Detecting and Understanding Real-World Differential Performance Bugs in Machine Learning Libraries

Saeid Tizpaz-Niari
University of Colorado Boulder
Boulder, CO, USA
saeid.tizpazniari@colorado.edu

Pavol Černý
TU Wien
Vienna, Austria
pavol.cerny@tuwien.ac.at

Ashutosh Trivedi
University of Colorado Boulder
Boulder, CO, USA
ashutosh.trivedi@colorado.edu

ABSTRACT

Programming errors that degrade the performance of systems are widespread, yet there is very little tool support for finding and diagnosing these bugs. We present a method and a tool based on *differential performance analysis* — we find inputs for which the performance varies widely, despite having the same size. To ensure that the differences in the performance are robust (i.e. hold also for large inputs), we compare the performance of not only single inputs, but of classes of inputs, where each class has similar inputs parameterized by their size. Thus, each class is represented by a performance function from the input size to performance. Importantly, we also provide an explanation for why the performance differs in a form that can be readily used to fix a performance bug.

The two main phases in our method are discovery with fuzzing and explanation with decision tree classifiers, each of which is supported by clustering. First, we propose an evolutionary fuzzing algorithm to generate inputs that characterize different performance functions. For this fuzzing task, the unique challenge is that we not only need the input class with the worst performance, but rather a set of classes exhibiting differential performance. We use clustering to merge similar input classes which significantly improves the efficiency of our fuzzer. Second, we explain the differential performance in terms of program inputs and internals (e.g., methods and conditions). We adapt discriminant learning approaches with clustering and decision trees to localize suspicious code regions.

We applied our techniques on a set of micro-benchmarks and real-world machine learning libraries. On a set of micro-benchmarks, we show that our approach outperforms state-of-the-art fuzzers in finding inputs to characterize differential performance. On a set of case-studies, we discover and explain multiple performance bugs in popular machine learning frameworks, for instance in implementations of logistic regression in `scikit-learn`. Four of these bugs, reported first in this paper, have since been fixed by the developers.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**.

KEYWORDS

Differential Performance Bugs, ML Libraries, Testing, Debugging

ACM Reference Format:
Saeid Tizpaz-Niari, Pavol Černý, and Ashutosh Trivedi. 2020. Detecting and Understanding Real-World Differential Performance Bugs in Machine Learning Libraries. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '20), July 18–22, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3395363.3404540

1 INTRODUCTION

The defects in software developments can lead to severe performance degradations and waste valuable system resources such as CPU cycles. Moreover, studies have shown that such performance bugs are widespread in real-world applications [13, 36, 38]. How can a user of a program recognize a performance bug? Most often, they can suspect a bug if the program has different performance for similar inputs. Recently, such bugs have been reported by users of machine learning libraries [18, 19, 27, 37]. For example, a user of random forest regression reported that the trees with 'mae' criterion are slower than those with 'mse'. Once such an issue is discovered, how can a maintainer of a library know whether the difference is the result of a bug, or if it is inherent in the problem being solved?

We present a method and a tool to address the challenges we just described. We use *fuzzing* to generate inputs that uncover performance bugs and *discriminant analysis* to explain the differences in performance. The fuzzing part of our study generalizes evolutionary-based fuzzing algorithms [1] in two ways. *First, we consider multiple populations of inputs.* Each population corresponds to a *simple path* [1] in the program's control-flow graph. An execution that takes a loop 5 times is represented by the same simple path as an execution that takes the loop 7 times. Therefore, each class (or population) of inputs is represented by a function from the input size to performance. *Second, rather than searching for the single worst-case input class, our fuzzer explores classes of inputs with significant performance differences.* To do so, we propose an evolutionary algorithm such that it both models the performance as a function of input size and finds a set of classes of inputs with diverse performance efficiently. The key idea for the efficiency is to combine evolutionary fuzzing algorithms with the functional data *clustering* [6]. The clustering is used to focus the search to retain representative paths from a few prominent clusters of paths instead of repeatedly exploring paths with similar performance.

[1]A path is simple if it contains an edge at most once.

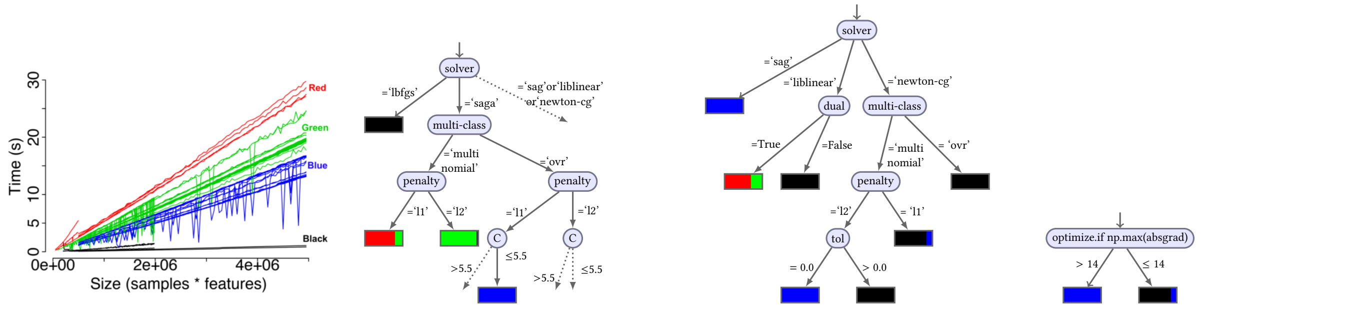

Figure 1: From left to right: (a) four classes of performances discovered by fuzzing and clustering, (b) decision tree learned in the space of parameter features (part 1), (c) decision tree learned in the space of parameter features (part 2), (d) decision tree learned using the internal features to explain differences in performance of 'newton-cg' solver.

Once a suspicious performance abnormality has been uncovered, the next step is to pinpoint the root causes for the differential performance. For this task, we adapt techniques from discriminant learning algorithms [2, 38] to functional data [17]. The causes of the diverse performance are explained using features from the space of program (hyper)parameters and from the space of program internals (such as the number of invocations of a particular method). We learn a discriminant model that shows what features are the same inside a cluster and what features distinguish one from another.

Previous works in performance fuzzing consider the worst-case algorithmic complexity [8, 16] and search for a single input with the significant resource usage. Differential fuzzing is used in security to detect timing side channels in Java applications [11, 40]. To the best of our knowledge, this is the first work to automatically generate inputs to characterize multiple performance classes. Also, this work utilizes the inputs from the fuzzer to automatically find code regions contributing to the differential performance, while previous works focusing on performance bug localization assume that the interesting inputs are given [36, 38].

We apply our approach on a set of micro-benchmark programs and larger machine learning libraries. On a set of micro-benchmarks that include well-known sorting, searching, tree, and graph algorithms [35], we demonstrate that although our approach is slower in generating inputs, it outperforms other fuzzing techniques [8, 16] in characterizing differential performance. On a set of eight larger machine learning tools and libraries, we find multiple previously-unreported performance bugs in widely used libraries such as in the implementation of logistic regression in scikit-learn framework [15]. We have reported these bugs, and four of them have since been fixed by the developers.

The key contributions of our paper are:

- We extend fuzzing algorithms to functional data and, crucially, use clustering *during* the fuzzing process to efficiently find classes of inputs with widely different performance. We show the importance of clustering during the fuzzing phase through comparison to state-of-the-art fuzzers.
- We use the discriminant learning approach alongside the fuzzing to find the root cause of performance issues.
- We implement our approach in the tool DPFuzz and evaluate it on eight machine learning libraries. We show the usefulness of DPFuzz in finding and explaining multiple performance bugs such as in scikit-learn libraries [15].

2 OVERVIEW

First, we show how DPFuzz can be used to detect a performance bug in a popular machine learning library. Then, we describe the components of DPFuzz.

A) Applying DPFuzz on Logistic Regression. Logistic regression in scikit-learn [15] is a popular classification model that supports various solvers and penalty functions. We refer the reader to [24] for more information about the functionality of this classifier. We analyzed the performance of logistic regression[2].

