PABLO: Helping Novices Debug Python Code Through Data-Driven Fault Localization

Benjamin Cosman
UC San Diego
blcosman@eng.ucsd.edu

Madeline Endres
University of Michigan
endremad@umich.edu

Georgios Sakkas
UC San Diego
gsakkas@eng.ucsd.edu

Leon Medvinsky
UC San Diego
leonkm2@illinois.edu

Yao-Yuan Yang
UC San Diego
b01902066@ntu.edu.tw

Ranjit Jhala
UC San Diego
jhala@cs.ucsd.edu

Kamalika Chaudhuri
UC San Diego
kamalika@cs.ucsd.edu

Westley Weimer
University of Michigan
weimerw@umich.edu

ABSTRACT

As dynamically-typed languages grow in popularity, especially among beginning programmers, there is an increased need to pinpoint their defects. Localization for novice bugs can be ambiguous: not all locations formally implicated are equally useful for beginners. We propose a scalable fault localization approach for dynamic languages that is helpful for debugging and generalizes to handle a wide variety of errors commonly faced by novice programmers. We base our approach on a combination of static, dynamic, and contextual features, guided by machine learning. We evaluate on over 980,000 diverse real user interactions across four years from the popular PythonTutor.com website, which is used both in classes and by non-traditional learners. We find that our approach is scalable, general, and quite accurate: up to 77% of these historical novice users would have been helped by our top-three responses, compared to 45% for the default interpreter. We also conducted a human study: participants preferred our approach to the baseline ($p = 0.018$), and found it additionally useful for bugs meriting multiple edits.

ACM Reference Format:
Benjamin Cosman, Madeline Endres, Georgios Sakkas, Leon Medvinsky, Yao-Yuan Yang, Ranjit Jhala, Kamalika Chaudhuri, and Westley Weimer. 2020. PABLO: Helping Novices Debug Python Code Through Data-Driven Fault Localization. In *The 51st ACM Technical Symposium on Computer Science Education (SIGCSE '20), March 11–14, 2020, Portland, OR, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3328778.3366860

1 INTRODUCTION

A key part of learning to program is learning to localize the root cause of a failure. Novices unfamiliar with debugging tools or lacking the expertise to interpret compiler error messages can have a difficult time pinpointing (and hence fixing) what *caused* their program to crash [7, 8, 23–25, 36, 43]. Solutions to this problem are needed as class sizes grow and increasing numbers of non-traditional learners progress with limited access to human instructional support.

```
1  year = int(time.strftime("%Y"))
2  age = input("Enter your age")
3  print("You will be twice as old in:")
4  print(year + age)
```

Figure 1: Multiple fault localizations (lines 1 and 2).

Dynamically-typed languages like Python are increasingly popular for teaching programming [12], and many of those learning Python require help debugging their code. For example, the PythonTutor online debugging and visualization tool alone is visited by more than sixty thousand users per month, and almost half of the program crashes they face take multiple attempts to resolve.

One key difficulty in localizing bugs is that there are often multiple possible fixes, some of which may be more informative to programmers [34]. Consider the program in Figure 1, adapted from our dataset. The program attempts to carry out arithmetic based on a given number and the current year, but raises an exception on line 4 (when adding an integer to a string). One possible "fix" would be to *remove* the `int()` conversion from line 1: the `+` on line 4 would then be string concatenation. However, another reasonable fix (and the one that the novice actually used) would be to add an `int()` cast on line 2. Both fixes are well-typed; but the second is more likely to help a novice.

Thus, a useful debugging aid needs to provide debugging hints that are *helpful*, implicating locations that correspond to developer intent or aid novice learning. It should also be *general*, applying to a wide variety of the errors that novices frequently encounter, including more complex bugs involving multiple simultaneous conceptual mistakes spanning multiple lines. Finally, it should be *scalable*, working quickly in large classes and non-traditional settings.

Our key insight is that we can *learn* a suite of heuristics for how to debug and fix programs from a large corpus of real-world examples of novices fixing their own bugs. Novice-written bugs and fixes contain static, dynamic and contextual information about where errors appear frequently *in practice* and about how those errors can be fixed.

