BugMiner: Automating Precise Bug Dataset Construction by Code Evolution History Mining

Xuezhi Song*, Yijian Wu†*, Junming Cao*, Bihuan Chen*, Yun Lin‡, Zhengjie Lu*, Dingji Wang*, and Xin Peng*
* Fudan University, Shanghai, China
{songxuezhi, wuyijian, 2111024004, bhchen, luzj19, dingjiwang2049, pengxin}@fudan.edu.cn
‡ Shanghai Jiao Tong University, Shanghai, China
lin_yun@sjtu.edu.cn

2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE) | 979-8-3503-2996-4/23/$31.00 ©2023 IEEE | DOI: 10.1109/ASE56229.2023.00201

***Abstract*—Bugs and their fixes in the code evolution histories are important assets for many software engineering tasks such as deriving new state-of-the-art automatic bug fixing techniques. Existing bug datasets are either manually built which is difficult to grow efficiently to a scale large enough for massive data analysis, or lack of precise information of how bugs are introduced and fixed which is critical for in-depth analysis such as buggy/fixing code identification. Moreover, the types of the bugs are typically missing in the existing bug datasets, limiting the possibility of developing high-precision type-specific approaches for enterprise-level purposes. In this work, we propose BugMiner, an approach to automatically collecting bugs from code repositories by isolating the critical changes of the bugs. We also propose a learning-based approach for automating bug type classification with relatively small manual labels of bug types. We evaluate our approach regarding the precision of bug information and the efficiency of the bug-mining process with 2,082 bugs automatically mined from 100 open-source projects. We demonstrate the improved effectiveness and efficiency in bug-fixing location identification, compared to the SOTA BugBuilder, and high recall and precision in bug-inducing location identification. We also compare our learning-based bug classification approach to traditional baseline method, indicating about 17% improvement in classification effectiveness under macro-F1.**

***Index Terms*—bug dataset, bug-fixing, bug-inducing, automatic bug mining**

I. Introduction

Bug datasets serve as a pivotal cornerstone for data-driven bug detection and fixing techniques. Automatically collecting precise bug-related data from real-world code repositories is essential for buidling large-scale dataset that is significantly useful for facilitating various research areas and industrial productivity improvement.

Recent works such as BEARS [1], BugSwarm [2], and BugBuilder [3] have emerged to automate the construction of real-world bug datasets. BEARS and BugSwarm collect reproducible bugs by monitoring the buggy and patched program versions from Continuous Integration (CI) system. Although the presence of CI data supports high precision of the collected bugs and the corresponding fixes, the difficulty in obtaining CI data restricts the bug data collection from the vast range of open-source repositories. BugBuilder is a well-designed tool to find bug-fixing revisions and to identify bug-fixing changes. It is able to collect large amount of bugs and can be used to construct a large bug dataset, similar to but much larger than Defects4J, without additional data from outside of the code repository, which is applicable in any code base with proper test code.

However, existing bug datasets, including Defects4J [4] and those mined by recent research advantages, suffer from the following weaknesses. First, the fixing patches of bugs are difficult to be identified automatically and precisely. The bugs in Defects4J are manually validated for precision, which prevents it from growing automatically. Other bug datasets may include the bug fixing commit which contains the fixing patch but the patch may not be minimal due to the fact that a commit in real-world development may contain multi-purpose source code changes. Second, the bug inducing changes are typically not included in the bug datasets. The location where a bug is fixed is not necessarily the same as where it is induced. A bug dataset that exactly marks not only bug fix patches but also bug inducing changes would enable a wide range of data-driven research such as regression bug detection. Third, bug types are not properly categorized. Missing required bug types constraints the task-specific applications of bug analysis. For example, the bug fixing patterns for null-pointer-exceptions and for concurrency bugs could be very different and may need type-based filtering in a large-scale bug dataset in order to find bugs with specified types.

In this work, we propose BugMiner, an approach to automatically collecting bugs from code repository. In contrast to existing work, BugMiner stands out by mining bugs directly from the code evolution history. It seeks potential bug-fixing commits by heuristic-based commit filtration and searches for the related test cases in the code repository. Consequently, BugMiner is able to mine a more extensive set of bugs with precision by running the tests accordingly. Moreover, BugMiner introduces a novel method for localizing critical bug changes based on delta debugging. This method demonstrates a significant efficiency improvement over the enumeration-followed-by-search strategy adopted by BugBuilder.

†Yijian Wu is the corresponding author.

2643-1572/23/$31.00 ©2023 IEEE
DOI 10.1109/ASE56229.2023.00201

We also publicize a large precise bug dataset that contains not only the bug fixing patch for each bug but also the bug inducing code, alongside the specific test cases that trigger these bugs. These test cases ensure the precise of our bug dataset, such that, for instance, a bug-fixing commit (*bfc*) is expected to pass the tests, while a bug-inducing commit (*bic*) should consistently fail them. The critical changes that induce the bugs and fix the bugs are correspondingly isolated from the original source code changes in the bug-inducing commit and bug-fixing commit such that the bug information is precise. We also employ a learning-based approach to categorize the bugs by the root causes of the bugs so that the bug types can be automatically determined.

In our experiments, we find that BugMiner is 1.2 times faster than the state-of-the-art bug dataset construction tool BugBuilder [3] by mining 432 bugs from 100 open source projects within 40 hours. BugMiner mined 2,082 bugs in around six days (157 hours), even more efficiently than BugBuilder, which mined 1,246 bugs in similar time. We also evaluate the effectiveness and efficiency of BugMiner in identifying bug-fixing locations and find that BugMiner produces about 1.5 times as many accurate bug-fixing locations in only three-fifths of time compared to BugBuilder. Moreover, BugMiner is able to identify over 80% bug-inducing locations, whereas BugBuilder is not capable. Regarding automatic bug classification, we find our learning-based approach is about 17% better in F1-score than the baseline TBCNN model, with regard to the manually labeled 1,618 bugs. The source code of BugMiner is available at https://github.com/SongXueZhi/BugMiner. The mined bug dataset and its demonstration is available at https://bugminer.github.io/.

In the rest of this paper, we first define basic concepts in Section II. In Section III, we present the overall process of BugMiner and explain the technical details in each step. In Section IV, we evaluate our approach and dataset by answering four research questions regarding the efficiency and accuracy in identifying bug fixing/inducing locations and the accuracy and diversity of bug categorization. In Section V, we discuss the strengths and weakness of our approach and the dataset. Finally, we conclude our work in Section VII.

II. Terminology and Problem Definition

We define the terminology used in this work and formulate the definition of bugs and the process of bug dataset construction.

Commit and Revision. In a code repository, a commit operation typically corresponds to two revisions, i.e., a revision before the commit operation and a revision after the commit operation. For the ease of presentation, we use the term "commit" to refer to the revision after the commit operation, which is also widely used in industrial contexts. Therefore, the term "commit" and the term "revision" are used interchangeably to represent the revision after the specific commit operation. If a commit c is applied in the code base, we denote the revision after the commit as c. The revision before this commit is denoted as $c-1$. We also use the term "commit $c-1$" to refer to this revision.

Bug-fixing Commit. A bug-fixing commit regarding a bug b is a commit c in which the bug b does not exist but does exist in the prior commit $c-1$. We denote a bug-fixing commit as *bfc*. Hence, the commit *bfc-1* is the buggy revision just prior to the *bfc*.