Task. We apply DPFuzz to automatically generate inputs that find diverse classes of performances. If there are multiple classes (more than one cluster), we explain the performance issues with DPFuzz.

Generating interesting inputs. We run the fuzzer for about two hours, and it generates 185 sets of inputs, where each input corresponds to a unique path in the control flow graph (CFG) of the program. Since each set includes multiple inputs, the performance of each path gives rise to functions. The fuzzer thus provides 185 different performance functions. The coverage of DPFuzz is 78% where it covers 2.1K LoC from almost 2.7K LoC.

Clustering performance functions. Given the performance functions, we use functional data clustering [6] algorithms to group them into a smaller number of clusters. Figure 1 (a) shows that the performance functions are clustered into 4 groups.

Explaining the clusters of performances. Given the set of inputs and their performance labels, we use the discriminant learning approach [39] with decision tree algorithms to explain the differences between performance clusters in terms of program hyper-parameters and program internal features.

The decision trees in Figure 1 (b) and (c) show the discriminant models in the space of hyper-parameters. One particular (expected) observation is that different solvers have different performance. However, more interesting parts are those with the differential performance in the same solver. For example, Figure 1 (c) shows the inputs with 'newton-cg' solver can be in black (fast) or blue (slow)

[2]https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html

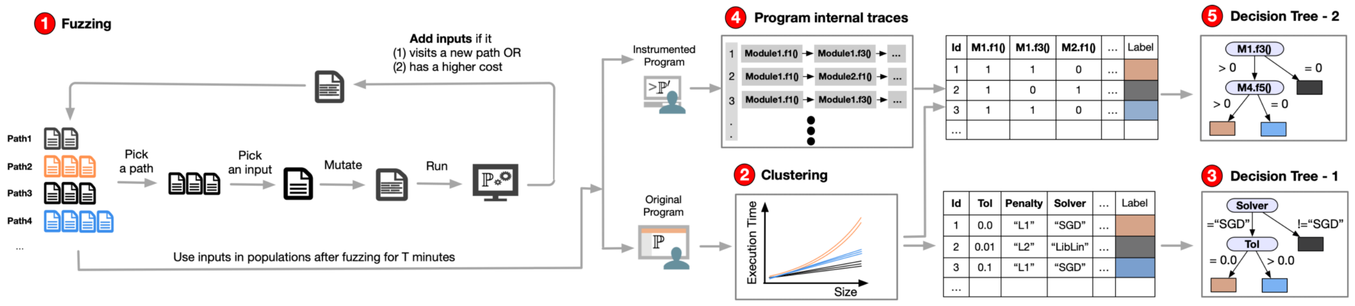
1 Fuzzing
Add inputs if it
(1) visits a new path OR
(2) has a higher cost
Path1
Path2
Path3
Path4
...
Pick a path
Pick an input
Mutate
Run
Use inputs in populations after fuzzing for T minutes
Instrumented Program
Original Program
4 Program internal traces
1 Module1.f1() → Module1.f3() → ...
2 Module1.f1() → Module2.f1() → ...
3 Module1.f1() → Module1.f3() → ...
2 Clustering
Execution Time
Size
Id | M1.f1() | M1.f3() | M2.f1() | ... | Label
1 | 1 | 1 | 0 | ...
2 | 1 | 0 | 1 | ...
3 | 1 | 1 | 0 | ...
...
Id | Tol | Penalty | Solver | ... | Label
1 | 0.0 | "L1" | "SGD" | ...
2 | 0.01 | "L2" | "LibLin" | ...
3 | 0.1 | "L1" | "SGD" | ...
...
5 Decision Tree - 2
M1.f3()
> 0
= 0
M4.f5()
> 0
= 0
3 Decision Tree - 1
Solver
="SGD"
!="SGD"
Tol
= 0.0
> 0.0

Figure 2: The workflow of DPFuzz. The three main tasks are (1) fuzzing to generate interesting inputs, (2) clustering to find classes of performance functions, and (3) explaining different performance classes with decision tree-1 in the space of program inputs and decision tree-2 in the space of program internals.

clusters. In particular, there is an unexpected performance differences between 'tol' = 0.0 and 'tol' > 0.0 (even for 'tol' = 0.000001). The inputs with 'tol' = 0.0 follow (slow) blue cluster whereas the inputs with positive values follow (fast) black cluster.

The next step is to help the library maintainer explain the differences between the fast and slow clusters in terms of library internals such as function calls. DPFuzz obtains 1,673 features about the internals of logistic regression through instrumentations. The decision tree model in Figure 1 (d) shows the discriminant model for 'newton-cg' solver in the space of the internal features (chosen from a set of 5 accurate models). The model shows that the number of calls to '`if np.max(absgrad)`' in the `optimize` module is the discriminant that distinguishes the blue and black clusters:

```python
while k < maxiter:
  # call to compute gradients
  absgrad = np.abs(fgrad)
  if np.max(absgrad) < tol:
    break
  ...
  # inner loop: call to compute loss
```

The outer loop of Newton iterations will be unnecessary taken even if `np.max(absgrad)` becomes 0.0 for the case where the tolerance ('tol') is set to zero. This wastes CPU resources by calling to compute the gradient, Hessian, and loss unnecessarily. We mark this as performance bug ①. With this error localization, the library maintainer can see that the fix is to replace the strict inequality with a non-strict one. We reported this bug [29], and the developers have confirmed and fixed it using this information [32].

Another interesting performance issue in logistic regression is related to 'saga' solver. DPFuzz automatically found the issues in the solver that was already reported in the issue database [27]. Figure 1 (b) shows the discriminant learned for 'saga' solver.

Comparison with the existing fuzzers. The existing performance fuzzers such as SlowFuzz [16] and PerfFuzz [8] are looking for the worst-case execution time, and it makes them unlikely to find differential performance bugs. In Figure 1 (a), the worst-case behavior (the red function) is not related to the bug found by DPFuzz. The performance bug ① is found because of the differences between blue and black functions, and neither of them is the worst-case behavior. Exploring and explaining various classes of performance are the novelty in DPFuzz that find these subtle performance bugs.

B) Inside DPFuzz Figure 2 shows different components of DPFuzz. The *first* component of DPFuzz is to generate inputs. For this part, we extend the evolutionary fuzzing algorithms [1, 9] with functional data analysis and clustering [6, 17]. Our fuzzing approach considers multiple populations (one population per a distinct path in the CFG). Then, it picks a cluster, a path from the cluster, and an input from the selected path. Next, it mutates and crossovers the input and runs the input on the target program. This returns the cost of executions (either in terms of actual execution times or in terms of executed lines), and the path characterization. The fuzzing approach adds a new input to the populations if the input has visited a new path in the program or the input has achieved higher costs in comparison to the inputs in the same simple path. The fuzzing stops after T time units and provides generated inputs for debugging.

The *second* component of DPFuzz is to characterize different performance classes. The set of inputs in a path (or a population) defines a performance function varied in the input size. Given n paths (corresponds to n performance functions), DPFuzz applies clustering algorithms to partition theses functions into k classes of performances ($k \le n$). The clustering is primarily based on the non-parametric functional data clustering [6] with l_1 distance. The clustering finds similar input classes and separates classes with significant performance differences. The plot in Figure 1 (a) is generated as a result of fuzzing and clustering steps.

The *third* component of DPFuzz is to explain the differential performance in terms of program inputs and internals. The CART decision tree inference [2] is mainly used to obtain the explanation models. In the space of program inputs, the features are input parameters such as the value of "solver", and the labels are the performance classes from the clustering algorithm. The decision tree-1 in Figure 2 is a sample model in the space of program inputs. The model shows what input parameters are common in the same cluster and what the parameters distinguish different clusters. The models in Figure 1 (b) and (c) are produced from this step. Using this decision tree, the user may realize that all or some aspects of differential performance are unexpected. The idea is to find code regions that contribute to the creation of an unexpected performance.