SIGCSE '20, March 11–14, 2020, Portland, OR, USA

ACM ISBN 978-1-4503-6793-6/20/03…$15.00
https://doi.org/10.1145/3328778.3366860

We build on this insight to present a novel approach to localizing defects in unannotated, beginner-written programs without test suites. We learn from a corpus of novice programs and produce answers which agree with human actions and judgments. Our system Pablo (Program Analytics-Based Localization Oracle):

(1) takes as input a set of pairs of programs, each representing a program that crashes and the fixed version of that program,
(2) computes a *bag of abstracted terms* (BOAT) [34] representation for each crashing program: each AST node is abstracted to a vector of syntactic, dynamic, and contextual features
(3) trains a classifier on these vectors

Then, given a new novice-written crashing program, Pablo will:

(1) compute BOAT vectors for each AST node of the program
(2) classify each vector to obtain the likelihood that each corresponding node is to blame for the crash
(3) return a list of k program locations, ranked by likelihood

Our hypothesis is that the combination of a rich corpus and domain-specific features admits automatically classifying common classes of defects and causes, thereby helping novices find the true sources of their bugs. In this paper, we make the following contributions:

(1) We identify a set of features that admit precise classification for dynamic imperative languages. We introduce and evaluate static, dynamic, contextual and slice-based features that enable *scalable* data-driven localization for Python.
(2) We evaluate Pablo in a user study and find that subjects find it *helpful*, and also *general* enough to provide useful debugging hints when multiple fixes are appropriate, which the baseline Python interpreter cannot do.
(3) We perform a systematic evaluation on over 980,000 programs from four years of interactions with the PythonTutor [12] website. Pablo is *helpful*, correctly identifying the exact expression that humans agree should be changed 59–77% of the time, compared to the Python interpreter's baseline of 45%, and *general*, retaining strong accuracy in all of the most common classes of bugs that novices encounter.

2 ALGORITHM OVERVIEW

We present Pablo, an algorithm for helping novices localize faults [13] in their Python programs that exhibit non-trivial uncaught exceptions. We do not consider syntax errors or references to undefined variables. Our algorithm uses machine learning models based on static, dynamic, contextual and slicing features to implicate suspicious expressions. Unlike short ranked lists [24, Sec. 5.6], voluminous fault localization output is not useful to developers in general [37] and novices in particular [14]. We produce Top-1 and Top-3 rankings; short lists are especially relevant for novices, who frequently make mistakes spanning multiple program locations.

Our algorithm first extracts static and dynamic features from a Python program (Section 2.1). Next, using a labeled training corpus, we train a machine learning model over those features (Section 2.3). Once the model has been trained, we localize faults in new Python programs by extracting their features and applying the model.

Drawing inspiration from localization algorithms such as Nate [34] and the natural language processing term frequency vector (or "bag of words") model [33], we represent each buggy program as a "bag of abstracted terms". A *term* is either a statement or expression.

We observe that many errors admit multiple logically-valid resolutions (see Figure 1): we thus cannot effectively localize through type constraints alone. Instead, we use static features to capture structured program meaning, contextual features to capture the relationship between a program fragment and its environment, and dynamic features to reason about conditional behavior. Dynamic features are calculated using a trace of the program [3] after applying a semantics-preserving transformation [10] that admits expression-level granularity (instead of Python's whole lines).

2.1 Model Features

Syntactic Forms (Static). We hypothesize that certain syntactic categories of terms may be more prone to bugs than others. For example, students might have more trouble with loop conditions than with simple assignments. The first feature we consider is the syntactic category of a node. This feature is categorical, using syntax tree labels such as Return Statement or Variable Expression.

Expression Size (Static). This feature counts the number of descendents in each subtree. Our intuition is that larger, complex expressions may be more fault prone.

Type (Dynamic). We observe that some types may be inherently suspect, especially for beginner-written code. For example, there are few reasons to have a variable of type `NoneType` in Python. This categorical feature includes all basic Python types (`int`, `tuple`, etc.) and three special values: `Statement` for statements, `Unknown` for expressions that are never evaluated, and `Multiple` for expressions which are evaluated multiple times in a trace and change type.

Slice (Dynamic). The goal of this feature is to help eliminate terms that cannot be the source of the crash [34]. We compute a dynamic program slice [15]: a set of terms that contributed at runtime to the exception. This boolean feature encodes whether the term is a member of the slice, and is discussed further in Section 2.2.

Crash Location (Dynamic). We observe that the precise term raising the exception is frequently useful for understanding and fixing the bug. This boolean feature flags the exception source.

Exception Type (Dynamic). The type of error thrown is useful for localization. For example, every division term may be more suspicious if the uncaught exception is `DivisionByZero`. We encode the exception type categorically.

Local contextual features. Our BOAT representation, like term frequency vectors in general, does not include contextual information such as ordering. However, the meaning of an expression may depend on its context. For example, consider each `0` in `x = 0` and `x / 0`. Both have the same syntactic form, type, etc. To distinguish the `0` in `x / 0` we require contextual information. Structures like syntax trees and flow graphs capture such contextual information, but are not immediately applicable to machine learning.

We desire to encode such information while retaining the use of scalable, accurate, off-the-shelf machine learning algorithms that operate on feature vectors. We thus embed context in a vector, borrowing insights from standard approaches in machine learning.

We associate with each term additional features that correspond to the static and dynamic features of its parent and child nodes. For representational regularity, we always model three children; terms without parents or three children are given special values (e.g., zero or NotApplicable) for those contextual features.