Bug. We define a bug as a five-tuple (*bfc, bfloc, biloc, spec, type*), where *bfc* is the bug-fixing commit, *bfloc* is the bug-fixing location where the bug-fix patch code exists, *biloc* is the bug-inducing location containing the buggy code, *spec* is the specification (typically a test case) that the correct program should hold (test passes) but the buggy program breaks (test fails), and *type* is the type of the bug that is categorized along various dimensions such as root cause or impact.

Note that the *bfloc* in the bug dataset should indicate the *minimal* change set in the bug-fixing commit, i.e., the critical bug-fixing patches/changes. This implies that any irrelevant changes like refactoring or introducing new features in the commit should be filtered out. The same minimal principle also applies to *biloc*.

Definition of the Bug Data Collection Problem. Given a code repository $Repo$, our target is to construct automatically a bug set $BUG = \{bug | bug = <$ *bfc, bfloc, biloc, spec, type* $>\}$, where $\forall bug \in BUG, \exists spec\ s.t.$ *bfc* $\in Repo$, *spec* holds in *bfc* but is broken in *bfc-1*, *biloc* $\in$ *bfc-1* is the location of code changes that directly breaks the *spec*, *bfloc* $\in$ *bfc* is the location of code changes (patch) that fix the faulty program to meet the *spec*, and *type* is the type of the bug determined by certain classification criteria.

While we use the *spec* as a guide to identify a bug, the above definition still imposes certain limitations in practice. We still need to address the following questions: (1) How to represent a *spec* and determine if a commit meets or breaks it? (2) How to identify the *biloc*, as there may be more than one location that cause the bug? For example, in the case of a null-pointer-exception (NPE) bug depicted in Figure 1a, the bug-inducing location could be determined as either the missing null-pointer detection before accessing *pomFiles* (line 17 in Figure 1b) or the incorrect value assignment of *pomFiles* in the method `getPomFiles()` (lines 24 and line 27 in Figure 1c).

Problem Reformulation. In this work, we opt to use *test cases* as the formal specification. The pass or failure of the test cases indicates whether a commit meets or breaks the specification. Consequently, we can use the test cases to determine whether the bug exists in a specific commit as long as we can run this test in the corresponding commit. Based on this, we are able to determine at which commit the bug is exactly introduced and identify the exact code changes that induce the bug. Therefore, we introduce the concept of Bug-inducing Commit and redefine the bug location *biloc*.

Bug-inducing Commit. A bug-inducing commit regarding a bug b is a commit c in which the bug b does exist but does

not exist in the prior commit $c-1$. We denote a bug-inducing commit as *bic*.

Bug location (biloc). The bug location is the minimal set of the source code changes that directly causes the bug in the buggy revision *bfc-1*. This code is introduced in bug-inducing commit *bic*.

Therefore, we redefine a bug as a six-tuple *(bfc, bic, bfloc, biloc, testcase, type)* where the $testcase$ is the test case related to the bug. We further reformulate the bug collection problem as follows. The bug dataset *BUG* is $\{bug|bug = <$ *bfc, bic, bfloc, biloc, testcase, type*$>\}$, where $\forall bug \in BUG$, $\exists spec\ s.t.$ for commits *bfc* and *bic* $\in Repo$, the $testcase$ passes in *bfc* and fails in *bfc-1* and *bic*, and the $testcase$ passes in *bic-1* or the commit *bic-1* does not contain the code tested by $testcase$, the source code change set *biloc* $\in$ *bfc-1* introduced in *bic* is the direct cause of the failure of $testcase$, the source code change set *bfloc* $\in$ *bfc* is the fixing patch of the buggy program, and *type* is the type of the bug.

III. Methodology

A. Overview

BugMiner consists of three essential steps: 1) collecting bug raw data, 2) locating critical changes, and 3) classifying bugs. The actions of each steps are depicted in Fig. 2, where a rounded-rectangle represents an action in a step and a rectangle represents the output/input artifact of the related action.

In the raw data collection step, BugMiner traverses all commits in the code repository to find all potential bug-fixing commits *pBFCs*. Then, it searches for a new test case in each commit *pbfc* $\in$ *pBFCs* and automatically runs the test case to verify whether it passes in the *pbfc*. If the test case fails, the *pbfc* is discarded. A *pbfc* is only confirmed to be a *bfc* when the test case passes in *pbfc* and fails in *pbfc-1*. We also collect commit message and related bug reports (if available) for each *bfc* as additional description for each bug, which will be useful for automatic classification later on.

Then, for each bug, BugMiner isolates the critical changes that fix the bug (*bfloc*) and those possibly induce the bug *biloc* out of all the source code changes in *bfc* and *bic*, respectively, by applying a delta-debugging-base algorithm.

Finally, a deep-learning based model is used to automatically classify bugs into different types, such as logic errors, boundary condition issues, performance problems, etc.

B. Collecting Bug Raw Data

We first search in the commit message texts for commits that have high possibility of fixing bugs by commits filtering. Then we search for test cases that could be related to the bugs. Finallly we associate commit message and issue description to the bugs for richer information that may describe the semantics related to the bugs.

1) Filtering Commits: To collect potential bug-fixing commits *pBFCs* , we examine the commit message of each commit for the presence of keywords such as *fix*, *closed*, *bug*, *issue* and other related terms. The heuristics is widely used in existing researches [5], [6], [7] which aim at gathering the commits that are likely to be related to bug fixes.

2) Searching for Test Cases: Given a potential bug-fixing commit *pbfc* $\in$ *pBFCs*, we search for related test cases in the following three sets, which we belive have high possibility to contain the test cases directedly related to the bug under investigation.

- *Test cases present in pbfc-1 and also present in pbfc*. These test cases that represent pre-existing tests that were not executed at the time the bug was introduced. It may also include test cases provided by the developer specifically for reproducing the bug before attempting to fix it.
- *Test cases added in pbfc*. These test cases were introduced in *pbfc* , but they do not exist in *pbfc-1*.
- *Test cases added in δ commits after pbfc*. These test cases are likely submitted by the developers as supplementary test cases following the bug fixes. Here, we use $\delta = 15$.

The last two search spaces imply that we need to migrate test cases from c to c_{inv} Here, c represents the commit where the test case was added, and c_{inv} represents the commit that needs to be tested but does not include the target test cases. In this case, we will use the test case migration technique proposed in RegMiner [8] for the migration process.

Note that, although we may find multiple test cases for a *bfc*, we currently consider each test to be specific to an individual bug. Finally, we will group them based on the results of bug-fixing location identification. Tests with the same bug-fixing location will be considered as tests for the same bug.

Once a test case passes on *pbfc* and fails on *pbfc-1*, we identify the *pbfc* as a *bfc* and relate the test case to the bug.

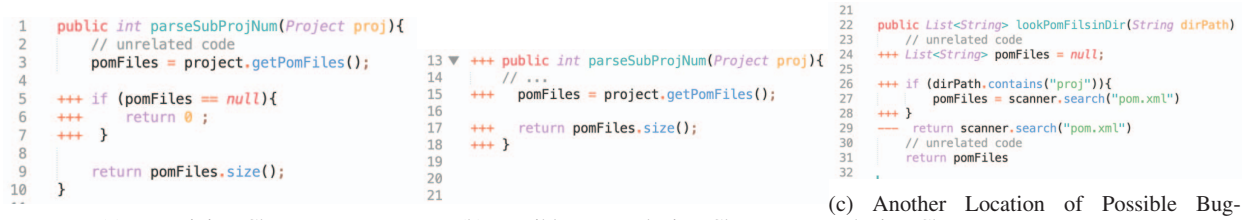
3) Relating Commit Message and Issue Description to the Bug: If a *bfc* references an issue in the commit message and the issue description also mentions the *bfc*, we consider the issue to be related to the bug. In such cases, we relate the *bfc* and the issue to the bug in consideration because it is very likely that the issue includes a description of the bug and explanatory information.