In the space of program internals, the instrumentation of target programs is used to obtain program internal traces. For this aim, tracing techniques [14] are applied to generate a trace of execution for inputs from either the whole population or relevant to the unexpected performance. Next, we gather program internal features from these traces. The features used in this work are the number of calls to functions, conditions, and loops. Given these program internal features and the performance class labels (from the clustering algorithm), the problem of localizing code regions (related to the differential performance) becomes a standard classification problem. The decision tree-2 in Figure 2 is learned in the space of program internal features that show what properties of program internals are most likely responsible for differential performance. Figure 1 (d) is produced from this step.

3 PROBLEM STATEMENT

Following the work of Hartmanis and Stearns [5], it is customary to characterize the resource complexity of a program as a function of the program input size characterizing the worst/average/best performance. However, often there are latent modes in the program inputs characterizing widely different complexity classes, and knowing the existence and explanation of their differences will serve as a debugging aid for the developers and users. We study the problem of discovery (via evolutionary fuzzing) and explanation (via classifications) of latent resource complexity classes. To formalize our problem, we use the following abstract model:

Definition 3.1 (Performance Abstraction of a Program). *An abstract performance model $[\![\mathcal{P}]\!]$ of a program $\mathcal{P}$ is a tuple $[\![\mathcal{P}]\!] = (X, Y, \mathcal{O}, \pi)$ where:*

- *$X = \{x_1, \ldots, x_n\}$ is the set of input variables characterizing the input space $\mathcal{D}_X$ of the program,*
- *$Y = \{y_1, y_2, \ldots, y_m\}$ is the set of trace variables characterizing the execution space $\mathcal{D}_Y$ of the program,*
- *$\mathcal{O} : \mathcal{D}_X \to \mathcal{D}_Y$ is the functional output of the program summarizing effect of the input on trace variables, and*
- *$\pi : \mathcal{D}_X \to \mathbb{R}_{\geq 0}$ is the performance (running time or memory usage) of the program.*

Following the asymptotic resource complexity convention, we assume the existence of a function $|\cdot| : \mathcal{D}_X \to \mathbb{N}$ providing an estimate of the input size. For ML applications, the size can be a product of number of samples and features. For a subset $S \subseteq \mathcal{D}_X$ of the input space, we define its performance class as the function $\pi_S : \mathbb{N} \to \mathbb{R}_{\geq 0}$ defined as worst-case complexity given below:

$$\pi_S \stackrel{\text{def}}{=} n \mapsto \sup_{\mathbf{x} \in S \,:\, |\mathbf{x}| = n} \pi(\mathbf{x}).$$

Note that π_S is a partial function for finite sets $S \subset \mathcal{D}_X$. Let Π be the set of all performance classes. We introduce the distance function $d_p : \Pi \times \Pi \to \mathbb{R}_{\geq 0}$ on the set of performance classes parameterized with p. The typical choice for d_p is p-norm ($p = 1, 2, \infty$) and R^2 coefficient of determinations.

Definition 3.2 (Differential Performance Fuzzing). *Given a program $\mathcal{P} = (X, Y, \mathcal{O}, \pi)$ and a separation bound $\varsigma > 0$, the differential performance fuzzing problem is to find a set partition $P = \{D_1, D_2, \ldots, D_k\}$ of $\mathcal{D}_X$ such that for every $A, B \in P$ we have that $d_p(\pi_A, \pi_B) > \varsigma$.*

Even when the input space is finite, the differential performance fuzzing problem requires an exhaustive search in the exponential set of subsets of input space, and hence is clearly intractable. In Section 4.1, we present an evolutionary algorithm to solve the problem by restricting the performance classes to polynomial functions.

Once the fuzzer reports a partition of the input space into sets with distinguishable performance classes, the debugging problem is to find an explanation of the distinction between various performance classes. Oftentimes, this explanation can not be reported based solely on the values of the input variables and is a function of syntactic structure (Y variables) of the program. To enable such explanations, we study the discriminant learning problem, where the goal is to learn the differences between various partitions of the inputs as predicates over the trace variables.

A predicate over the execution space $\mathcal{D}_Y$ is a function $\phi : \mathcal{D}_Y \to \{\mathsf{T}, \mathsf{F}\}$. Let $\Phi(\mathcal{D}_Y)$ be the set of predicates over $\mathcal{D}_Y$. Given an input partition $P = \{D_1, D_2, \ldots, D_k\}$ of a program $[\![\mathcal{P}]\!] = (X, Y, \mathcal{O}, \pi)$, a discriminant is function $\Delta : P \to \Phi$ such that for every $D_i \in P$ we have that $\mathbf{x} \in D_i$ implies $\Delta(D_i)(\mathcal{O}(\mathbf{x})) = \mathsf{T}$.

Definition 3.3 (Differential Performance Debugging). *Given a program $\mathcal{P} = (X, Y, \mathcal{O}, \pi)$ and an input partition $P = \{D_1, \ldots, D_k\}$, the differential performance debugging problem is to find a discriminant $\Delta : P \to \{\mathsf{T}, \mathsf{F}\}$ of P.*

Tizpaz-Niari et al. [39] showed that the differential performance debugging problem is already NP-hard for programs with a finite set of inputs. In Section 4.2, we present a data-driven approach to learn discriminants as decision trees.

4 DATA-DRIVEN APPROACH

To address problems 3.2 and 3.3, we propose a data-driven approach in two steps. First, we extend gray-box evolutionary fuzzing algorithms [1, 9] to generate performance differentiating inputs. Our fuzzer is equipped with clustering to identify performance classes while generating inputs. Second, we use discriminant learning [38] to pinpoint code regions connected to the differential performance.

4.1 Differential Performance Fuzzing

The overall fuzzing algorithm is shown in Algorithm 1. Given a program $\mathcal{P}$, the goal is to discover widely varying performance classes. Next, we describe important components of our evolutionary search based Algorithm 1.

Cost Measure. The performance (cost) model π summarizes the resource usages. We consider both 1) abstractions of resource usages such as the number of lines executed and 2) concrete resource usages such as the execution times.

Trace Summary. We consider an instantiation of trace summary function $\mathcal{O}$ for fuzzing where the function $\mathcal{O}$ takes an input $x \in X$ and returns a set of edges in the control flow graph (CFG) visited during the execution of the input x.

Path Model. We characterize the subset S of input space using information from program traces. In particular, we use the path information to determine if two inputs are in the same class $s \in S$. We represent paths using the hash values of their ids. For each edge in the CFG, we consider a unique edge id. Then, we apply a hash function H on the set of edge ids visited for executing an input. Two distinct inputs $x_1, x_2 \in X$ are in the same performance class if $H(y_1) = H(y_2)$ where $y_1 = \mathcal{O}(x_1)$ and $y_2 = \mathcal{O}(x_2)$.

Algorithm 1: DPFuzz: Evolutionary fuzzing for differential performance.

```
Input: Program $\mathcal{P}$, initial seed $X$, max. number of iterations
       $N$, steps to do clustering $M$, the tolerance $\epsilon$, the
       separation $\varsigma$, num. of clusters $k$.
Output: Inputs/clusters manifest performance of $\mathcal{P}$.
1  for $x \in X$ do
2      $path, cost \leftarrow$ run($\mathcal{P}, x$)
3      $Cov[path]$.add($cost$), $Pop[path]$.add($s$)
4  $clusters \leftarrow$ clust($Cov, Pop, \epsilon$)
5  $step \leftarrow 1$
6  while $step \leq N \wedge |clusters_\varsigma| < k$ do
7      if $step$ % $M = 0$ then
8          $clusters \leftarrow$ clust($Cov, Pop, \epsilon$)
9      $cluster \leftarrow$ choice$_R$($clusters$)
10     $path \leftarrow$ choice$_W$($cluster$)
11     $x \leftarrow$ choice$_W$($Pop[path]$)
12     $x' \leftarrow$ mutate($x$)
13     $path', cost' \leftarrow$ run($\mathcal{P}, x'$)
14     if $path' \notin Cov.keys()$ then
15         $Cov[path']$.add($cost'$),$Pop[path']$.add($inp'$)
16     else if promisingInput($cost', path', n$) then
17         $Cov[path']$.add($cost'$),$Pop[path']$.add($inp'$)
18     $step \leftarrow step + 1$
19 return $Cov, Pop, clusters$.
```

New Input. We add an input to the population if the path is new, i.e. the id of the path is not in the coverage key (line 14 in Algorithm 1), or the input has a higher cost in comparison to other inputs in the same path (line 16 in Algorithm 1).