Global contextual features. We hypothesize that advancing novices will both use a wider range of Python syntax and face different kinds of errors. We thus add boolean features indicating which node types appear anywhere in the program. These features will be sparsely populated for all nodes in simpler programs, and densely populated in programs using richer language features.

2.2 Dynamic Slicing Algorithm

Slicing information can help our model avoid implicating irrelevant nodes in the fault localization. Program slicing is a well-studied field with many explored tradeoffs (e.g., see Xu *et al.* for a survey [39]). We desire a slicing algorithm that is efficient (to scale to hundreds of thousands of instances) yet will admit high accuracy: we achieve this by focusing on features relevant to beginner-written programs. We follow the basic approach of Korel and Laski [15, 16], building a graph of data and control dependencies. We then traverse the graph backwards to collect the set of terms that the excepting line transitively depended on. This excludes lines that could not have caused the exception, such as lines that never ran, or lines that had no connection to the excepting step.

To balance ease of prototype implementation against coverage for beginner-written programs, our slicing algorithm handles every syntax node supported by the Python3 standard `ast` library except:

- Assignments where the left hand side is not a variable or a simple chain of attribute indexing or subscripting
- Assignments where attribute indexing or subscripting on the left hand side means something other than the default, (e.g., if the operations are overridden by a class)
- Lambda, generator and starred expressions
- Set and dictionary comprehensions
- Await, yield, delete, with and raise statements
- Variable argument ellipsis
- Coroutine definitions and asynchronous loops

Dynamic slicing involves an unavoidable tradeoff between unsoundness and over-approximation. For example, when the condition of an `if` statement is not met and so the code it guards is not run, we exclude the entire `if` block from the slice, even though the bug may indeed be in the condition. We observe that one common case this strategy fails is the "early break" case (Figure 2). We thus check if a break, return, or other statement for escaping structured control flow is present inside of a conditional statement, and add dependencies in the dependency graph between the enclosing conditional and the statements that would have been skipped by the break or return. While this heuristic is effective in practice, it does not overcome all related problems: we thus treat slice information as one of many features rather than as a hard constraint.

2.3 Machine Learning Model Generation

We formulate the localization problem as a standard binary classification problem. For each term in a program, we extract the features

```
1  while True:
2    x = float(input())
3    if x != 0: # bug: should be ==, not !=
4      break
5    print("One over your number is: %d" % (1 / x))
```

Figure 2: Example of slicing imprecision (line 3 and line 5).

(Section 2.1) and we assign it a label representing whether it should be blamed (Section 3.2). We represent all features numerically by performing one-hot encoding on categorical features.

Random Forests. We choose to work with random forest models [5], which train groups of decision trees, to balance accuracy with scalability to our large dataset. Each decision tree is a binary tree of simple thresholding classifiers. At each node, the training procedure chooses a feature, and then directs each incoming sample to one of its two children by comparing that feature to the chosen threshold. The feature and threshold are chosen to minimize the impurity of the two resulting partitions (e.g., measured by the Gini index or entropy). Decision trees scale well to large datasets and can learn non-linear prediction rules [6, 30, 31].

Decision trees are prone to overfitting. To mitigate this problem, each tree in a random forest is trained on a subset of the data and a subset of the features. The prediction and confidence of the model as a whole is a weighted average of the predictions and confidences of the individual trees. Random forests thus trade some of the low computational cost of a plain decision tree for additional accuracy. We use 500 trees, each with a maximum depth of 30. Other parameters use the default scikit-learn [27] settings.

Training methodology. Given feature vectors describing every term of every program in our dataset, we train a model on a random 80% of the data and report the model's performance on the remaining 20%. Programs by the same user are always assigned together to either training or testing. We report the average of five such 80–20 splits. Each trained model takes in a feature vector representing a single term in a buggy program, and returns a confidence score representing how likely it is that the term was one of the terms changed between the fixed and buggy programs. We treat the model as providing a *ranking* over all terms by confidence. For a given k, we score the model based on Top-k accuracy: the proportion of programs for which a correct answer (i.e., a term that was actually changed historically) is present in the top k results. This is an imbalanced dataset in that non-buggy terms are much more common than buggy terms, so during training we re-weight to the reciprocal of the frequency of the corresponding class.

3 EVALUATION

We conducted both a large-scale empirical evaluation of Pablo and also a human study to address these research questions:

RQ1 Do our localizations agree with human judgements?
RQ2 Which model features are the most important?
RQ3 How well does our algorithm handle different Python errors?
RQ4 Is our algorithm accurate on diverse programs?
RQ5 Do humans find our algorithm useful when multiple lines need to be edited?