In practice, we extract the issue ID from each *bfc*'s commit message by the regular expressions such as "`^#\d+|[a-zA-Z]+-\d+$`" to establish the tracing from commit to issue. Inversely, in order to find commit ID from issue description, we extract the commit ID in issue description by the regular expression "`\b[0-9a-f]{7,40}\b`", or employ a GitHub API[1] to get the events which also record the related commit ID.

C. Locating Critical Changes

1) Identifying Bug-Fixing Location: We employ the state-of-the-art Probabilistic Delta Debugging (ProbDD) [9]

[1]https://docs.github.com/en/rest/issues/timeline?apiVersion=2022-11-28#about-timeline-events



(a) Bug-Fixing Changes (b) Possible Bug-Inducing Changes (c) Another Location of Possible Bug-Inducing Changes

Fig. 1: Multiple code change sets may be taken as the bug-inducing changes. This NPE bug in method `parseSubProjNum` is fixed by adding an `if` statement. Two possible sets of code changes could be considered as bug-inducing.

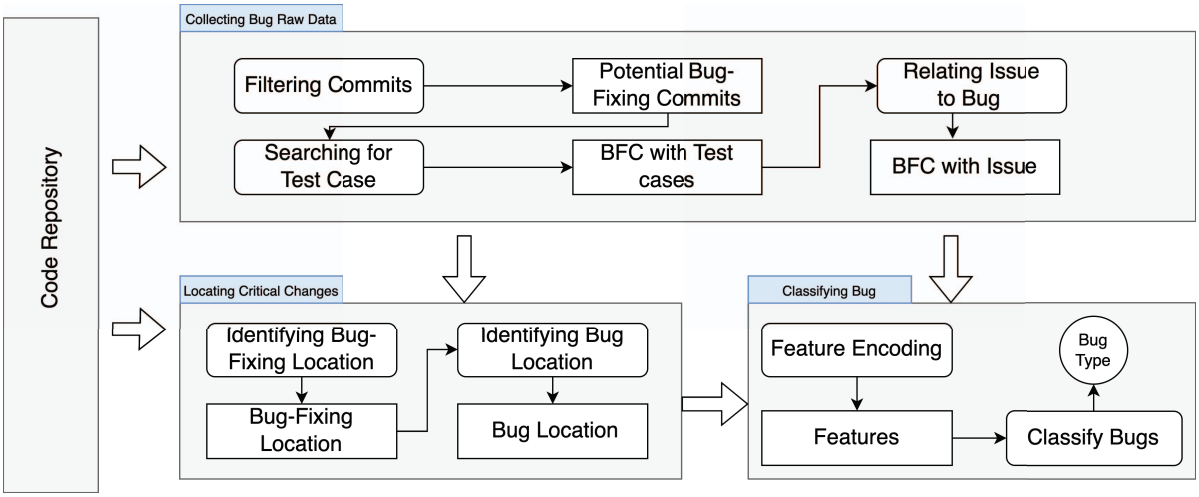


Fig. 2: Overview of BugMiner

technique to locate the critical bug-fixing changes *bfloc* in the *bfc*. ProbDD is designed for test case reduction tasks, which is suitable for finding the minimal set of source code that fixes the bug. Given all changed elements *diff* in *bfc*, our purpose is to search for $diff^* = \text{argmin}_{diff' \in diff} |diff'|, f(diff^*) \rightarrow T$, where the function f represents the act of reverting changes on *bfc* and executing the tests and T represents the outcome of the test, indicating a test failure. That is, if we revert all these changes back to the *bfc-1*, the program will exhibit the desired bug (i.e., test failure). Therefore, the task aims to search minimal set of changes that, when reverted, still cause the test to fail.

Note that, ProbDD assumes the independence between any code elements. Therefore, we use existing source code decomposition technique [10] to group the code elements in the changes into independent parts. Before the decomposition, we first use Diff/TS [11] to capture the code changes between the *bfc* and *bfc-1*. We choose Diff/TS because it enables fine-grained change analysis between two commits, keeping focus on the structural differences in the code. This facilitates the identification of the specific code changes that are critical for bug fixing in the *bfc*.

2) Identifying Bug Location: According to the definitions provided in Section II, we locate the buggy code (determining *biloc*) by searching for the bug-inducing commit (*bic*) and pinpointing bug-inducing changes in the *bic*, and ultimately finding the location of bug-inducing changes in the commit *bfc-1* to serve as the *biloc* .

The buggy code locating process is described in Algorithm 1. The process first searches for bug-inducing commit *bic* starting from the bug-fixing commit *bfc* backwards in the code repository by a *git-bisect*-based approach introduced by RegMiner [8]. This searching step (Line 1) basically runs the test for each commit examined and is able to correctly and efficiently handle the commits that cannot be correctly compiled (so-called "no feedback commits"). The output bug-inducing commit *bic* is the commit c that satisfies either of the two cases:

(1) The test fails at commit c and passes in commit $c-1$. In this case, we confirm that c is the commit that introduces the bug because, when we revert these changes on c, the bug will disappear. Now that we have c as the *bic*, we utilize Diff/TS to extract all the changes between *bic* and *bic-1* and apply the *decompose* [10] technique to partition these changes into independent parts before using ProbDD to locate the minimal bug-inducing changes.

(2) The test fails at commit c but a compilation error occurs at commit $c-1$. In this case, we are not sure whether c exactly introduces the bug. It is very likely that the bug already exists earlier in history because of some

Algorithm 1: Locating Buggy Code

Input : Bug-Fixing Commit bfc; Test Case $test$; Code Repository $repo$
Output: The location of buggy code $biloc$

```
   // Search for Bug-inducing Commit
1  bic = git_bisect_based_search(bfc, repo, test);
2  diff = [];
3  if bic passes and bic − 1 fails then
4  |  diff = AST_diff(bic, bic − 1);
   |  // Decompose the changes into distinct
   |     parts according to the relations
   |     between code change elements
5  |  grouped_diff = decompose(bic);
   |  // Identify critical bug-inducing changes
6  |  buggy_code = delta_debugging(bic, grouped_diff);
7  else if bic passes and bic − 1 is compile-error then
   |  // Estimate feature/function code elements
8  |  feature_codes = feature_estimate(bfc, test, repo);
   |  // Search for feature-introdcution commit
9  |  bic' =
   |     search_feature_intro_commit(bic, feature_codes);
   |  // Get all changes related to the bug
10 |  diff = AST_diff(bic, bic' − 1);
   |  // Identify critical bug-inducing changes
   |     by tracing analysis
11 |  buggy_code = backward_tracing(diff, test, bic);
12 else
   |  // This should be an invalid case.
13 |  return null
   // Align buggy_code in bic to bfc-1
14 bloc = code_alignment(bfc − 1, bic, buggy_code);
15 return biloc
```