Functional Data Clustering. We use non-parametric functional data clustering algorithms [6] to cluster paths into a few similar performance groups (line 4 and 8 in Algorithm 1). For a set of inputs $x_1, \ldots, x_n$ mapped to the same path h, i.e. $H(y_1) = \ldots = H(y_n) = h$, we fit linear $a|x| + b$ and polynomial $a|x|^b$ functions [4] to model the performance function π_h. Then, we calculate the l_1 distances between the performance functions and apply `KMeans` clustering [10] with the tolerance bound $\epsilon > 0$ on the distance matrix to partition the paths into k clusters Σ. The clustering guarantees the following condition: if two paths (and their corresponding performance functions) are in the same cluster ($h, h' \in \Sigma_i$), then $d_1(\pi_h, \pi_{h'}) \leq \epsilon$.

The `clust` function in Algorithm 1 works as the following: starting with one cluster ($k = 1$), we apply KMeans clustering and check if the condition holds, i.e. all the functions in the same cluster are ϵ close to each other. If this is the case, we return the clusters. Otherwise, we increase k to $k + 1$ and run the algorithm again. The clustering algorithm helps explore (few) paths with distinguishable performance functions as opposed to (too many) paths in the program. Each cluster contains one or more paths that have similar performance behaviors. The selection function (line 10 in Algorithm 1) chooses a path inside a cluster based on weighted probabilities where a path with higher score of cost has a better chance of selection. Similar criterion has used to choose an input from the set of possible inputs inside the path population (line 11 in Algorithm 1).

Mutations and Crossover. We consider 8 well-established [1, 8, 16] mutation operations to guide the search algorithm. We also consider crossover operation where it mixes the current input with another input from the population.

Termination Conditions. The Algorithm 1 terminates either if it reaches to the maximum steps N or there are k clusters with significant performance differences (line 6 in Algorithm 1).

4.2 Differential Performance Debugging

Given the set of inputs $Pop=\{X_1, \ldots, X_k\}$ and their performance label Σ from the fuzzing step, the algorithm 2 explains the differential performance in the inputs. First, the user of DPFuzz (optionally) runs the clustering algorithm with the parameter k to obtain performance clusters (line 1 in Algorithm 2). The debugging procedure uses the inputs Pop as features and their clusters as labels to learn a decision tree model that explains the differential performance in the space of input parameters. We use CART decision tree algorithms [2] to learn the set of predicates P in the space of input parameters (line 3 in Algorithm 2). For example, in the decision tree of Figure 1, `solver`='saga'∧`multi-class`='multinomial' ∧ `penalty`='l2' is the predicate for the green cluster such that inputs satisfying these parameters belong to the green cluster.

The critical step in debugging is to explain the differences based on the program internals. For this step, the inputs Pop are feed into the instrumented program $\mathcal{P}'$ that generates program internal features such as whether a method or a condition are invoked and how many times they are called (line 4 in Algorithm 2). Given the set of (internal) features $\{\mathcal{P}'(X_1), \ldots, \mathcal{P}'(X_k)\}$ and their cluster labels Σ, the problem of discriminant learning becomes a standard classification problem. We use CART algorithms to learn the set of predicates Φ in the space of program internal features (line 5 in Algorithm 2). These predicates partition the space of internal features into hyper-rectangular sub-spaces $\{\phi_1, \ldots, \phi_k\}$ such that if $H(O(x)) \in \Sigma_j$, then $\mathcal{P}'(x) \models \phi_j$, i.e., the predicate ϕ_j evaluates to true for the evaluation of input x trace feature given that the input x is in the performance cluster j. For example, a predicate that evaluates whether a particular method is invoked determines the complexity of its performance class. In Figure 1 (d), the predicate based on whether the number of calls to the condition `np.max(absgrad)` is more than 14 distinguishes the blue and black clusters. The debugger uses this information to localize regions in the code and to potentially fix a performance bug.

5 EXPERIMENTS

5.1 Implementation Details

Environment Setup. We use a super-computing machine for running our fuzzer. The machine has a Linux Red Hat 7 OS with 24 cores of 2.5 GHz CPU each with 4.8 GB RAM. Since the performance measure for machine learning libraries is the actual execution times (noisy observations), we re-run the generated inputs from the fuzzer on a more precise but less powerful NUC5i5RYH machine and use the measurements of this machine for clustering. We consider the version 2.7 of python and 0.20.3 of scikit-learn.

Algorithm 2: DPFuzz: Explaining differential performances.

```
Input: Program $\mathcal{P}$, instrumented program $\mathcal{P}'$, desired
       clusters $k$, inputs $Pop$, the class $label$.
Output: The set of predicates $P, \Phi$ explain differential
        performances.
1 $label \leftarrow$ clust($Pop, k$)
2 $P \leftarrow$ DecisionTree($Pop, label$)
3 $Y \leftarrow \mathcal{P}'(Pop)$
4 $\Phi \leftarrow$ DecisionTree($Y, label$)
5 return $P, \Phi$.
```

Fuzzing and Clustering. We implement the fuzzing of DPFuzz in python by extending the Fuzzing Book framework [43]. The implementation is over 900 lines of code and can fuzz both python and Java applications. The performance measure is the actual running times in case-studies and the number of executed lines in micro-benchmarks. We use trace library [14] (python) and Javassist [3] (Java) to model paths and measure performances. We use numpy polynomial module [12] to fit performance functions. We implement the KMeans clustering algorithm using scikit-learn [15].
Debugging and Instrumentation. We instrument python libraries with tracing [14] and Java applications with Javassist [3] to extract internal features. We implement the decision tree classifier using CART algorithm in scikit-learn [15].

5.2 Micro-benchmark Results

We compare our fuzzing technique DPFuzz against state-of-the-art performance fuzzers. For the benchmark, we consider standard sorting, searching, tree, and graph algorithms from [35]. These benchmarks are standard programs used to evaluate performance fuzzers. We consider SlowFuzz [16] and PerfFuzz [8] from the literature. All three fuzzers share the same functionality such as mutations. The differences are in the population model, adding a new input to the population, and choosing an input from the population.
SlowFuzz. The SlowFuzz [16] aims to find the worst-case algorithmic complexity. The fuzzing approach has a global population where it adds a new input to the population if it achieves a higher cost (performance measure) than any other inputs in the population. The fuzzing approach chooses an input for mutations from the current population randomly.
PerfFuzz. This fuzzing [8] aims to find the worst-case algorithmic complexity for each entity (such as an edge) in the CFG. The fuzzing has a global population, and it adds a new input to the population if the input has visited a new edge (discovered a new path) or the input has achieved the highest cost in visiting at least one edge. It picks an input for mutation based on whether the input has had the highest cost for at least one entity.
DPFuzz. The fuzzing follows Algorithm 1 where there are multiple populations, one for each unique path in the CFG. An input is added to the population if the path induced from it has visited a new edge in the CFG (forms a new population) or the input has the highest cost in the population of this path. DPFuzz performs clustering after many steps (set to 1,000 in experiments) and uses the clustering information to pick an input from the population.