3.1 Dataset and Program Collection

Our raw data consist of every Python 3 program that a user executed on PythonTutor.com [12] (not in "live" mode) from late 2015 to end of 2018, other than those with syntax errors or undefined variables. Each program which throws an uncaught Python exception is paired with the next program by the same user that does not crash, under the assumption that the latter is the fixed version of the former. We discard pairs where the difference between crashing and fixed versions is too high (more than a standard deviation above average), since these are usually unrelated submissions or complete refactorings. We also discard submissions that violate PythonTutor's policies (e.g., those using forbidden libraries).

Ultimately, the dataset used in this evaluation contains 985,780 usable program pairs, representing students from dozens of universities (PythonTutor has been used in many introductory courses [12]) as well as non-traditional novices.

3.2 Labeled Training and Ground Truth

Our algorithm is based on supervised machine learning and thus requires labeled training instances — a ground truth notion of which terms correspond to correct fault localizations. We use the terms changed in fixes by actual users as that ground truth. Many PythonTutor interactions are iterative: users start out by writing a program that crashes, and then edit it until it no longer crashes. Our dataset contains only those crashing programs for which the same user later submitted a program that did not crash. We compute a tree-diff [17] between the original, buggy submission and the first fixed submission. For example, if the expression `len({3,4})` is changed to `len([3,4])`, then the node corresponding to the set `{3,4}` as a whole will appear in the diff (since it has been changed to a list), but neither its parent node nor its children nodes appear.

We define the *ground truth* correct answer to be the set of terms in the crashing program that also appear in the diff. We discuss the implications of this choice in Section 3.9. Given that notion of ground truth, a candidate fault localization answer is *accurate* if it is in the ground truth set. That is, if the human user changed terms X and Y, a technique (either our algorithm or a baseline) is given credit for returning either X or Y. A ranked response list is *top-k accurate* if any one of the top k answers is in the ground truth set.

3.3 RQ 1 — Fault Localization Helpfulness

We train random forests and compute their Top-1, Top-2, and Top-3 accuracy. For a baseline we compare to the standard Python interpreter, i.e., blaming the expression whose evaluation raises the uncaught exception. For fairness, we modify the Python interpreter to report expressions instead of its default of whole lines (see Section 2). We discuss other fault localization approaches and why they are not applicable baselines for our setting in Section 4.

Pablo produces a correct answer in the Top-1, Top-2, and Top-3 rankings 59%, 70%, and 77% of the time. The expression blamed by the Python interpreter is only changed by the user 45% of the time.

Thus, our most directly-comparable model (Top-1), significantly outperforms this baseline. Users who are only willing to look at a single error message would have been better-served by our Top-1 model on this historical data. In addition, previous studies have shown that developers are willing to use very short ranked lists [24, Sec. 5.6], but not voluminous ones. Our Top-3 accuracy of 77% dramatically improves upon the current state of practice for scalable localization in Python.

3.4 RQ 2 — Feature Predictive Power

Having established the efficacy of our approach, we now investigate which elements of our algorithmic design (Section 2) contributed to that success. We rank the ~500 features in our model by Gini importance (or mean decrease in impurity), a common measure for decision tree and random forest models [5]. Informally, the Gini importance conveys a weighted count of the number of times a feature is used to split a node: a feature that is learned to guide more model classification decisions is more important. We also rank the features by a standard analysis of variance (ANOVA). In both cases, we find that a mixed combination of static, dynamic, and contextual features are important: no single category alone suffices.

To support that observation, we also present the results of a leave-one-out analysis in which entire categories of features are removed and the model is trained and tested only on those that remain. When the model is trained without typing, syntactic, or contextual features, the model's accuracy drops by 14%, 15%, and 19% respectively. We generally conclude that syntactic, dynamic, and contextual features (i.e., the design decisions of our algorithm) are crucial to our algorithm's accuracy.

3.5 RQ 3 — Defect Categories

We investigate the sensitivity of our algorithm to different categories of Python errors: does Pablo apply to many kinds of novice defects? We investigate training and testing Top-1 decision trees on only those subsets of the dataset corresponding to each of the five most common uncaught exceptions: `TypeError`, `IndexError`, `AttributeError`, `ValueError`, and `KeyError`. Together, these five exceptions are the most common faced by our novices, making up 97% of the errors in our dataset (54%, 23%, 11%, 7%, and 3%, respectively).

These per-defect models have normalized accuracy between 86% and 115% of a comparable model trained on the dataset as a whole. This shows that our algorithm is robust and able to give high-accuracy fault localizations on a variety of defect types. Having consistent, rather than defect-type-sensitive, performance is important for debugging-tool usability [1, 4, 24].

3.6 RQ 4 — Diversity of Programs

To demonstrate that our evaluation dataset is not only larger but also more diverse than those used in previous work, we compare the diversity of programs used here to those in a relevant baseline. The Nate algorithm [34] also provides error localization using a machine-learning approach, and its evaluation also focused on beginner-written programs. However, Nate targets strongly statically typed OCaml programs: submissions to just 23 different university homework problems. Such a dataset is comparatively homogeneous, raising concerns about whether associated evaluation results would generalize to more diverse settings. In contrast, in our PythonTutor dataset, users were not constrained to specific university assignments. We hypothesize that the data are thus more heterogeneous. To assess this quantitatively, we used agglomerative clustering to find the number of "natural" program categories

present in both our dataset and the Nate dataset. Datasets with more natural program categories are more heterogeneous.