newly added code representing new features. Therefore, we employ the feature estimation technique proposed by RegMiner [8] to identify which code elements may represent the feature/function in the test code. We then track the feature code elements from $c-1$ backwards until these code elements cannot be further tracked back at commit c'[2]. Commit c' is the commit that the feature does not exist. Then we consider the commit $c'+1$ as the commit that actually introduces the bug (the bic' at Line 9). Now we can extract all code changes between c (the original *bic* output at Line 1) and bic'-1 (the commit c' previously mentioned), with the help of Diff/TS, as an aggregated code change set related to bug-inducing. In order to identify the code changes that are directly related to the bug, we use code *instrumentation* techniques[3] to record the execution traces of the test in commit *bic*. These traces only include code changes between *bic* and bic'-1. We then conduct backward tracing by data flow and control flow analysis to find the specific code elements that are responsible for the erroneous state such as assertion failure or triggering exceptions. For instance, in the bug depicted in Fig.1a, Line 9 will throw a null pointer exception between the fix. If the changes in the bug-inducing commit are as in Fig.1b, then the method will be traced back to Line 17 of Fig.1b, where the statement itself is causing the bug to be executed due to the absence of null pointer detection. If the changes in the bug-inducing commit are as in Fig.1c, then the method will be traced back to Line 26 of Fig.1c, where the incorrect variable initialization within the if-else statement occurs.

Finally, we use a differencing technique [12] to align the identified critical bug-inducing changes, in either case described above, with the source code in the commit *bfc-1*, where the exact location of the bug is specified.

D. Classifying Bugs

Bugs can be categorized based on symptoms, root causes, or fix patterns [13], [14]. In this work, we focus on the root causes of bugs. We adopted the bug type taxonomy from Ni et al.'s work [15], as summarized in Table I.

In their work, Ni et al. [15] employed Tree-Based Convolutional Neural Network (TBCNN) [16] for bug classification. However, pre-trained representations of commits, such as CC2Vec [17] and FIRA [18], have demonstrated superior performance over CNN- or RNN-based models in tasks such as commit message generation and bug-fixing patch identification. This could be attributed to the models' ability to utilize large-scale unlabeled data through pre-training tasks, which is especially helpful when the labeled data is limited.

Therefore, we propose a *hy*brid *bu*g *c*lassification model HyBuC (pronounced like *high-buck*), which learns semantics from bug reports, commit messages, and bug-fixing/inducing locations through pre-trained models for bug classification.

1) Bug Type Labeling: Since Ni et al. [15] did not open-source their dataset, we manually annotated 1,618 bugs in total, which included all 809 bugs from Defects4J as of July 2023 and a subset of 809 bugs from the 2,082 bugs mined by BugMiner. Two of the authors independently labeled each bug, thoroughly examining the bug-fixing commit patch, commit message, and associated bug report during the labeling process. A third author was introduced to resolve disagreements. We utilized Cohen's Kappa coefficient to measure the agreement, and it achieved a value of 0.63, which indicates moderate agreement. We train the HyBuC model with these labeled data.

2) Classification Model: Fig. 3 illustrates the overall framework of HyBuC. HyBuC takes bug-fixing/inducing locations and source code, commit messages, and bug reports as the input. It utilizes feature encoding networks to encode each source of input separately and concatenate the encoded feature representation vectors as a whole embedding vector of the bug. Finally, a fully connected neural network (FCNN) is used to produce a bug type label as the bug classification result. For situations where the bug report is not available and/or the commit message is empty, we pad them to the fixed sequence size when batching.

Let's denote the input of HyBuC as $b = \langle br, commit\_msg, bfloc, biloc \rangle$, and its output as bug

[2]The track-back is implemented by an code-overlapping-based algorithm. If the overlapping code drops below a threshold ϱ (0.5 by default), we say the code elements representing the feature no longer exist

[3]https://docs.oracle.com/javase/8/docs/api/java/lang/instrument/Instrumentation.html

TABLE I: Bug Types and Descriptions

Cause category	Description
01 Function	The overall function cannot be implemented normally or there exists errors in the set of steps used to solve a particular problem or calculation, including errors in calculations. error implementations of algorithms.
02 Interface	Interaction issues within components, between components or with other systems, including mismatched calls and incorrect opening, reading, writing, or closing of files and databases.
03 Logic	The logic is incorrect, including incorrect branch statement, ignoring extreme values or situations(like null checks and boundary values), incorrect logic test condition or logical order, incorrect loop logic.
04 Computation	Using wrong operators, using incorrect operands(including misuse of operand in operational expression) or non-sufficient precision of data.
05 Assignment	Data initialization error, incorrect access to the field, inconsistent subroutine parameters, incorrect data range or type, incorrect input or output data, inspection issues of abnormal data or incorrect variable/constant name.
06 Others	All the bugs not in the above categories.

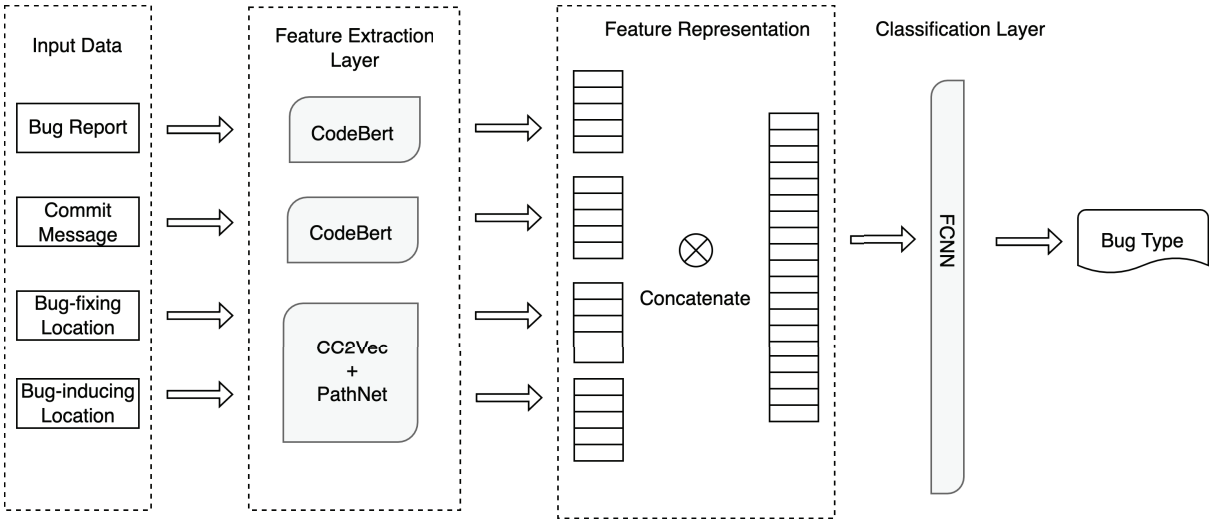


Fig. 3: Overview of the Hybrid Bug Classification Approach (HyBuC)

type l. Here, the bug report br and commit message consist of natural language (NL) tokens. The bug-fixing location $bfloc$ and bug-introducing location $biloc$ can be represented as $\langle P_a, P_r \rangle$, where $p_a \in P_a$ and $p_r \in P_r$ represent the added and removed patches. Each p_a and p_r consist of n_{p_a} and n_{p_r} code tokens, respectively. The bug type $l \in L$, where L is the set of bug types mentioned in Table I.

