CS527-Team 9: Evaluating Testing and Debugging Tools on Real-world Bugs

Shreya Ujwal Chaudhary
University of Illinois Urbana-Champaign
suc2@illinois.edu

Shrushti Santosh Jagtap
University of Illinois Urbana-Champaign
sjagtap3@illinois.edu

Sejal Sanjay Pekam
University of Illinois Urbana-Champaign
spekam2@illinois.edu

Andres Suazo
University of Illinois Urbana-Champaign
suazo2@illinois.edu

ABSTRACT

This study investigates the effectiveness of automated bug localization techniques on real-world software bugs. We leverage Randoop and EvoSuite to generate test cases targeting bug fixes. A suspiciousness score is calculated to rank code elements based on their likelihood of causing test failures. We evaluate our approach using Average Rank (AR) and First Rank (FR) metrics on datasets including Defects4J, Bears, BugSwarm, and QuixBugs. This research provides valuable insights into the potential of automated techniques for pinpointing faulty code segments in real-world projects.

KEYWORDS

Software Testing, Software Debugging, Real-world Bugs, QuixBugs, Defects4J, Bears, BugSwarm, Localization, Benchmarking

ACM Reference Format:
Shreya Ujwal Chaudhary, Shrushti Santosh Jagtap, Sejal Sanjay Pekam, and Andres Suazo. 2024. CS527-Team 9: Evaluating Testing and Debugging Tools on Real-world Bugs. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 TEAM INFORMATION

Link to the team's GitHub Repo:
`https://github.com/ShreyaChaudhary1211/CS527-Project.git`
Link to the team's Overleaf draft:
`https://www.overleaf.com/1889112563sctsshxzcnwy#4f0263`

2 INTRODUCTION

As time has passed, software has transitioned from being some innovation used and trusted by a handful of enthusiast to a crucial part of the day to day of even the most secluded human. With this increased reliance on digital solutions, developers now face the burden of ensuring their creations are reliable, consistent, and completely dependable. With this in mind, software bugs, specially those who are the source of crucial errors, have become extremely dangerous, specially in high stake environments such as finance and healthcare. Thus, the ability to quickly and accurately localize bugs is now more paramount than ever. Not only does effective bug localization minimize potential risks of vulnerabilities, malfunctions and user experience, but also reduce the downtime and man-hours incurred in finding and patching buggy code.

In our research we aim to evaluate thoroughly the effectiveness of popular bug localization tools in real-world datasets. We will introduce two main tools, Randoop and Evosuite, which were designed and developed to generate test cases that could expose bugs in existing codebases. Each tool reaches said goal in its own way. Randoop focuses on generating error revealing unit tests for the targeted codebase. On the other hand, Evosuite creates entire test suites aimed at satisfying a certain coverage criterion. By adding and modifying small sets of assertion to the result being produced, it further help developers pinpoint possible deviations from intended behaviors. Both tools propose the idea of automating testing as a key driver of reliable software development and what should be a recommended, if not mandatory, step in the software develoment cycle.

To determine their adequacy and their relevant impact, we ran the tools across various datasets, including Defects4J, Bears, BugSwarm, and QuixBugs in order to generate a auxiliary set of tests. For each of the generated tests we calculated their likelihood of causing failures (suspiciousness score) as well as ranked how accurate they were at identifying the root causes of bugs. With the gathered metrics, we then evaluated the strengths and weaknesses of each tool and how viable we deem them to be in large scale systems.

3 EMPIRICAL SETUP

In this section, we will discuss the tools we used and how did we set them up. We will briefly go over the bug datasets that we used, the testing techniques and the bug localization methodology as well.

3.1 Bug Datasets

For this project, we will be using five datasets namely Defects4J [1], BugSwarm [2], Bears [3], ManySStuBs4J [4] and QuixBugs [5]. Lets explore the datasets one by one:

3.1.1 Defects4J.

Defects4J is a collection of bugs characterized for being reproducible, isolated and relevant to current software design and development best practices [1]. Bugs are collected in large by repository contributors manually proposing and verifying them. While this makes the process slower, each bug is compatible and testable, as well as allowing for the collection of granular data for each bug such as trigger tests, fix commits, reports etc. In all, the dataset has collected 835 bugs (29 additional deprecated bugs).