Distance Metric. Many clustering algorithms depend on distance metrics. To measure the distance between programs, we flattened their ASTs into strings of tokens, and then computed the Levenshtein edit distance [18]. We do not compute an AST distance directly since that is less tractable on our large dataset (i.e., cubic [26]). Levenshtein distance is not a good *absolute* measure of program diversity since similar programs can have different tree structures, but it does show *comparative* diversity.

Clustering Algorithm. We performed agglomerative clustering on the datasets of flattened programs [22]: every datapoint starts in its own cluster, and the two closest clusters are merged until there are no clusters that differ by less than some threshold. We used a single linkage approach in which the distance between clusters is the minimum distance between their elements. To account for differences between Python and OCaml, we z-score nodes against not only others at the same tree depth, but also against others one or two levels below [41]. This makes cluster counts at each threshold value comparable. Our implementation uses the standard SCIPY library (`scipy.cluster.hierarchy.fcluster` with the `inconsistent` method).

We compare the Nate dataset to a random sample of equal size from our dataset. For all values of the inconsistency threshold, there are at least 48% more clusters in our sample than in the OCaml dataset. This suggests that our dataset contains a more diverse set of programs. We are not claiming any advances in clustering accuracy in this determination (indeed, scalability concerns limited us to coarser approaches); instead, our claim is that even with simple clustering, it is clear that our dataset contains a greater diversity of programs, even when controlling for size, than were considered by previously-published evaluations. We view it as an advantage of our algorithm that it can apply to many different program categories.

3.7 RQ 5 — Multi-Edit Bug Fixes

In addition to the automated metrics described above, we also evaluate PABLO in an IRB-approved human study. We selected 30 programs at random from the PythonTutor dataset, and presented each with 3 highlighted lines representing the Top-3 output of PABLO. For this study we worked at the granularity of lines rather than of expressions to simplify the presentation of three distinct and non-overlapping localizations for comparison. Each participant was shown a random 10 of these annotated-program stimuli and asked, for each highlighted line, whether it "either clarifies an error's root cause or needs to be modified?" Not all participants answered all questions, but we were able to use data from 42 participants in our analyses.

Overall, participants find the first and second lines from PABLO useful 75% and 28% of the time; at least one of PABLO's top three is useful 84% of the time. On the other hand, participants find the line indicated by Python's error message helpful only 77.3% of the time. That is, the output of PABLO outperforms vanilla Python by 6.5% ($p = 0.018$, two-tailed Mann-Whitney test).

When considering only the 14 programs with complex bugs where the original novice programmer made edits to multiple lines, humans find PABLO even more helpful; PABLO's first and second

```
1  def devowel(word):
2      word = 'foo'
3      w_list = list(word)
4      vowels = ['a', 'e', 'i', 'o', 'u']
5      for letter in w_list:
6          if letter not in vowels:
7              w_list = w_list.remove(letter)
8      word = ''.join(w_list)
9  print(devowel('foo'))
```

Figure 3: A bug - 'remove' edits in place and returns None

```
1  areaCodes = [800, 555]
2  for i in range(0, len(areaCodes) + 1):
3      print(areaCodes[i])
```

Figure 4: A program with an off-by-one bug on line 2.

lines are helpful 79% and 36% of the time, and at least one of the top three is useful 89% of the time. We observe that multi-edit bugs are quite common, accounting for almost half the bugs in our data set. For these complex multi-edit bugs, the Python interpreter alone provides novices limited support while PABLO provides additional useful information more than one-third of the time.

3.8 Qualitative Analysis

We highlight two indicative examples in detail to demonstrate how our algorithm accurately localizes faults. These examples are simplified slightly for presentation and to protect the anonymity of the programmers, but retain their essential character.

NoneType. The function in Figure 3 attempts to remove all vowels from a given word. However the `remove` method in line 7 actually modifies the list in place and returns `None`. The user-corrected version forgoes the assignment and just calls `w_list.remove(letter)`. In the buggy case, Python does not crash until line 8, where its message is the somewhat-misleading `TypeError: can only join an iterable`. However, PABLO flags the correct statement, based on the features that it is an assignment statement whose second child has type `NoneType`. This captures the intuition that there is rarely a good reason to assign the value `None` to a variable in novice programs.

Note that PABLO uses both the syntactic form of the statement (an assignment) and the dynamic type of one of its children, so all our categories — syntactic, dynamic, and contextual — were useful.