We employ CodeBERT [19], a widely-used, transformer-based pre-trained model for code-related tasks, to encode br and $commit\_msg$, generating the pooled output as embedding $e_{br}, e_{commit\_msg} \in \mathbb{R}^{1\times D}$, where D represents the dimension of the representation. To enable batch training and inference, each input instance is padded or truncated to the same length.

For each patch p in the $bfloc$ and $biloc$, we utilize both PatchNet [20] and CC2Vec [17] to encode the patch into a vector $e_{\text{code\_bfloc}}$ or $e_{\text{code\_biloc}}$, respectively. PatchNet is a 3D-CNN model that extracts code change representations and has demonstrated effectiveness in commit-related tasks [20]. CC2Vec is a pre-trained model based on the hierarchical attention network designed for code commit representation. It has been proven to enhance the performance of PatchNet when used together [17]. After concatenating all the *added* patches in $bfloc$, we encode them using PatchNet and CCVec[4] to obtain $e_{\text{bfloc\_add\_cc}}, e_{\text{bfloc\_add\_patch}} \in \mathbb{R}^{1\times D}$. Similarly, we obtain $e_{\text{bfloc\_remove\_cc}}, e_{\text{bfloc\_remove\_patch}}$ for the *removed* patches. Then, we set $e_{\text{code\_bfloc}} = [e_{\text{bfloc\_add\_patch}}; e_{\text{bfloc\_remove\_patch}}; e_{\text{bfloc\_add\_cc}}; e_{\text{bfloc\_remove\_cc}}] \in \mathbb{R}^{4\times D}$. Similarly, we get $e_{\text{code\_biloc}} \in \mathbb{R}^{4\times D}$

Next, we concatenate $e_{\text{br}}, e_{\text{msg}}, e_{\text{code\_bfloc}}, e_{\text{code\_biloc}}$ to obtain $e_b \in \mathbb{R}^{10\times D}$. After passing it through FCNN and the softmax normalization function, we obtain the probabilities $\in \mathbb{R}^{1\times L}$ for each label. We adopt the focal loss function [23] to alleviate the bug type imbalance problem.

[4]Since CC2Vec is pre-trained on a dataset based on the C programming language, and the bugs in Defects4j and BugMiner are in Java, we had to continue pre-train CC2Vec with 1,670 samples from Defects4J, BugMiner, and other widely-used datasets [21], [22] for commit message generation tasks. The pre-training task is the same as the original CC2Vec, which is to generate commit messages from code commits. In total, 92,331 instances of data are used for pre-training.

IV. Experiment

In this section, we report the experiments conducted to address the following research questions:

RQ1: What is the efficiency of BugMiner to collect bugs compared to existing approaches?

RQ2: What is the efficiency and effectiveness of BugMiner to locating bug-fixing code compared to existing approaches?

RQ3: What is the efficiency and effectiveness of BugMiner to locating bug-inducing code?

RQ4: What is the accuracy of BugMiner in determining bug types compared to existing approaches?

A. Experiment Setup

1) RQ1: Efficiency of Collecting Bugs: To conduct a comparative experiment, we used BugMiner and BugBuilder with the same set of project candidates, consisting of 100 open-source projects. We measured the number of bugs collected by each tool within a fixed time period and also recorded the total number of bugs collected after consuming all the projects without time restrictions.

2) RQ2: Efficiency and Effectiveness of Locating Bug-Fixing Code: To address RQ2, we conducted experiments to compare BugMiner and BugBuilder on the bug-fixing location identification task. For the task, we also use 809 bugs selected from Defects4J, which are manually validated, as the ground-truth benchmark for evaluation.

We compare the effectiveness of BugMiner and BugBuilder by contrasting the token-level sizes of their results with the ground-truth critical changes in Defects4J. If the result exceeds the size of the actual critical change, we consider it an erroneous outcome case, which affects the precision of each method. Furthermore, if a method fails to provide any result, we consider it a failure case, which affects the precision and recall of each method. Additionally, we compare the efficiency of the two methods by contrasting their average execution times.

3) RQ3: Efficiency and Effectiveness of Locating Bug-Inducing Code: To the best of our knowledge, there is no existing approach that is comparable in locating bug-inducing code. Therefore, we conducted an open-world experiment to address RQ3. In the 100 projects from RQ1, BugMiner was asked to locate bug location on the bug collection results. We evaluate the efficiency by calculating the average time taken for bug localization across all bugs. Subsequently, we evaluate the effectiveness by sampling the results for human validation of their accuracy. Two graduate students with over 3 years of Java development experiences were invited to participate in the validation process. They were asked to answer the following questions for each sampled bug:

- What is the root cause of the bug?
- Does the BugMiner's output help you fix the bug?

They were asked to sample 50 bugs for validation. The first question aimed to help them understand the bugs, while the second question aimed to validate the accuracy of the method's results. After they completed the validation of all cases, we introduced a third person, one of the authors of this paper, to facilitate a discussion to resolve any conflicts that arose.

4) RQ4: Accuracy of Bug Classification: To address RQ4, 1,618 bugs, as described in Section III.D, sampled and manually annotated from Defects4J and BugMiner, are used as our dataset. We divided the dataset into training and testing sets in an 8:2 ratio. We employed TBCNN [15] as the baseline model, which extracts fix trees from bug fixing commits as features. We followed the training settings as presented in the paper [15], including the optimizer, dropout, etc. For evaluation, we calculated common metrics in classification tasks, namely macro-precision, macro-recall, and macro-F1-score.

Our experiments were performed on a Linux server with a 10-core 40-thread Intel(R) Core(TM) i9-10980XE CPU @ 3.00GHz, an NVIDIA GeForce RTX 3090 GPU, and the Ubuntu Linux 22.04 operating system.

B. Results

1) RQ1: Efficiency and Effectiveness of Collecting Bugs: The results are shown in Table II. Firstly, within the limited time(40 hours), BugMiner achieved high efficiency. BugMiner collected a total of 432 defects, which is 20% more compared to BugBuilder. Secondly, when the time was unlimited, BugMiner performed even better effectively in bug collection. After running for about 8 weeks, BugMiner has collected a total of 2,082 defects across 100 projects, which is 67% more compared to BugBuilder.

2) RQ2: Efficiency and Effectiveness of Locating Bug-Fixing Code: We discovered that BugMiner achieved high accuracy, recall rates and efficiency, as shown in Table III. Firstly, BugMiner generated 658 bug-fixing locations out of 809 bugs, resulting 81.3% recall, 41.3% higher than BugBuilder. For the 658 generated bug-fixing locations, 464 of them are complete and concise, which means BugMiner achieved the precision of 70.5%. The 464 accurate bug-fixing locations, out of total 809 bugs, bring a high accuracy of 57.4%, much higher (19.3% improvment) than BugBuilder's 38.1% (308/809). Notably, for 56 bugs, BugMiner provided even more concise bug fixing locations. However, the precision of BugMiner is 25.6% lower than BugBuilder because BugBuilder tries to enumerate all of the possible bug-fixing locations in practice. The high complexity of this algorithm will restrict the capability of generating correct bug-fixing locations successfully. Finally, the average time spent by BugMiner for generating each bug-fixing location is 0.3 hours, which is 0.2 hours less than BugBuilder, showing that BugMiner is more efficient.