We extracted all the bugs from the dataset. Some samples of the bugs we encountered are:

(1) A bug from project *jfreechart*; among the 64 tests in total, 63 passed and 1 failed. As you can see in the code below, the bug fix was to change the conditional operators.

```
- if (dataset != null) {
+ if (dataset == null) {
```

(2) A bug from project *commons-math*; for which one test ran and failed. As you can see in the code below, the bug fix was datatype conversion.

```
- return getSampleSize() * (getNumberOfSuccesses()
    / (double) getPopulationSize());
+ return double(getSampleSize() * (
      getNumberOfSuccesses())/ (double)
      getPopulationSize());
```

3.1.2 BugSwarm.

BugSwarm dataset consists of a collection of software bugs and information about them from different open source software projects. [2] It consists of bugs, lines of code changed, commits made and other github related information. It is usually collected using a tool that crawls over the repositories of projects and extracts bugs and information like the time, what the bug is about, how the code was changed and fixed, etc. There are a total of 223 projects, of which 70 are in python and 150 are in Java. There are a total of 2917 bugs. There are 934 buggy commits in Python projects and 1983 in Java projects.

The whole dataset is available at `bugswarm.org`. The website describes the bug data comprising of which repository it belongs to, job id, build id, no. of tests passed, no. of tests failed and a link to the difference between the buggy and correct code. It also provides a JSON file comprising of all the bugs along with the details. We wrote a script that refers to the SHA's in the json file and the diff urls to extract the files that we require.

We extracted all the bugs from the dataset. Some samples of the bugs we encountered are:

(1) A bug from project *verktyg*; for which 247 tests ran in total out of which 242 passed and 5 failed. As you can see in the code below, the bug fix comprised of some code addition and code deletion.

```
    +   qs = wsgi_decoding_dance(
    +           self.environ.get('QUERY_STRING', ''),
    +           charset=self.url_charset,
    +               errors=self.encoding_errors,
            )
      -         wsgi_get_bytes(self.environ.get('
      QUERY_STRING', '')),
      -         encoding=self.url_charset,
                 errors=self.encoding_errors,
      -         qs, encoding=self.url_charset,
      -         errors=self.encoding_errors,
```

(2) A bug from project *ontop*; for which 0 tests ran in total out of which 0 passed and 0 failed. As you can see in the code below, the bug fix comprised of a version change.

```
        -  <version>3.0.0</version>
        +  <version>1.7.1</version>
```

3.1.3 Bears.

Bears provides a collection of bugs aimed to serve as a resource for researchers to evaluate and develop program repair tools [3]. The dataset itself contains 118 bugs, collected from open source java projects found on GitHub. While manual verification is still needed, automation is leveraged to identify bugs through various methods such as scanning commit messages and issues.

Once bugs are identified, Travis CI builds are gathered for the pre and post bug fix code states. When all data for a bug is gathered, a branch is created for it and each commit is mapped to the different states of the bug (pre-fix, failed-tests, post-fix, post-fix-eith-metadad). There are total 118 bugs collected in the dataset.

We extracted all the bugs from the dataset. Some samples of the bugs we encountered are:

(1) BUG ID 121; for which 264 tests ran in total out of which 119 passed and 145 failed. As you can see in the code below, the bug fix was adding a not null check to the channel object.

```
- if (channel.getPipeline().get(HttpRequestDecoder
    .class) != null {
+ if (channel!=null && channel.getPipeline().get(
    HttpRequestDecoder.class) != null {
```

(2) BUG ID 245; for which 334 tests ran in total out of which 333 passed and 1 failed. As you can see in the code below, the bug fix was adding an exceptional handler for one condition.

```
+ if (!modifier.equals(Modifier.FINAL)) {
+    throw new IllegalStateException("unexpected
    parameter modifier: " + modifier);
+ }
```

3.1.4 ManySStuBs4J.

ManySStuBs4J consists of a collection of fixes for bugs found in 100 Java Maven Projects and top 1000 Java Project repositories. In order to collect bugs, they have created a set of 16 syntactic templates (Stubs) which essentially are small bug fix changes.

The creators reviewed the codes of these projects and scanned for the commit messages with words "bug"/"fix"/"fault" to determine if the commits have bug fixes. All bugs and their fixes which were found are stored in JSON files along with type of the bug (one out of the 16 predefined stubs) and other metadata about the bug. I assume techniques like defined pattern matching/ regular expressions were used to map the bug to one of Stubs. Most of the bug fixes are just single statement fixes, to ensure that they are simple and manageable for analysis.

We also can find statistics related to the stubs in the dataset i.e for every stub, the stub pattern (i.e change modifier, less/ more specific if, missing throws exception, wrong function name, etc) and how many times they occurred in both the repositories. There were 99369 bugs mined in total, 12598 from the 100 Java Maven Projects and 86771 from the top 1000 Java Projects.