Off-by-one bugs. In Figure 4, the programmer incorrectly adds one to the high end of a range, causing an `IndexError: list index out of range` on line 3 during the final iteration of the loop. The correct expression to blame is the addition `len(areaCodes) + 1`. Some of the features PABLO uses to correctly localize this bug include that the error is an index error, the type of the parent, and the fact that it is an addition expression. This example also highlights the use of all three categories of features simultaneously to capture a notion of root cause that more closely aligns with human expectations.

3.9 Threats to validity

Although our evaluation demonstrates that our algorithm scales to accurately localize Python errors in large datasets of novice programs, our results may not generalize to all use cases.

Language choice. We have only demonstrated that our technique works for Python 3. We hypothesize that it should apply to similar dynamically-typed languages, such as Ruby, but such evaluations remain future work. We mitigate this threat slightly by constructing

our algorithm so that it does not depend on Python-specific features (e.g., in Section 2.2 we explicitly omit relatively "exotic" features such as generators that may not be present in other languages).

Target population. Unlike many classroom studies of students, we have less information about the makeup of our subject population. The general popularity of PythonTutor is an advantage for collecting a large, indicative dataset, but it does mean that we have no specific information about the programmers or what they were trying to write. In general, while the website is used by many classes, most of the users appear to be non-traditional students; our results may apply most directly to that population.

Ground truth. The size of our dataset precluded the manual annotation of each buggy program. Instead, we used historical successful edits from actual users as our ground truth notion of the desired fault localization. This has the advantage of aligning our algorithm with novice intuitions in cases where there are multiple logically-consistent answers (see Figure 1), and thus increasing the utility of our tool. However, this definition of ground truth may be overly permissive: the next correct program in the historical sequence may contain additional spurious changes beyond those strictly needed to fix the bug. We mitigate this threat by discarding as outliers program pairs that had very large relative changes.

3.10 Evaluation Summary

> Pablo is *helpful*, providing high-accuracy fault localization that implicates the correct terms 59–77% of the time (for Top-1 to Top-3 lists, compared to the baseline Python interpreter's 45% accuracy) and outperforming the baseline ($p = 0.018$). Pablo is *general*, performing similarly on the top five exceptions that make up 97% of novices crashes, and providing helpful information for multi-line fixes 36% of the time (compared to the baseline Python interpreter's 0%).

In addition, we investigated our algorithm's design decisions, finding that all our categories (i.e., static, dynamic, and contextual) were critical. Our evaluations involved over 980,000 beginner-written Python programs as well as a direct human study of 42 participants.

4 RELATED WORK

Broadly, the most relevant areas of related work are software engineering approaches to fault localization (typically based on dynamic test information) and programming languages approaches to fault localization (typically based on static type information). Fault localization has only increased in relevance with the rise of automated program repair [21], where many techniques depend critically on accurate fault localization [29].

A significant body of work in fault localization follows from the Tarantula project [13]. Jones *et al.* proposed that statements executed often in failing test runs but rarely in successful test runs were likely to be implicated in the defect. Such "spectrum"-based approaches gather dynamic information and rank statements by a mathematically-computed suspiciousness score. Projects like Multric [40] use machine learning to combine these spectrum-based suspiciousness scores based on empirical data, and Savant [2], TraPT [19], and FLUCCS [35] refine this process by also using inferred invariants, mutation testing, and code metrics like age and churn. CrashLocator [38] computes suspiciousness scores without needing positive test cases, but it requires a large suite of crashing cases as well as an oracle that groups crashes by similarity.

Where spectrum-based approaches traditionally focus on industrial-scale programs, we target beginner-written software. Spectrum methods require multiple test cases (ideally very many of them); we use just one program execution and our work is aimed at novices who may not even be familiar with the notion of test suites. Spectrum methods focus heavily on dynamic features; we make critical use of syntactic, contextual and type information as well. Indeed, the machine learning based approaches above use features that are entirely disjoint from ours and inapplicable in our setting, with the sole exception of the code complexity metrics of FLUCCS. Unlike spectrum features, many of our features have no obvious connection to faultiness, so our surprising positive result is that we can still use these features to localize faults in our domain. In addition, some human studies have focused specifically on accuracy and expertise for fault localization [11, 28, 32]; our decision to use features to support novices is informed by such insights.

Zeller's popular Delta Debugging algorithm [42], interpreted generally, efficiently finds a minimal "interesting" subset from among a large set of elements. When the set of elements represents changes made to a source code version control system and interesting is defined with respect to failing a test suite, it can quickly locate program edits that cause regressions. Alternatively, when considering correct and failing execution states, such approaches can help focus on variable values that cause failures [9]. While our slicing information can be viewed as a coarse approximation to the precise, fine-grained localization such an approach can provide, a key difference is our use of machine learning to agree with human judgments in cases where multiple causes are equally logically valid.