Empirical Research Question. For a given program over a fixed time of fuzzing, we compare the fuzzing techniques on 5 criteria: the number of generated inputs, the worst-case computational complexity in terms of executed lines, the number of visited unique paths, the number of unique performance functions, and the number of clusters of performance functions. We repeat each experiment for different fuzzers 5 times and report the best results obtained by the fuzzers. Table 1 shows the outcome of each fuzzing technique on 5 different criteria for 10 different algorithms.
Number of generated samples. We fix the duration of fuzzing to be 90 minutes for different fuzzers in all benchmarks. We compare the number of inputs generated with different fuzzers. Note that the quality of inputs such as the ones with higher costs of executions affects the number of generated inputs. In general, SlowFuzz and PerfFuzz could generate inputs faster in comparison to DPFuzz. Examples such as Quick sort, Merge sort, and Binary search provide fair comparison in terms of generated inputs since all fuzzers have similar performances in finding worst-case execution times. The slowdown in DPFuzz is almost 2× compared to SlowFuzz and PerfFuzz. We emphasize that the slowdown is expected due to functional fitting and clustering in DPFuzz.
Worst-case cost of execution. We examine the fuzzers outcomes in finding inputs with the highest cost in terms of executed lines. Table 1 shows that DPFuzz finds inputs with higher costs in 5 out of 10 benchmarks in comparison to SlowFuzz and PerfFuzz.
Discovered paths. We consider the number of unique paths discovered by the fuzzers. A path is unique if it has visited an edge that is not visited by any other paths. Table 1 shows SlowFuzz has discovered the fewest paths. In 3 out of 10 cases, PerfFuzz has discovered more paths compared to other fuzzers.
Number of distinct performance functions. One requirement of characterizing performance classes is to have multiple inputs (varied by size) for a path and fit performance functions. We compare the number of performance functions discovered by the fuzzers. Table 1 shows DPFuzz finds more performance functions in 7 out of 10 benchmarks.
Number of functional clusters. We consider the number of clusters in performance functions to show performance classes. We use the l_1 distance between functions for clustering. We note that the number of clusters should be chosen based on the quality of clustering for fair comparisons. As we increase the number of clusters, we measure the sum of intra-cluster distances as the error. We pick the optimal number of clusters when the error is below 1,000 in accordance with standard elbow method for choosing ideal number of clusters. Since the error value is the same in all experiments, the resulting number of clusters is a fair indication of detecting differential performance. Table 1 shows DPFuzz finds more clusters in 4 out of 10 benchmarks.
Summary. Although DPFuzz is slower in generating inputs, it outperforms other fuzzers in finding worst-case costs, performance functions, and clusters. Figure 3 shows an example of Optimized Insertion sort (InsertionX). The plots are the performances (in terms of executed lines) versus the size of inputs for $2 * 10^6$ samples generated by the fuzzers.

Table 1: Micro-benchmark results. Comparing DPFuzz vs SlowFuzz [16] and PerfFuzz [8]. Legend: #L: lines of code, #N: number of generated samples, T: fuzzing time (min), W: worst-case computation cost (in term of executed lines), #P: number of unique paths, #M: Number of distinct performance functions, #K: Number of functional clusters. Note: $M = 10^6$ and $K = 10^3$.

			SlowFuzz [16]					PerfFuzz [8]					DPFuzz				
Algorithm	#L	T	#N	W	#P	#M	#K	#N	W	#P	#M	#K	#N	W	#P	#M	#K
Quick Sort	80	90	13.6M	721	5	4	2	10.8M	716	14	8	3	7.8M	721	14	11	3
3-Ways Q-Sort	83	90	12.0M	756	4	2	1	12.1M	801	10	6	3	7.1M	847	10	8	5
InsertionX Sort	42	90	15.0M	496	3	2	1	15.2M	490	10	4	2	10.9M	497	10	9	4
Merge Sort	53	90	24.9M	516	4	1	1	22.5M	516	6	5	1	10.4M	516	6	5	1
Binary Search	75	90	11.2M	530	7	4	1	11.2M	527	35	21	6	5.7M	529	34	26	6
Seq. Search	26	90	43.8M	60	2	2	1	50.0M	60	6	3	1	13.2M	72	6	4	1
Boyer Moore	88	90	29.2M	204	4	0	0	34.7M	372	8	1	1	9.6M	372	8	1	1
BST Insert	47	90	21.2M	501	6	3	1	18.1M	561	13	12	3	7.7M	566	13	12	5
Is BST	141	90	22.3M	280	14	7	1	26.3M	224	46	19	2	10.0M	280	40	25	2
Prim's MST	294	90	7.9M	1,006	10	5	2	7.9M	975	119	23	5	5.3M	1,020	118	70	11

Listing 1: InsertionX

```
def insertionX(a,n):
  exchange, i = 0, n−1
  while i > 0:
    if a[i] < a[i−1]:
      swap(arr,i,i−1)
      exchange += 1
    i −= 1
  if exchange == 0:
    return a
  i = 2
  while i < n:
    v, j = a[i], i
    while v < a[j−1]:
      a[j] = a[j−1]
      j −= 1
    a[j], i = v, i+1
  return a
```

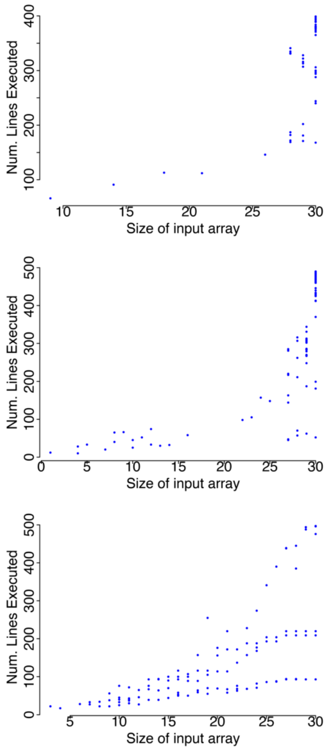

Figure 3: InsertionX. Results for SlowFuzz [16], PerfFuzz [8], and DPFuzz, from top to bottom, respectively.

6 ML LIBRARY ANALYSIS

We analyze 8 larger machine learning (ML) libraries from scikit-learn [15]. Although our approach is general enough to apply to any software library and system (we currently support both Python and Java applications), there are properties in ML applications that make our approach more practical. In particular, there is usually a clear distinction between the parameters of the library and the data in ML libraries. Once we fix the parameters in the given ML algorithm, the execution times are often a simpler function (e.g. linear and quadratic) of the number of samples and features in the data. General programs often do not follow such a structured discipline. The main research questions are "does DPFuzz (a) scale well for real-world ML libraries and (b) provide useful information to debug performance issues?"

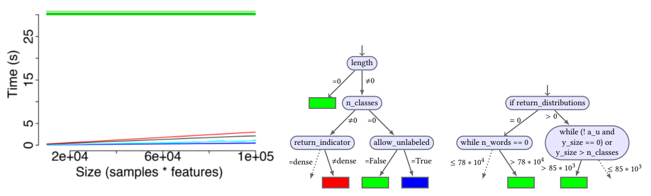

Figure 4: (a) Inputs of make classification are clustered into 5 groups. (b) Decision tree model based on input parameter features. (c) Decision tree model based on internal features.

A) Logistic Regression Classifier. In summary, DPFuzz detects 4 clusters after fuzzing for about 2 hours. The debugging revealed a performance bug in the implementation of logistic regression that has since been fixed (see details in Overview section 2).

B) Make Classification Data Set Util. We analyze the performance of `make_multilabel_classification` method inside `sample_generator` module [25].

Fuzzing and clustering. DPFuzz provides 243 sets of inputs related to different paths in the module after running for 4 hours. The fuzzing covers 293 LoC from almost 350 LoC in the implementations of this method and its dependencies. Figure 4 (a) shows that the 243 performance functions are clustered into 5 groups. The clustering shows a huge differential performance between the green cluster and other clusters.

Analyzing input space. Figure 4 (b) shows that if the length parameter sets to 0, then the input is in the green cluster. Another way to see the expensive green cluster is to set the value of n_class parameter to 0 and allow_unlabeled parameter to False.

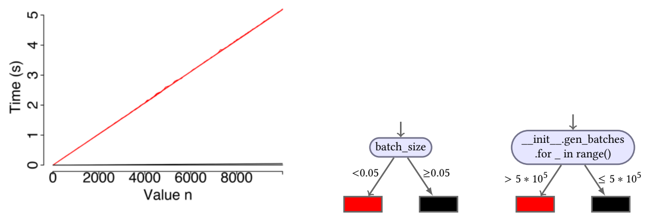

Figure 5: (a) Inputs generated by DPFuzz for util module clustered. (b) Decision tree using the input parameters of util. (c) Decision tree using the internal features of util.