We extracted all the bugs from the dataset. Some samples of the bugs we encountered are:

(1) One of the bugs from the 100 Java Maven projects along with the bug fix which was changing boolean parameters in the function return statement.

```
 public Map<String, VariableInstance>
   getVariableInstancesLocal(String executionId,
    Collection<String> variableNames) {
-    return commandExecutor.execute(new
   GetExecutionVariableInstancesCmd(executionId,
    variableNames, false));
+    return commandExecutor.execute(new
   GetExecutionVariableInstancesCmd(executionId,
    variableNames, true));
 }

 public Map<String, VariableInstance>
   getVariableInstancesLocal(String executionId,
    Collection<String> variableNames, String
   locale, boolean withLocalizationFallback) {
-    return commandExecutor.execute(new
   GetExecutionVariableInstancesCmd(executionId,
    variableNames, false, locale,
   withLocalizationFallback));
+    return commandExecutor.execute(new
   GetExecutionVariableInstancesCmd(executionId,
    variableNames, true, locale,
   withLocalizationFallback));
 }
```

(2) One of the bugs from top 1000 Java Projects where the bug was that the variable passed to the get method was wrong: "key" instead of "k".

```
        for (String k : val.keySet()) {
-           result.put(key, new
   ByteArrayByteIterator(val.get(key)));
+           result.put(key, new
   ByteArrayByteIterator(val.get(k)));
        }
```

3.1.5 *QuixBugs.*

QuixBugs is a program-repair tool which tries to find bugs with multi-lingual support - Python and Java [5]. QuixBugs contains 40 programs, each program containing only one line bugs. It also introduces test cases which are common to both the languages. The repository itself contains buggy programs, fixed programs and common testcases for Python and Java.

When Java testcases are run, a report is auto-generated on gradle stating how many test cases exist, how many passed, how many failed: all the distribution statistics. Whereas for Python, running the testcases via pytest directly shows output in the terminal itself with all the distribution statistics. There are total 260 bugs present in the dataset.

Here is a bug from one of the programs in the dataset along with its fix - "Get Factors" in both Java and Python respectively, where instead of returning a empty array it returns a array "n". For this bug, 11 tests ran, 1 passed and 10 failed.

```
-        return new ArrayList<Integer>();
+        return new ArrayList<Integer>(Arrays.asList(n));
```

```
-    return []
+    return [n]
```

3.2 Testing Techniques

This section introduces two powerful tools, Randoop and EvoSuite, that leverage automated test generation to find bugs and validate bug fixes.

Randoop is a unit test generator for Java. It automatically creates unit tests for your classes, in JUnit format. Its unique approach involves two test types:

(1) **Regression Tests:** These tests ensure existing functionalities remain intact after code modifications.
(2) **Error-Revealing Tests:** These gems target hidden bugs by exposing situations that cause the buggy version to fail while the patched version sails through. By pinpointing these discrepancies, Randoop aids developers in verifying bug fixes.

EvoSuite is a tool that automatically generates test cases with assertions for classes written in Java code. To achieve this, EvoSuite applies a novel hybrid approach that generates and optimizes whole test suites towards satisfying a coverage criterion.

For the produced test suites, EvoSuite suggests possible oracles by adding small and effective sets of assertions that concisely summarize the current behavior; these assertions allow the developer to detect deviations from expected behavior, and to capture the current behavior in order to protect against future defects breaking this behaviour.

In our project, we'll strategically leverage both tools. Randoop will help us identify tests that expose bug fixes, while EvoSuite will provide a broader regression test suite to validate the overall stability of the patched versions.

Running the Tools:
We'll focus on classes and methods identified in Milestone 2 related to the bug.

(1) **Randoop**
  (a) The following command will be used, replacing placeholders with specific details.

```
        java -classpath myclasspath:\${
   RANDOOP_JAR} randoop.main.Main gentests
   --testclass=the.class.under.Test

```

  (b) Budget allocation for test generation is crucial. An insufficient budget might hinder the generation of error-revealing tests.
  (c) Generated tests will be saved under the project's test folder with a prefix of "Randoop" for clarity.

(2) **Evosuite**
  (a) We'll target specific classes using the command

```
        java -jar $PATH/evosuite.jar -class
    the.class.under.Test
```

(b) Budget allocation is also important for EvoSuite to generate useful tests.
(c) Generated tests will be placed under the project's test folder with a prefix of "EvoSuite" for identification.

While both tools are widely used and our projects are functional, there's a possibility that one or both tools might not generate tests for specific projects. In such cases, a clear explanation for the ineffectiveness will be provided. By following these steps and addressing potential challenges, we can leverage Randoop and EvoSuite to identify test cases that pinpoint the effectiveness of bug fixes in the patched versions of our projects.