In the programming languages community, a large body of work has focused on localizing faults and providing better error messages, typically through the use of type information. Mycroft [20] modifies existing type inference algorithms to produce a "correcting set" for a program with a type error. Mycroft assumes that the minimal such set is the most desirable to the user, and has no way to rank multiple equally small sets. We instead use machine learning to agree with human judgements. Our work is most directly inspired by the Nate [34] system, which introduced the notion of training classifiers over pairs of buggy-and-fixed programs, and uses machine learning on static and contextual features to localize type errors in OCaml code. However, this work was limited to purely functional OCaml programs where the static type discipline was crucial in both restricting the class of errors, and providing the features that enabled learning. We employ a similar approach to localize Python faults, but we use dynamic features as well as static and contextual ones, we handle a variety of errors, and we evaluate our approach on a set of programs far more heterogeneous and more than two orders of magnitude larger.

5 CONCLUSION

We present an approach for accurately localizing novice errors in off-the-shelf, beginner-written Python programs. Our approach uses a combination of static, dynamic and contextual features. Static

features, such as syntactic forms and expression sizes, are a particularly powerful heuristic for novice programmer defects. Dynamic features can both implicate relevant terms and rule out irrelevant program regions. Contextual features allow our approach to gain the benefits of precise AST- or CFG-style information while retaining scalable performance. We use off-the-shelf machine learning to accurately combine those disparate features in a way that captures and models human judgments of ground-truth correct answers — a notion that is especially relevant when multiple program locations are equally formally implicated but not equally useful to the user.

We desire an approach that is *helpful*, *general* and *scalable*. We evaluate our approach with respect to historical defects and fixes. All feature categories (static, dynamic, and contextual) were relevant to success, as measured by multiple analyses (Gini, ANOVA and leave-one-out). Our evaluation demonstrates significant scalability and generality. Our 980,000 instances were two orders of magnitude more numerous than similar related work and measurably more diverse; we augmented our dataset with a direct human study of 42 participants. Ultimately, Pablo was quite accurate, implicating the correct program location 59–77% of the time (compared to the Python interpreter's 45% accuracy), outperforming the baseline ($p = 0.018$) and providing additional useful information 36% of the time (compared to Python's 0%).

ACKNOWLEDGMENTS

We acknowledge the partial support of the NSF (CCF-1908633, CCF-1763674) and the Air Force (FA8750-19-2-0006, FA8750-19-1-0501). We thank Philip Guo for the use of the PythonTutor data.

REFERENCES

[1] N. Ayewah and W. Pugh. The Google Findbugs fixit. In *Proceedings of the 19th International Symposium on Software Testing and Analysis*, ISSTA '10, pages 241–252, 2010.

[2] T.-D. B Le, D. Lo, C. Le Goues, and L. Grunske. A learning-to-rank based fault localization approach using likely invariants. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pages 177–188. ACM, 2016.

[3] bdb Debugger Framework. https://docs.python.org/2/library/bdb.html.

[4] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. Engler. A few billion lines of code later: Using static analysis to find bugs in the real world. *Commun. ACM*, 53(2):66–75, Feb. 2010.

[5] L. Breiman. Random forests. *Machine learning*, 45(1):5–32, 2001.

[6] L. Breiman. *Classification and regression trees*. Routledge, 2017.

[7] S. Chen and M. Erwig. Counter-factual typing for debugging type errors. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, pages 583–594, New York, NY, USA, 2014. ACM.

[8] D. R. Christiansen. Reflect on your mistakes! lightweight domain-specific error messages. In *Preproceedings of the 15th Symposium on Trends in Functional Programming*, 2014.

[9] H. Cleve and A. Zeller. Locating causes of program failures. In *ICSE*, pages 342–351, 2005.

[10] C. Flanagan, A. Sabry, B. F. Duba, and M. Felleisen. The essence of compiling with continuations. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*, PLDI '93, pages 237–247, New York, NY, USA, 1993. ACM.

[11] Z. P. Fry and W. Weimer. A human study of fault localization accuracy. In *26th IEEE International Conference on Software Maintenance*, pages 1–10, 2010.

[12] P. J. Guo. Online Python tutor: Embeddable web-based program visualization for CS education. In *Proceeding of the 44th ACM Technical Symposium on Computer Science Education*, SIGCSE '13, pages 579–584, New York, NY, USA, 2013. ACM.

[13] J. A. Jones and M. J. Harrold. Empirical evaluation of the Tarantula automatic fault-localization technique. In *Automated Software Engineering*, pages 273–282, 2005.

[14] T. Kohn. The error behind the message: Finding the cause of error messages in python. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*, SIGCSE '19, pages 524–530, New York, NY, USA, 2019. ACM.

[15] B. Korel and J. Laski. Dynamic program slicing. *Information Processing Letters*, 29(3):155 – 163, 1988.