Bug localization. The decision tree model in Figure 4 (c) shows the green cluster is associated with the calls to two different loops. The following code snippet shows these parts in make multilabel classification module:

```
def sample_example():
  ...
  while (not allow_unlabeled and
    y_size == 0) or y_size > n_classes:
      y_size = generator.poisson(n_labels)
  ...
  while n_words == 0:
      n_words = generator.poisson(length)
  ...
```

The code snippet shows the possibility of an infinite number of executions for the two loop bodies (the fuzzer terminates processes if the execution of inputs takes more than 15 minutes). We mark these two parts in make_multilabel_ classification method performance bugs (2),(3). We have reported these issues to scikit-learn developers [30]. They confirmed the bugs and have since fixed them [31]. The fix does not allow parameters n_classes and length to accept zero values. There are also two division by zero crashes in make_classification method discovered during fuzzing if the n_classes parameter or n_clusters_per _class parameter set to zero:

```
  ...
  weights = [1.0 / n_classes] * n_classes
  ...
  weights[k % n_c] / n_clusters_per_class
```

Scalability. In 240 minutes, DPFuzz generates 243 performance functions. For debugging, DPFuzz generates 293 internal features and uses the features to learn the decision tree in 1(s).
Usefulness. DPFuzz discovers and pinpoints 2 performance bugs and 2 division by zero crashes in the implementations of make_multi label_classification method.
C) Batch Generator. We analyze the implementation of util[3] in [15]. This module provides various utilities such as generating slices of certain sizes for the given data.
Fuzzing and clustering. We fuzz batch generator methods of this module for 30 minutes and obtain 20 sets of inputs. Figure 5 (a) shows that there are two clusters of performance.
Analyzing input space. Figure 5 (b) shows that the differences between red and black patterns are related to batch_size parameter and the library accepts small positive float values for this parameter such as 0.001.

[3]https://github.com/scikit-learn/scikit-learn/blob/master/sklearn/utils/__init__.py

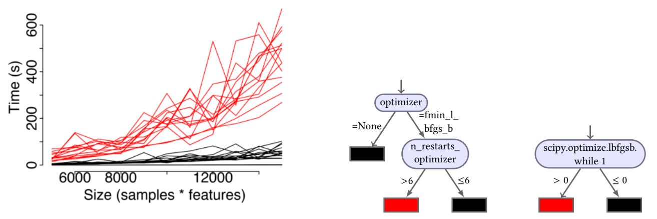

Figure 6: (a) Inputs generated by DPFuzz for Gaussian classifier clustered. (b) Decision tree using the input parameters of Gaussian. (c) Decision tree using the internal features of Gaussian.

Bug localization. Figure 5 (c) pinpoints the following loop body in gen_batches():

```
  for _ in range(int(n // batch_size)):
    end = start + batch_size
    ...
```

The decision tree shows that the loop above can be taken millions of times if the batch_size sets to small positive values close to 0. We mark this as performance bug (4). This bug has since been confirmed and fixed by the developers [33, 34]. The fix checks the batch_size parameter to be both integer and greater than or equal to 1.
Scalability. During 30 mins of fuzzing, DPFuzz generates 20 performance functions. DPFuzz generates 15 internals features and uses the features to infer the decision tree in 0.1(s).
Usefulness. DPFuzz discovers and pinpoints a performance bug in the util module.
D) Gaussian Process Classification. We analyze the implementations of Gaussian Process (GP) as a classifier model [22] in the scikit-learn library [15]. This classifier is specifically used for probabilistic classification (see [21] for more details about the functionality of this classifier). We fix non-deterministic (stochastic) behaviors in the library to a deterministic random value.

Fuzzing and clustering. We run DPFuzz for 240 minutes and obtain 173 sets of inputs. Figure 6 (a) shows a huge performance difference between the black and red clusters (more than 10 times).
Analyzing input space. Figure 6 (b) shows that if the optimizer parameter sets to 'fmin_l_bfgs_b' and n_restarts_optimizer parameter sets to values more than 6, then the input is in the (slow) red cluster. Otherwise, the input follows the (fast) black cluster.
Bug localization. Figure 6 (c) shows that the number of calls to the loop body inside scipy.optimize.lbfgsb module causes the performance differences:

```
while 1:
  ...
  _lbfgsb.setulb(...
  ...
```

It seems that the Gaussian classifier runs for the number of n_restarts_optimizer parameter and it causes huge performance differences by calling to lbfgsb optimizer. Since the behavior is related to external library, we left further analysis for future work to determine if the behavior is intrinsic to the problem, or it is a performance bug.

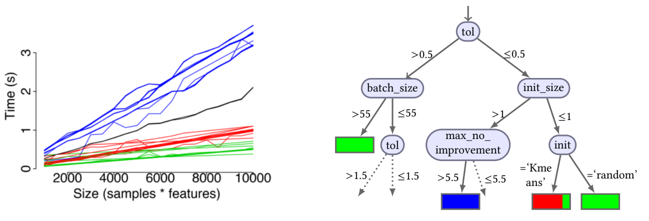

Figure 7: (a) Inputs generated by DPFuzz for mini-batch `KMeans` are clustered into 4 groups. (b) Decision tree using the input parameters of `KMeans`.

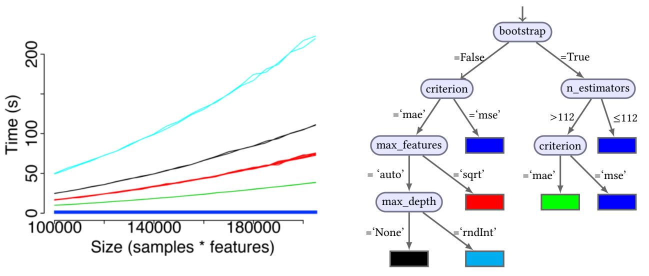

Figure 8: (a) Inputs generated by DPFuzz for forest `regressor` are clustered into 5 groups. (b) Decision tree using the input features of forest `regressor`.

Scalability. During 240 mins, DPFuzz obtains 173 performance functions. DPFuzz generates 1, 333 features about program internals and uses the features to learn the decision tree model in 5(s).
Usefulness. DPFuzz discovers and pin-points a code region in an external library (scipy optimizer).
E) Mini-batch KMeans. We analyze the implementations of KMeans mini-batch clustering [26]. This is a variant of the KMeans algorithm which uses mini-batches to reduce the computation time, while still attempting to optimize the same objective function. We assume that the initial cluster centroids are deterministically chosen by setting seed values to a constant value.

During a run of 240 mins, we obtain 325 input sets from DPFuzz. Figure 7 (a) shows that 325 functions are clustered in 4 groups. Figure 7 (b) shows the expensive blue cluster happens if the 'tol' is less than or equal to 0.5, the 'init_size' is larger than or equal to 1, and the 'max_no_improvement' is larger than 5.5. Looking into the internal features, we observe that the number of calls to calculate the euclidean distance between points and the squared differences between the current and the previous errors are important discriminant features. However, the discriminant features seem to explain behaviors that are intrinsic to the problem.
F) Random Forest Regressor. A random forest [28] is a meta estimator that fits a number of classifying decision trees on various sub-samples of the dataset and uses averaging to improve the predictive accuracy and control over-fitting. We analyze its implementations in scikit-learn library [15].