3) RQ3: Efficiency and Effectiveness of Locating Bug-Inducing Code: The results are shown in Table IV. We discovered that BugMiner also achieved high accuracy, recall and efficiency. BugMiner generated 1667 bug-inducing locations out of 2,082 bugs collected with the recall rate of 80.1%. After sampling 50 instances and being validated manually, we found that, for 42 out of the 50 instances,

TABLE II: Performance of Bug Collection

Technique	Bugs Collected Within Fixed Time(40h)	Bug Collection Without Time Restriction
BugMiner	432	2,082 (157h)
BugBuilder	360	1,246 (142h)

TABLE III: Performance of Locating Bug-Fixing Code

Technique	Bug-fixing Commits	Identified Bug-fixing Locations	Accurate Locations	Precision	Accuracy	Recall	Average Time
BugMiner	809	658	464	70.5%	57.4%	81.3%	0.3h
BugBuilder	809	324	308	95.1%	38.1%	40.0%	0.5h

TABLE IV: Performance of Locating Bug-Inducing Code

Technique	Generated Bug-inducing Locations	Accurate Locations	Accuracy	Recall	Average Time
BugMiner	1667 (2,082)	42(50)	84.0%	80.1%	0.4h

TABLE V: Performance of Bug Classification Models

Datasets	TBCNN			HyBuC		
	macro-P	macro-R	macro-F1	macro-P	macro-R	macro-F1
Defects4J	0.426	0.362	0.391	0.612	0.556	0.583
BugMiner	0.432	0.351	0.387	0.587	0.510	0.546
Avg	0.428	0.357	0.389	0.600	0.533	0.565

BugMiner generated accurate bug-inducing locations, meaning that the accuracy of BugMiner for sample set is 84%. Notably, the efficiency is also acceptable. The time spent for each bug-inducing location is 0.4 hours, which is longer than identifying bug-fixing locations due to the complexity of this task.

4) RQ4: Accuracy of Bug Classification: Overall, both TBCNN and HyBuC exhibit relatively low classification accuracy, as shown in Table V. This is primarily due to the severe class imbalance in our bug dataset, where most bugs belong to the Logic and Function categories, and other categories contain very few bugs. This leads to the model's inability to classify these less frequent categories effectively. Compared to TBCNN, HyBuC improves the average precision and recall by 0.172 and 0.176, respectively. However, HyBuC performs less effectively on the BugMiner than on the Defects4J, since some bugs in BugMiner lack associated bug reports whereas all bugs in Defects4J include bug reports.

V. Discussion

Our experiment demonstrates that BugMiner can effectively construct a runnable bug repository from code evolution history, and extract critical information such as bug locations and bug types. This research can serves as a fundamental advancement in facilitating data-driven software engineering tasks. However, the quality and diversity of the dataset remain significant concerns.

On one hand, ensuring data quality still relies on human involvement. However, accurately annotating and correcting bug data is not as straightforward as in image or natural language domains, and it requires a high level of expertise from the annotators. On the other hand, achieving diversity in bug types relies on community contributions. Building a community and fostering the sharing of bugs across different domains pose significant challenges.

Therefore, we call for and foresee the establishment of a crowdsourcing platform and deem that such a platform should have the following features or address the following issues:

- It should have interactive algorithms and models that recommend critical bug location information, bug types, etc., while suggesting other potentially useful information that can help users verify these details. It should also allow users to check and provide feedback on each piece of data, using this feedback to enable the model to understand how to help users validate more quickly and provide more accurate results
- It should propose a reward mechanism to stimulate community bug contributions. This mechanism needs to ensure the accuracy of user-contributed data, and then reward users to continue obtaining the required data on the platform.
- It is necessary to establish a collaborative mechanism among users to ensure data quality and to explore further possibilities for improvement.

VI. Related Work

Bug Datasets Construction. Bug datasets play a pivotal role in providing empirical and experimental foundations for various software engineering and programming language tasks, encompassing software testing, debugging, fault localization, and repair. The research community has witnessed the proposal of numerous bug datasets, each contributing to this domain. Noteworthy among them is the SIR dataset, introduced by Do et al. [24], which stands as a pioneering work in bug dataset construction. In addition, datasets like Marmoset [25], QuixBugs [26], IntroClass [27], Codeflaws [28], and others have emerged from programming assignments and competitions. Open source projects have also played a significant role in contributing to the construction of bug datasets, with examples such as DBG-

Bench [29], Defects4j [4], BugsJS [30], Bugs.jar [31], and more. Moreover, bug datasets have been constructed utilizing runtime continuous integration scenarios like BEARS [1] and BugSwarm [2]. It is worth mentioning the notable contribution by Marcel et al. [32], who proposed Corebench, the largest regression dataset comprising 70 C/C++ regressions. However, these bug datasets are primarily constructed manually, which poses limitations in terms of scalability and representativeness. To address this concern, Dallmeier et al. [33] took the initiative to construct bug datasets in a semi-automatic manner by analyzing bug issues and relevant commits. Zhao et al. [34] further replicated bugs based on Android bug reports, adding to the diversity of bug datasets. Recently, Jiang et al. [3] introduced BugBuilder, a tool designed specifically for constructing datasets by isolating bug-relevant changes. Song et al.introduced RegMiner, an approach to designed for auto constructing regression datasets from code evolution history.

Bug Classification. Classifying bug categories has many benefits, such as monitoring frequently occurring bug types in a project to provide corresponding solutions, conducting targeted studies on the characteristics of different bug types, and proposing appropriate repair methods [14], [13], [35]. Additionally, it allows for evaluating the effectiveness of automated bug localization and bug fixing techniques across different bug categories [36], [37], [38], [39], [40].

Most existing approaches classified bugs based on bug reports [41], [42], [43]. Zhou et al. [41] proposed a semi-supervised Named Entity Recognition (NER) method classify bug reports, and defined a set of features for model training. Catolino et al. [42] compared three methods of extracting features from bug report summaries: TF-IDF[44], word2vec[45], and doc2vec[46]. They also examined four classifiers, namely Naive Bayes [47], Support Vector Machines (SVM) [48], Logistic Regression [49], and Random Forest [50]. Ultimately, they found that the combination of TF-IDF and SVM resulted in the best bug classification performance. Only few works utilized bug-fixing commit perform bug classification [15], [51]. Ni et al. [15] proposed a taxonomy of bug types based on commit patches, and utilize Tree-Based Convolutional Neural Network (TBCNN) [16] for bug classification. Thung et al. [51] extracted manually designed features from bug report and bug-fixing commit, and adopted SVM to classify them.

In conclusion, current bug classification methods do not fully utilize the multi-modal information available in bug reports, commit messages, bug-fixing commits and bug-introducing commits. Additionally, these methods rely either on manually designed rules or use CNN and LSTM for feature extraction, without leveraging state-of-the-art pre-trained models like CodeBERT [19] and CC2Vec [17].

VII. Conclusion and Future Work

In this work, we have introduced BugMiner, which automatically constructs bug dataset from code evolution history and provides critical information such as bug locations, types, and descriptions. We have further defined bug location and presented effective solutions for it. Our experiments have demonstrated that BugMiner achieves acceptable accuracy and recall rates. Moving forward, we aim to enhance the efficacy and performance of Probabilistic Delta Debugging, proposing enhanced models to grant the method the ability to detect and rectify erroneous results in the decompose part.