[16] B. Korel and J. Laski. Dynamic slicing of computer programs. *Journal of Systems and Software*, 13(3):187 – 195, 1990.

[17] E. Lempsink, S. Leather, and A. Löh. Type-safe diff for families of datatypes. In *Proceedings of the 2009 ACM SIGPLAN workshop on Generic programming*, pages 61–72. ACM, 2009.

[18] V. I. Levenshtein. Binary Codes Capable of Correcting Deletions, Insertions and Reversals. *Soviet Physics Doklady*, 10:707, Feb. 1966.

[19] X. Li and L. Zhang. Transforming programs and tests in tandem for fault localization. *Proc. ACM Program. Lang.*, 1(OOPSLA):92:1–92:30, Oct. 2017.

[20] C. Loncaric, S. Chandra, C. Schlesinger, and M. Sridharan. A practical framework for type inference error explanation. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2016, pages 781–799, New York, NY, USA, 2016. ACM.

[21] M. Monperrus. Automatic software repair: A bibliography. *ACM Comput. Surv.*, 51(1):17:1–17:24, 2018.

[22] D. Müllner. Modern hierarchical, agglomerative clustering algorithms. *ArXiv e-prints*, Sept. 2011.

[23] M. Neubauer and P. Thiemann. Discriminative sum types locate the source of type errors. In *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming*, ICFP '03, pages 15–26, New York, NY, USA, 2003. ACM.

[24] C. Parnin and A. Orso. Are automated debugging techniques actually helping programmers? In *International Symposium on Software Testing and Analysis*, pages 199–209, 2011.

[25] Z. Pavlinovic, T. King, and T. Wies. Finding minimum type error sources. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14, pages 525–542, New York, NY, USA, 2014. ACM.

[26] M. Pawlik and N. Augsten. Tree edit distance: Robust and memory-efficient. *Information Systems*, 56:157 – 173, 2016.

[27] 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. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[28] S. Prabhakararao, C. Cook, J. Ruthruff, E. Creswick, M. Main, M. Durham, and M. Burnett. Strategies and behaviors of end-user programmers with interactive fault localization. In *Human Centric Computing Languages and Environments*, pages 15–22, 2003.

[29] Y. Qi, X. Mao, Y. Lei, and C. Wang. Using automated program repair for evaluating the effectiveness of fault localization techniques. In *International Symposium on Software Testing and Analysis*, 2013.

[30] J. R. Quinlan. Induction of decision trees. *Machine learning*, 1(1):81–106, 1986.

[31] J. R. Quinlan. *C4. 5: programs for machine learning*. Elsevier, 2014.

[32] J. R. Ruthruff, M. Burnett, and G. Rothermel. An empirical study of fault localization for end-user programmers. In *International Conference on Software Engineering*, pages 352–361, 2005.

[33] G. Salton and M. J. McGill. *Introduction to Modern Information Retrieval*. McGraw-Hill, Inc., New York, NY, USA, 1986.

[34] E. L. Seidel, H. Sibghat, K. Chaudhuri, W. Weimer, and R. Jhala. Learning to blame: Localizing novice type errors with data-driven diagnosis. *Proc. ACM Program. Lang.*, 1(OOPSLA):60:1–60:27, Oct. 2017.

[35] J. Sohn and S. Yoo. Fluccs: Using code and change metrics to improve fault localization. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2017, pages 273–283, New York, NY, USA, 2017. ACM.

[36] P. J. Stuckey, M. Sulzmann, and J. Wazny. Improving type error diagnosis. In *Proceedings of the 2004 ACM SIGPLAN workshop on Haskell*, Haskell '04, pages 80–91, New York, NY, USA, 22 Sept. 2004. ACM.

[37] Q. Wang, C. Parnin, and A. Orso. Evaluating the usefulness of IR-based fault localization techniques. In *International Symposium on Software Testing and Analysis*, pages 1–11, 2015.

[38] R. Wu, H. Zhang, S.-C. Cheung, and S. Kim. Crashlocator: locating crashing faults based on crash stacks. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, pages 204–214. ACM, 2014.

[39] B. Xu, J. Qian, X. Zhang, Z. Wu, and L. Chen. A brief survey of program slicing. *SIGSOFT Softw. Eng. Notes*, 30(2):1–36, Mar. 2005.

[40] J. Xuan and M. Monperrus. Learning to combine multiple ranking metrics for fault localization. In *Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on*, pages 191–200. IEEE, 2014.

[41] C. T. Zahn. Graph-theoretical methods for detecting and describing gestalt clusters. *IEEE Trans. Comput.*, 20(1):68–86, Jan. 1971.

[42] A. Zeller. Yesterday, my program worked. today, it does not. why? In *ESEC / SIGSOFT FSE*, pages 253–267, 1999.

[43] D. Zhang and A. C. Myers. Toward general diagnosis of static errors. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, pages 569–581, New York, NY, USA, 2014. ACM.