We obtain 160 sets of inputs from DPFuzz after running for 240 mins. Figure 8 (a) shows (subset of) inputs are clustered into 5 groups. The clustering shows that the computational complexity in the random forest regressor can be quadratic. The decision tree model in Figure 8 (b) shows that the inputs with criterion parameter sets to 'mae' have higher costs. A similar issue has already reported as a potential performance bug in the regressor [18]. Learning forest regressors is expensive with 'mae' criteria since it sorts inputs in each step of learning to calculate the median. Since the root causes seem to be related to external library, we left further analysis for future work to determine if the behavior is intrinsic to the problem, or it is a performance bug.
G) Discriminant Analysis. The discriminant analysis [23] is a classic classifier with linear and quadratic decision boundaries. We analyze linear and quadratic discriminant analysis implemented in scikit-learn [15]. During 52 mins of fuzzing, we obtain 78 sets of inputs. Upon clustering, we observe that the point-wise distances between clusters are in order of fractions of a second. Therefore, we gain more confidence that the discriminant analysis is free of performance issues.
H) Decision Tree Classifier. The decision tree classifier [20] is a white-box classification model that predicts the target variable by inferring decision rules. We analyze the implementations of this model in scikit-learn [15]. After fuzzing for 240 mins, DPFuzz generates 492 sets of inputs. Upon clustering, we realize the point-wise distances between centroids are less than 0.1 second. Thus, we become more confident that the decision tree classifier is free of performance bugs.

7 RELATED WORK

Performance Fuzzing. Evolutionary algorithms have been widely used for finding inputs that trigger the worst-case complexity [8, 16]. SlowFuzz [16] extends the libFuzzer [9] to discover DoS bugs, that is, inputs with expensive computations such as exponential. In contrast, DPFuzz is looking for different classes of computational complexities rather than only the worst-case one. PerfFuzz [8] is the closest fuzzing technique to us. The goal is to maximize the cost of different entities in the program (edges in CFG) that help characterize the worst-case behaviors in large-scale systems. PerfFuzz has a single global population whereas DPFuzz has multiple populations, one population per path. This model of population in DPFuzz enables the debugger to model performance functions precisely and obtain diverse classes of performances. In addition, DPFuzz utilizes clustering during fuzzing to discover a few diverse performance classes rather than all entity classes, many of those may have similar performance.
Differential Fuzzing. DifFuzz [11] has developed on top of AFL [1] and Kelinci [7] to discover information leaks due to timing side channels in Java programs. DifFuzz adapts the traditional notion of confidentiality, noninterference. A program is *unsafe* iff for a pair of secret values s_1 and s_2, there exists a public value p such that the behavior of the program on (s_1, p) is observably different than on (s_2, p). The goal of DifFuzz is to maximize the following objective: $\delta = |c(p, s_1) - c(p, s_2)|$, that is, to find two distinct secret values s_1, s_2 and a public value p that give the maximum cost (c) difference in two runs of a program. If the difference of any pair of secret values is more than ϵ, the program is considered to be vulnerable to timing side-channel attacks. In contrast to DifFuzz, we are looking

for k sets of inputs to discover differential performances in python-based machine learning libraries. Fuchsia [40] is another technique that performs differential analysis to debug timing side channels. For detection, Fuchsia extends AFL to characterize response times as functions over public inputs. Fuchsia performs fuzzing and clustering in two separate steps, wheres DPFuzz combines these two steps to explore the space of input efficiently. In addition, the time model in Fuchsia can be in arbitrary shapes, while the performance model in DPFuzz often follows simpler shapes such as polynomials.
Debugging Performances. Machine learning and statistical models have been used for fault localization [42] and debugging of performance issues [36, 38, 39]. These works generally assume that the interesting inputs are given, while we adapt evolutionary-based fuzzing techniques to automatically generate interesting inputs. DPDEBUGGER [38] considers a model of programs where the inputs are not gathered as functional data. Therefore, it needs to discover performance functions. DPDEBUGGER extends Kmeans and Spectral clusterings to find clusters of performance from independent data points. On the contrary, DPFuzz considers a model of programs where inputs are given as functional data. Then, it uses non-parametric functional data clustering [6] to detect clusters of performance. DPDEBUGGER [38] is limited to linear performance functions, while DPFuzz can model complex functions such as polynomials. Similar to [38], DPFuzz uses decision tree classifiers to pinpoint code regions contributing to the differential performance.

8 THREAT TO VALIDITY

Evolutionary-based Fuzzing. Our approach requires a diverse set of inputs generated automatically using the fuzzing component. The quality of the debugging significantly depends on the characterization of differential performance in the given input set. Similar to existing evolutionary-based fuzzers, our approach relies solely on heuristics to generate a diverse set of inputs and is not guaranteed to find inputs that characterize all performance classes.
Benign Differential Performance vs Performance Bugs. In general, it is indeed challenging to determine whether a differential performance is a bug or it is intrinsic to the problem being solved. To mitigate this issue, we turn to debugging with the help of auxiliary features from the space of inputs and internals.

First, we find an explanation based on the input features such as the type of solver. Based on this, the user may decide the differences due to using solver='A' versus solver='B' are benign whereas the differences due to using tolerance=0.0 versus tolerance=0.000001 under the same solver='A' are unexpected.

Once an unexpected behavior is detected, the next step is to investigate the issue further in the source code and localize suspicious code regions for a fix. These two steps help the user determine whether the differences are intrinsic, or they are performance bugs that need to be fixed.
Experiments and Comparisons to Existing Fuzzers. Existing approaches in fuzzing such as SlowFuzz [16] and PerfFuzz [8] are mainly developed for C and C++ programs and extended recently for Java applications [11]. Our work provides substantial (and unprecedented) support specifically for python-based ML libraries.

To enable ourselves to compare DPFuzz against the existing performance fuzzers, we adapt their main fuzzing algorithm and implement them in our python-based fuzzing framework. This, however, can lead to degradations in the performance of these fuzzers. To alleviate this in our comparisons, we use the same functionality in all aspects of three fuzzers such as mutation and crossover operations. The differences are in modeling population (multiple populations with clustering versus single global population) and adding new inputs to the population (based on the cost, the path, or combinations). These differences are the specific choices of each fuzzer to achieve certain goals as described in their algorithms.
Overhead in Dynamic Analysis. We proposed a dynamic analysis approach to discover and understand performance bugs. Dynamic analysis often scales well to large applications. However, as compared with static analysis, they present additional overheads such as time required to discover variegated inputs and time needed for data collection.
Polynomial Functions and Decision Tree Models. Our design guiding principles are based on two important factors: efficiency (especially for fuzzing) and human interpretability (for debugging). For example, we restrict the search for performance functions to be polynomials so as to generate inputs quickly. Other models such as Gaussian processes can lead to better results, but they may degrade the throughput of fuzzing. Similarly, we use decision tree models to give interpretable explanations. Graph models can be used to learn complex discriminants and overcome the decision tree limitations such as the hyper-rectangular partitions of search spaces. However, such models are notorious to be uninterpretable.
Standard Algorithms as Benchmarks. We use standard sort, search, tree, and graph algorithms to evaluate our fuzzing. While these algorithms do not contain performance bugs, they have (well-known) diverse classes of performances, and finding those classes is a hard problem. Characterizing these classes in the well-known algorithm through fuzzing is important to manifest differential performance bugs in real-world applications.
Machine Learning Libraries as Case Studies. In this work, we focus on medium-sized ML libraries for few reasons. First, there is a little bit of support for the performance aspects of ML libraries. Second, there is often a clear distinction between data and parameters in ML libraries, and they tend to have simpler performance functions such as linear or polynomial. Our approach is applicable for general software given that there is a clear measure defined to map inputs to the size. Examples are the number of bytes in a file, the number of set bits in a key, and the number of Kleene star in a regular expression. Given this measure, our approach can characterize performance differences and aid to localize the root causes. We left further analysis to apply this technique on general and large software for future work.
Time Measurements. We use actual execution times as opposed to abstractions such as the number of executed lines in the case studies. While this is important to factor the cost of black-box components (such as external libraries and solvers in other languages) during the fuzzing, the noise in timing observations can lead to false positive in the fuzzing process. To overcome this issue, we re-run the inputs once more on NUC5i5RYH machine to allow for higher precisions. To further mitigate the effects of environmental factors in NUC measurements, we run the libraries in isolations and take the average of timing measurements over multiple samples.

9 CONCLUSION AND FUTURE WORK

We developed a method and a tool for differential performance analysis. We showed that the fuzzing, clustering, and decision tree algorithms presented for functional data are scalable to debug real-world machine learning libraries. In addition, we illustrated the usefulness of our approach in finding multiple performance bugs in these libraries and in comparing to existing performance fuzzers.

