{\#\text{runs}}\). Then it computes the number of re-runs using: \(n = \lceil \log(1 - C)/\log(1 - p) \rceil\), where C is the developer provided confidence level (as in Algorithm 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 2 and \(\alpha\) 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 \(\sum_{k=1}^{\infty} p^{k-1} \cdot (1 - p) \cdot k \cdot W\).

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 tighter than the original bound, indicating that the assertion bounds are not propose the fix to the developers.

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 \(|x| < |y|\) or \(|x| \leq |y|\), \(\alpha\), and some other APIs in unittest (e.g., assertGreater(x, y), assertLess(x, y), and assert_array_less(x, y)). To fix an assertion, FLEX simply replaces \(\alpha\) with the bound it computes. Listing 3 shows an example of such a fix from the ICB-DCM/pyPESTO project.

```
#assert statistic < 0.1
#assert statistic < 0.3
```

Listing 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 = C1, atol = C2), where C, C1, and C2 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 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-5, atol=1e-4)
```

Listing 4: Fix for test in microsoft/hummingbird

6 METHODOLOGY

6.1 Projects and Flaky Tests

We follow a similar methodology as Dutta et al. [19] to select machine learning projects for our evaluation. We start with two popular machine learning libraries (PyTorch [61] and TensorFlow [76]) and four probabilistic programming systems (Pyro [7], NumPyro [57], TensorFlow-Probability [16], and PyMC3 [71]) on GitHub. We use GitHub’s feature to track the projects dependent on these libraries and also have more than 10 stars, as an indication of popularity.

![Table 1: Details of projects used](image)

<table>
<thead>
<tr>
<th>Project</th>
<th>Dependent</th>
<th>Filtered</th>
</tr>
</thead>
<tbody>
<tr>
<td>TensorFlow</td>
<td>836</td>
<td>100</td>
</tr>
<tr>
<td>PyTorch</td>
<td>906</td>
<td>100</td>
</tr>
<tr>
<td>TensorFlow-Prob</td>
<td>283</td>
<td>100</td>
</tr>
<tr>
<td>NumPyro</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>Pyro</td>
<td>13</td>
<td>13</td>
</tr>
<tr>
<td>PyMC3</td>
<td>31</td>
<td>31</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>2072</strong></td>
<td><strong>347</strong></td>
</tr>
<tr>
<td><strong>Unique</strong></td>
<td><strong>1836</strong></td>
<td><strong>305</strong></td>
</tr>
<tr>
<td><strong>Successful at Testing</strong></td>
<td>-</td>
<td><strong>144</strong></td>
</tr>
<tr>
<td><strong>Projects with Flaky Tests</strong></td>
<td>-</td>
<td><strong>21</strong></td>
</tr>
</tbody>
</table>

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 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 [1], 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), or required specialized build/testing systems (such as Bazel [6]). 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

---

1We 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.
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).

6.2 FLEX Configuration

For our evaluation, we configure FLEX to first collect an initial 100 samples (INITIALSAMPLES from Algorithm 1). If more samples are needed, we configure FLEX to collect more samples in batches of 50 (NEXTSAMPLES in Algorithm 1). We specify FLEX to collect at most 3000 samples before stopping (MAXSAMPLES in Algorithm 1).

We set the minimum number of tail samples when testing for convergence to be 50 (MINTAILSAMPLES in Algorithm 2). We use SIGNIFICANCELEVEL of 0.05 for the GPD tests. For the confidence level (C in Algorithm 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.

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) 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.

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?

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 6.1. Table 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 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), but adjusts the recommendations based on the context of the test (Section 6.3). NF means that flaky test was not fixed, because FLEX’s proposed new assertion bound is tighter than the original (Section 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 7.2.

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.

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 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 $\text{count} \leq \text{length}$ that are lower bounded by zero (e.g., $\text{assert (np.count_nonzero(scores) > 0)}$ 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 $\geq 0$ 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:

```python
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 $3.27 \pm 0.96$. 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.

### 7.3 Developer Response to FLEX's Fixes

Using our methodology for sending pull requests to developers (Section 6.3), we ultimately sent 19 pull requests for tests for which FLEX proposes a fix. Table 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 pycm-learn/pycm-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.
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

Without providing any reason. For
we provide the estimates for the other percentiles (Section 6.3).
we use a different percentile after discussion with developers, and

Listing 5: Fix for test in zfit/zfit

Of the 6 rejected pull requests, the developers accepted different

test bounds. For two of our pull requests toMicrosoft/hummingbird[40,41], the developers reasoned that our proposed

Bound selections[73]. However, these prior works focused on flaky tests in ML applications. They found

8 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[19]. 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[19].

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 ob-
serve 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 sig-
nificance 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.[18]. Like other prior work on repairing
tests[12,13,47,52,81], 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.

9 RELATED WORK

Flaky Tests. Luo et al.[49] 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[27,46], asynchronous waits[44], or unordered collec-
tions[73]. However, these prior works focused on flaky tests in
traditional software.

Dutta et al.[17] 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[17] 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.

Table 3: Pull Requests

<table>
<thead>
<tr>
<th>Project</th>
<th>Tests</th>
<th>PRs</th>
<th>A</th>
<th>P</th>
<th>R</th>
<th>U</th>
</tr>
</thead>
<tbody>
<tr>
<td>cox-dev/coax[11]</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>deepchem/deepchem[15]</td>
<td>4</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>fastnlp/fastNLP[24]</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>rtworgroup/garage[28]</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Rale-Technologies/gensim[29]</td>
<td>4</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>microsoft/hummingbird[30–41]</td>
<td>3</td>
<td>3</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>plasticityai/magnitude[30]</td>
<td>3</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>IntelLabs/nlp-architect[55]</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>facebookresearch/parlai[60]</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>pgmpy/pgmpy[62]</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>ICB-DCM/pyPSTO[66]</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>pymc-learn/pymc-team</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>tristandeus/python-met[69]</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>reftx/refitx[70]</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>stellargraph/stellargraph[75]</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Lmcinnes/unmap[78]</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>zfit/zfit[82,83]</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Σn 28 19 9 4 6 9

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 other percentiles (Section 6.3).

We also focus on flaky tests that use approximate assertions, found
to be a common type of flaky test from prior work[19]. 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[19].
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 ob-
serve 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 sig-
nificance 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.[18]. Like other prior work on repairing
tests[12,13,47,52,81], 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.

9 RELATED WORK

Flaky Tests. Luo et al.[49] 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[27,46], asynchronous waits[44], or unordered collec-
tions[73]. However, these prior works focused on flaky tests in
traditional software.

Dutta et al.[17] 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[17] 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 [12, 13, 47, 52, 81]. 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 [74] or due to unordered collections [84]. The goal of these techniques is to make flaky tests no longer fail due to their flakiness root cause. Lam et al. [45] 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 [18] 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 [14, 26, 64] to determine tail distributions of the computed values in approximate assertions. While we rely on the Peak Over Threshold (POT) [64] method to apply EVT, there are other popular methods as well. Block Maxima Method (BMM) [14] 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-Tippett theorem [26], 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.

**Conclusion.** 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.

**Acknowledgments.** We would like to thank the anonymous reviewers for their valuable feedback for improving the paper. This work was supported in part by NSF, Grants No. CCF-1846354, CCF-1956374, CCF-2008883, CCF-2028861, Microsoft Azure, and Facebook PhD Fellowship.

**References.**