Furthermore, this paper presents a preliminary exploration of bug classification, and identifies potential improvements in future research in terms of data and model aspects. Regarding data, the current manually annotated dataset is rather limited and unevenly distributed across bug types. As a solution, we will consider employing semi-supervised learning methods [52] to expand the training set and data augmentation techniques [53] to enhance underrepresented bug categories. As for the model, we will investigate attention-based approaches [54] to better integrate features, and propose new pre-training models specific to bug scenarios.

VIII. Acknowledgment

The authors would like to thank deeply the anonymous referees for their valuable comments and helpful suggestions. This work was supported in part by National Natural Science Foundation of China (62172099).

References

[1] F. Madeiral, S. Urli, M. de Almeida Maia, and M. Monperrus, "BEARS: an extensible java bug benchmark for automatic program repair studies," in *26th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2019, Hangzhou, China, February 24-27, 2019* (X. Wang, D. Lo, and E. Shihab, eds.), pp. 468–478, IEEE, 2019.

[2] D. A. Tomassi, N. Dmeiri, Y. Wang, A. Bhowmick, Y. Liu, P. T. Devanbu, B. Vasilescu, and C. Rubio-González, "Bugswarm: mining and continuously growing a dataset of reproducible failures and fixes," in *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019* (J. M. Atlee, T. Bultan, and J. Whittle, eds.), pp. 339–349, IEEE / ACM, 2019.

[3] Y. Jiang, H. Liu, N. Niu, L. Zhang, and Y. Hu, "Extracting concise bug-fixing patches from human-written patches in version control systems," in *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021*, pp. 686–698, IEEE, 2021.

[4] R. Just, D. Jalali, and M. D. Ernst, "Defects4j: a database of existing faults to enable controlled testing studies for java programs," in *International Symposium on Software Testing and Analysis, ISSTA '14, San Jose, CA, USA - July 21 - 26, 2014* (C. S. Pasareanu and D. Marinov, eds.), pp. 437–440, ACM, 2014.

[5] A. Mockus and L. G. Votta, "Identifying reasons for software changes using historic databases," in *2000 International Conference on Software Maintenance, ICSM 2000, San Jose, California, USA, October 11-14, 2000*, pp. 120–130, IEEE Computer Society, 2000.

[6] M. Mondal, C. K. Roy, and K. A. Schneider, "Identifying code clones having high possibilities of containing bugs," in *Proceedings of the 25th International Conference on Program Comprehension, ICPC 2017, Buenos Aires, Argentina, May 22-23, 2017* (G. Scanniello, D. Lo, and A. Serebrenik, eds.), pp. 99–109, IEEE Computer Society, 2017.

[7] B. Hu, Y. Wu, X. Peng, J. Sun, N. Zhan, and J. Wu, "Assessing code clone harmfulness: Indicators, factors, and counter measures," in *28th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2021, Honolulu, HI, USA, March 9-12, 2021*, pp. 225–236, IEEE, 2021.

[8] X. Song, Y. Lin, S. H. Ng, Y. Wu, X. Peng, J. S. Dong, and H. Mei, "Regminer: towards constructing a large regression dataset from code evolution history," in *ISSTA '22: 31st ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, South Korea, July 18 - 22, 2022* (S. Ryu and Y. Smaragdakis, eds.), pp. 314–326, ACM, 2022.

[9] G. Wang, R. Shen, J. Chen, Y. Xiong, and L. Zhang, "Probabilistic delta debugging," in *ESEC/FSE '21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens, Greece, August 23-28, 2021* (D. Spinellis, G. Gousios, M. Chechik, and M. D. Penta, eds.), pp. 881–892, ACM, 2021.

[10] M. Hashimoto, A. Mori, and T. Izumida, "Automated patch extraction via syntax- and semantics-aware delta debugging on source code changes," in *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018* (G. T. Leavens, A. Garcia, and C. S. Pasareanu, eds.), pp. 598–609, ACM, 2018.

[11] M. Hashimoto and A. Mori, "Diff/ts: A tool for fine-grained structural change analysis," in *WCRE 2008, Proceedings of the 15th Working Conference on Reverse Engineering, Antwerp, Belgium, October 15-18, 2008* (A. E. Hassan, A. Zaidman, and M. D. Penta, eds.), pp. 279–288, IEEE Computer Society, 2008.

[12] Z. Xing and E. Stroulia, "Umldiff: an algorithm for object-oriented design differencing," in *20th IEEE/ACM International Conference on Automated Software Engineering (ASE 2005), November 7-11, 2005, Long Beach, CA, USA* (D. F. Redmiles, T. Ellman, and A. Zisman, eds.), pp. 54–65, ACM, 2005.

[13] G. Jin, L. Song, X. Shi, J. Scherpelz, and S. Lu, "Understanding and detecting real-world performance bugs," *ACM SIGPLAN Notices*, vol. 47, no. 6, pp. 77–88, 2012.

[14] J. Cao, B. Chen, C. Sun, L. Hu, S. Wu, and X. Peng, "Understanding performance problems in deep learning systems," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 357–369, 2022.

[15] Z. Ni, B. Li, X. Sun, T. Chen, B. Tang, and X. Shi, "Analyzing bug fix for automatic bug cause classification," *Journal of Systems and Software*, vol. 163, p. 110538, 2020.

[16] L. Mou, G. Li, L. Zhang, T. Wang, and Z. Jin, "Convolutional neural networks over tree structures for programming language processing," in *Proceedings of the AAAI conference on artificial intelligence*, vol. 30, 2016.

[17] T. Hoang, H. J. Kang, D. Lo, and J. Lawall, "Cc2vec: Distributed representations of code changes," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pp. 518–529, 2020.

[18] J. Dong, Y. Lou, Q. Zhu, Z. Sun, Z. Li, W. Zhang, and D. Hao, "Fira: fine-grained graph-based code change representation for automated commit message generation," in *Proceedings of the 44th International Conference on Software Engineering*, pp. 970–981, 2022.

[19] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, *et al.*, "Codebert: A pre-trained model for programming and natural languages," *arXiv preprint arXiv:2002.08155*, 2020.

[20] T. Hoang, J. Lawall, Y. Tian, R. J. Oentaryo, and D. Lo, "Patchnet: Hierarchical deep learning-based stable patch identification for the linux kernel," *IEEE Transactions on Software Engineering*, vol. 47, no. 11, pp. 2471–2486, 2019.

[21] S. Jiang and C. McMillan, "Towards automatic generation of short summaries of commits," in *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*, pp. 320–323, IEEE, 2017.

[22] S. Xu, Y. Yao, F. Xu, T. Gu, H. Tong, and J. Lu, "Commit message generation for source code changes," in *IJCAI*, 2019.

[23] T.-Y. Lin, P. Goyal, R. Girshick, K. He, and P. Dollár, "Focal loss for dense object detection," in *Proceedings of the IEEE international conference on computer vision*, pp. 2980–2988, 2017.

[24] H. Do, S. G. Elbaum, and G. Rothermel, "Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact," *Empir. Softw. Eng.*, vol. 10, no. 4, pp. 405–435, 2005.

[25] J. Spacco, J. Strecker, D. Hovemeyer, and W. Pugh, "Software repository mining with marmoset: An automated programming project snapshot and testing system," *SIGSOFT Softw. Eng. Notes*, vol. 30, p. 1–5, may 2005.