For future work, there are few interesting directions. One direction is to study security implications of differential performance. The feasibility of (hyper)parameter leaks [41] via timing side channels in ML applications is a relevant and challenging open problem. Another direction is to study the relationships between accuracy and performance. Given a lower-bound on the accuracy of a learning task, the idea is to synthesize parameters and hyper-parameters in the model such that the performance of underlying systems such as IoT and CPS are optimized.

ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers for their comments to improve this paper. In addition, the authors thank the developers of `scikit-learn` for discussing the potential issues reported by us. This work utilized resources from the University of Colorado Boulder Research Computing Group, which is supported by NSF, CU Boulder, and CSU. This research was supported by DARPA under agreement FA8750-15-2-0096.

REFERENCES

[1] AFL. 2016. American fuzzy lop. http://lcamtuf.coredump.cx/afl/. Online.
[2] Leo Breiman, Jerome Friedman, Charles J Stone, and Richard A Olshen. 1984. *Classification and regression trees.* CRC press.
[3] Shigeru Chiba. 1998. Javassist - a reflection-based programming wizard for Java. In *Proceedings of OOPSLA'98 Workshop on Reflective Programming in C++ and Java*, Vol. 174.
[4] Simon F Goldsmith, Alex S Aiken, and Daniel S Wilkerson. 2007. Measuring empirical computational complexity. In *FSE.* ACM, 395–404.
[5] J. Hartmanis and R. E. Stearns. 1965. On the Computational Complexity of Algorithms. *Trans. Amer. Math. Soc.* 117 (1965), 285–306. http://www.jstor.org/stable/1994208
[6] Julien Jacques and Cristian Preda. 2014. Functional data clustering: a survey. *Advances in Data Analysis and Classification* 8, 3 (2014), 231–255.
[7] Rody Kersten, Kasper Luckow, and Corina S Păsăreanu. 2017. POSTER: AFL-based Fuzzing for Java with Kelinci. In *CCS.* ACM, 2511–2513.
[8] Caroline Lemieux, Rohan Padhye, Koushik Sen, and Dawn Song. 2018. Perffuzz: Automatically generating pathological inputs. In *ISSTA.* ACM, 254–265.
[9] libFuzzer. 2016. A library for coverage-guided fuzz testing (part of LLVM 3.9). http://llvm.org/docs/LibFuzzer.html. Online.
[10] Stuart Lloyd. 1982. Least squares quantization in PCM. *IEEE transactions on information theory* 28, 2 (1982), 129–137.
[11] Shirin Nilizadeh, Yannic Noller, and Corina S. Pasareanu. 2019. DifFuzz: Differential Fuzzing for Side-Channel Analysis. *ICSE* (2019). http://arxiv.org/abs/1811.07005
[12] Travis Oliphant. 2006–. NumPy: A guide to NumPy. http://www.numpy.org/.
[13] Oswaldo Olivo, Isil Dillig, and Calvin Lin. 2015. Static detection of asymptotic performance bugs in collection traversals. In *PLDI*, Vol. 50. ACM, 369–378.
[14] Zooko O'Whielacronx. 2018. A program/module to trace Python program or function execution. https://github.com/python/cpython/blob/2.7/Lib/trace.py. Online.
[15] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12 (2011), 2825–2830.
[16] Theofilos Petsios, Jason Zhao, Angelos D Keromytis, and Suman Jana. 2017. Slowfuzz: Automated domain-independent detection of algorithmic complexity vulnerabilities. In *CCS.* ACM, 2155–2168.
[17] James O Ramsay. 2006. *Functional data analysis.* Wiley Online Library.
[18] Scikit-learn. 2017. Trees with MAE criterion are slow to train. https://github.com/scikit-learn/scikit-learn/issues/9626. Online.
[19] Scikit-learn. 2018. Sqeuclidean metric is much slower than euclidean. https://github.com/scikit-learn/scikit-learn/issues/12600. Online.
[20] Scikit-learn. 2019. Decision Tree Classifier. https://scikit-learn.org/stable/modules/tree.html. Online.
[21] Scikit-learn. 2019. Gaussian Process Classifier in scikit-learn: description. https://scikit-learn.org/stable/modules/gaussian_process.html#gaussian-process-classification-gpc. Online.
[22] Scikit-learn. 2019. Gaussian Process Classifier in scikit-learn: implementations. https://scikit-learn.org/stable/modules/generated/sklearn.gaussian_process.GaussianProcessClassifier.html. Online.
[23] Scikit-learn. 2019. Linear and quadratic discriminant analysis. https://scikit-learn.org/stable/modules/lda_qda.html. Online.
[24] Scikit-learn. 2019. Logistic Regression in scikit-learn. https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression. Online.
[25] Scikit-learn. 2019. Make Multilabel Classification. sklearn.datasets.make_multilabel_classification.
[26] Scikit-learn. 2019. Mini-batch KMeans. sklearn.cluster.MiniBatchKMeans.
[27] Scikit-learn. 2019. Performance of Logistic Regression with saga. https://github.com/scikit-learn/scikit-learn/issues/13316. Online.
[28] Scikit-learn. 2019. Random Forest Regressor. sklearn.ensemble.RandomForestRegressor.
[29] Scikit-learn. 2020. Performance bug in logistic regression with newton-cg. https://github.com/scikit-learn/scikit-learn/issues/16186. Online.
[30] Scikit-learn. 2020. Performance bug in Make Classification Data Set Util. https://github.com/scikit-learn/scikit-learn/issues/16001. Online.
[31] Scikit-learn. 2020. Performance bug in Make Classification Data Set Util fixed. https://github.com/scikit-learn/scikit-learn/pull/16006/files. Online.
[32] Scikit-learn. 2020. Performance bug in regression with newton-cg fixed. https://github.com/scikit-learn/scikit-learn/pull/16266/files. Online.
[33] Scikit-learn. 2020. Performance bug in Util Batch Generator module. https://github.com/scikit-learn/scikit-learn/issues/16158. Online.
[34] Scikit-learn. 2020. Performance bug in Util Batch Generator module fixed. https://github.com/scikit-learn/scikit-learn/pull/16181/files. Online.
[35] Robert Sedgewick and Kevin Wayne. 2011. Algorithms (4th ed.). Addison-Wesley Professional.
[36] Linhai Song and Shan Lu. 2014. Statistical debugging for real-world performance problems. *OOPSLA* 49, 10 (2014), 561–578.
[37] Tensorflow. 2019. Transpose can be very slow on CPU. https://github.com/tensorflow/tensorflow/issues/27383. Online.
[38] Saeid Tizpaz-Niari, Pavol Černý, Bor-Yuh Evan Chang, and Ashutosh Trivedi. 2018. Differential Performance Debugging with Discriminant Regression Trees. In *AAAI.* 2468–2475.
[39] Saeid Tizpaz-Niari, Pavol Černỳ, Bor-Yuh Evan Chang, Sriram Sankaranarayanan, and Ashutosh Trivedi. 2017. Discriminating Traces with Time. In *TACAS.* Springer, 21–37.
[40] Saeid Tizpaz-Niari, Pavol Cerny, and Ashutosh Trivedi. 2020. Data-Driven Debugging for Functional Side Channels. https://arxiv.org/abs/1808.10502. In NDSS.
[41] Binghui Wang and Neil Zhenqiang Gong. 2018. Stealing hyperparameters in machine learning. In *2018 IEEE Symposium on Security and Privacy (SP).* IEEE, 36–52.
[42] W Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. 2016. A survey on software fault localization. *IEEE Transactions on Software Engineering* 42, 8 (2016), 707–740.
[43] Andreas Zeller, Rahul Gopinath, Marcel Böhme, Gordon Fraser, and Christian Holler. 2019. The Fuzzing Book. In *The Fuzzing Book.* Saarland University. https://www.fuzzingbook.org/ Retrieved 2019-09-09 16:42:54+02:00.