[26] D. Lin, J. Koppel, A. Chen, and A. Solar-Lezama, "Quixbugs: a multi-lingual program repair benchmark set based on the quixey challenge," in *Proceedings Companion of the 2017 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity, SPLASH 2017, Vancouver, BC, Canada, October 23 - 27, 2017* (G. C. Murphy, ed.), pp. 55–56, ACM, 2017.

[27] C. L. Goues, N. J. Holtschulte, E. K. Smith, Y. Brun, P. T. Devanbu, S. Forrest, and W. Weimer, "The manybugs and introclass benchmarks for automated repair of C programs," *IEEE Trans. Software Eng.*, vol. 41, no. 12, pp. 1236–1256, 2015.

[28] S. H. Tan, J. Yi, Yulis, S. Mechtaev, and A. Roychoudhury, "Codeflaws: a programming competition benchmark for evaluating automated program repair tools," in *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017 - Companion Volume* (S. Uchitel, A. Orso, and M. P. Robillard, eds.), pp. 180–182, IEEE Computer Society, 2017.

[29] M. Böhme, E. O. Soremekun, S. Chattopadhyay, E. Ugherughe, and A. Zeller, "Where is the bug and how is it fixed? an experiment with practitioners," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017* (E. Bodden, W. Schäfer, A. van Deursen, and A. Zisman, eds.), pp. 117–128, ACM, 2017.

[30] P. Gyimesi, B. Vancsics, A. Stocco, D. Mazinanian, Á. Beszédes, R. Ferenc, and A. Mesbah, "Bugsjs: a benchmark of javascript bugs," in *12th IEEE Conference on Software Testing, Validation and Verification, ICST 2019, Xi'an, China, April 22-27, 2019*, pp. 90–101, IEEE, 2019.

[31] R. K. Saha, Y. Lyu, W. Lam, H. Yoshida, and M. R. Prasad, "Bugs.jar: a large-scale, diverse dataset of real-world java bugs," in *Proceedings of the 15th International Conference on Mining Software Repositories, MSR 2018, Gothenburg, Sweden, May 28-29, 2018* (A. Zaidman, Y. Kamei, and E. Hill, eds.), pp. 10–13, ACM, 2018.

[32] M. Böhme and A. Roychoudhury, "Corebench: studying complexity of regression errors," in *International Symposium on Software Testing and Analysis, ISSTA '14, San Jose, CA, USA - July 21 - 26, 2014* (C. S. Pasareanu and D. Marinov, eds.), pp. 105–115, ACM, 2014.

[33] V. Dallmeier and T. Zimmermann, "Extraction of bug localization benchmarks from history," in *22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007), November 5-9, 2007, Atlanta, Georgia, USA* (R. E. K. Stirewalt, A. Egyed, and B. Fischer, eds.), pp. 433–436, ACM, 2007.

[34] Y. Zhao, K. Miller, T. Yu, W. Zheng, and M. Pu, "Automatically extracting bug reproducing steps from android bug reports," in *Reuse in the Big Data Era - 18th International Conference on Software and Systems Reuse, ICSR 2019, Cincinnati, OH, USA, June 26-28, 2019, Proceedings* (X. Peng, A. Ampatzoglou, and T. Bhowmik, eds.), vol. 11602 of *Lecture Notes in Computer Science*, pp. 100–111, Springer, 2019.

[35] Y. Zhang, Y. Chen, S.-C. Cheung, Y. Xiong, and L. Zhang, "An empirical study on tensorflow program bugs," in *Proceedings of the 27th ACM SIGSOFT international symposium on software testing and analysis*, pp. 129–140, 2018.

[36] M. Wen, R. Wu, and S.-C. Cheung, "Locus: Locating bugs from software changes," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pp. 262–273, 2016.

[37] J. Cao, S. Yang, W. Jiang, H. Zeng, B. Shen, and H. Zhong, "Bugpecker: Locating faulty methods with deep learning on revision graphs," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, pp. 1214–1218, 2020.

[38] S. Yang, J. Cao, H. Zeng, B. Shen, and H. Zhong, "Locating faulty methods with a mixed rnn and attention model," in *2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC)*, pp. 207–218, IEEE, 2021.

[39] J. Bader, A. Scott, M. Pradel, and S. Chandra, "Getafix: Learning to fix bugs automatically," *Proceedings of the ACM on Programming Languages*, vol. 3, no. OOPSLA, pp. 1–27, 2019.

[40] E. Dinella, H. Dai, Z. Li, M. Naik, L. Song, and K. Wang, "Hoppity: Learning graph transformations to detect and fix bugs in programs," in *International Conference on Learning Representations (ICLR)*, 2020.

[41] C. Zhou, B. Li, X. Sun, and H. Guo, "Recognizing software bug-specific named entity in software bug repository," in *Proceedings of the 26th Conference on Program Comprehension*, pp. 108–119, 2018.

[42] G. Catolino, F. Palomba, A. Zaidman, and F. Ferrucci, "Not all bugs are the same: Understanding, characterizing, and classifying bug types," *Journal of Systems and Software*, vol. 152, pp. 165–181, 2019.
[43] K. Herzig, S. Just, and A. Zeller, "It's not a bug, it's a feature: how misclassification impacts bug prediction," in *2013 35th international conference on software engineering (ICSE)*, pp. 392–401, IEEE, 2013.
[44] G. Salton and C. Buckley, "Term-weighting approaches in automatic text retrieval," *Inf. Process. Manag.*, vol. 24, no. 5, pp. 513–523, 1988.
[45] Y. Goldberg and O. Levy, "word2vec explained: deriving mikolov et al.'s negative-sampling word-embedding method," *CoRR*, vol. abs/1402.3722, 2014.
[46] Q. Le and T. Mikolov, "Distributed representations of sentences and documents," in *Proceedings of the 31st International Conference on International Conference on Machine Learning - Volume 32*, ICML'14, p. II–1188–II–1196, JMLR.org, 2014.
[47] G. I. Webb, E. Keogh, and R. Miikkulainen, "Naïve bayes.," *Encyclopedia of machine learning*, vol. 15, no. 1, pp. 713–714, 2010.
[48] D. A. Pisner and D. M. Schnyer, "Support vector machine," in *Machine learning*, pp. 101–121, Elsevier, 2020.
[49] M. P. LaValley, "Logistic regression," *Circulation*, vol. 117, no. 18, pp. 2395–2399, 2008.
[50] S. J. Rigatti, "Random forest," *Journal of Insurance Medicine*, vol. 47, no. 1, pp. 31–39, 2017.
[51] F. Thung, D. Lo, and L. Jiang, "Automatic defect categorization," in *2012 19th working conference on reverse engineering*, pp. 205–214, IEEE, 2012.
[52] A. Ratner, S. H. Bach, H. Ehrenberg, J. Fries, S. Wu, and C. Ré, "Snorkel: Rapid training data creation with weak supervision," in *Proceedings of the VLDB Endowment. International Conference on Very Large Data Bases*, vol. 11, p. 269, NIH Public Access, 2017.
[53] H. Xie, Y. Lei, M. Yan, Y. Yu, X. Xia, and X. Mao, "A universal data augmentation approach for fault localization," in *Proceedings of the 44th International Conference on Software Engineering*, pp. 48–60, 2022.
[54] Y. Dai, F. Gieseke, S. Oehmcke, Y. Wu, and K. Barnard, "Attentional feature fusion," in *Proceedings of the IEEE/CVF winter conference on applications of computer vision*, pp. 3560–3569, 2021.