Code Coverage:
After generating the tests, we'll collect information on how they perform and the code coverage they achieve. Our focus is on test execution results and code coverage.

We have updated the text file containing the list of "failed" tests will be updated with the automatically generated tests that expose the bug fix (passing on patched, failing on buggy) for every bug.

3.3 Bug Localization Techniques

Bug localization is a critical aspect of software development aimed at identifying and isolating faulty code segments responsible for triggering bugs. In this project, we employed automated bug localization techniques to pinpoint the exact statements within the bug repositories that contribute to software defects. Our approach involved a systematic process of analyzing test coverage and computing suspiciousness scores to prioritize potentially buggy code segments.

In our bug localization methodology, we utilized the following steps:

(1) **Dataset Selection**
From every bug dataset (Defects4J, Bears, QuixBugs and BugSwarm) we selected minimum three buggy versions which satisfied the following conditions:
   (a) The ability to generate regression tests for the buggy version using either Randoop or Evosuite or both
   (b) Extraction of test coverage information using Clover or an alternative code coverage collection tool.

In the Bugswarm dataset, the changes (bug fix) for few selected bugs were made within test classes, unlike regular classes in a codebase. Test classes don't generate a '.class' file after compilation, which means tools used for code coverage analysis may not gather coverage information for changes made in these classes. This led to a lack of coverage information for those bugs. Due to this, for this scope of this Milestone we chose to include five new bugs which had proper Diff, and we were able to generate coverage reports for them.

For other bug datasets- Bears, Defects4J and QuixBugs; we choose bugs from the existing ones. Finally, below mentioned are the bugs which we choose for this Milestone:
- **Bears:** Bears-141, Bears-143, Bears-137, Bears-131, Bears-130, Bears-21, Bears-222
- **BugSwarm:** commons-lang-224267191, owlapi-158989792, byte-buddy-140517155, mp4parser-107859078, mp4parser-133036862
- **Defects4J:** Csv_2, Csv_1, Compress_1, Csv_3, Csv_4, Codec_2, Codec_3
- **QuixBugs:** FLATTEN, LCS_LENGTH, SHUNTING_YARD, IS_VALID_PARENTHESIZATION, SUBSEQUENCES, TO_BASE, GET_FACTORS

(2) **Suspiciousness Score Calculation**
For each bug, we computed the suspiciousness score of each statement within the classes involved. This information was obtained from milestone 2, existing in the bug's diff.
The suspiciousness score ranks code elements based on their likelihood of causing test failures. It's calculated by analyzing test execution traces: elements executed more by failing tests are ranked higher. This prioritizes debugging efforts, directing attention to the most probable sources of faults in the codebase for quicker resolution.

Here is the formula we used in order to calculate it-

$$\text{Suspiciousness Score} = \frac{\left(\frac{\text{fail(s)}}{\text{total fail}}\right)}{\left(\frac{\text{fail(s)}}{\text{total fail}} + \frac{\text{pass(s)}}{\text{total pass}}\right)}$$

In this formula: - fail(s) is the number of failing test cases that cover a specific line of code.
- total fail is the total number of failing test cases.
- pass(s) is the number of passing test cases that cover the same line of code.
- total pass is the total number of passing test cases.

It considers the proportion of failing test cases that cover the line compared to the total number of failing test cases, relative to the proportion of passing test cases that cover the line compared to the total number of passing test cases. A higher value indicates that the line of code is more suspicious, meaning it's more likely to be faulty or problematic.

We used a excel sheets to track coverage informtion that we require for calculating the above score. Then we choose to use excel macros in order to calculate suspiciousness score, sort and then finally rank every statement as per their suspiciousness scores.

(3) **Metric Computation**
With ground-truth bug localization information available (identifying which statements in the bug version were removed or modified in the patch), we computed the following

metrics for each of the 12+ bugs:

- **Average Rank (AR):** The average rank of all the statements involved in the bug. Involved statements refer to those removed or modified in the patch. For instance, if a bug involves three statements, the AR value is computed as the average rank position of all three statements.
  We already had highlighted the cell's corresponding to the bug fix from Diff file in the excel sheets. We simply calculated the average ranks of these involved statements.

- **First Rank (FR):** The highest rank achieved where one of the involved statements in the bug is located. For example, if a bug involves three statements, the FR value is determined by the highest rank achieved among these statements.
  We directly got this rank from the sorted excel sheet created earlier.

(4) **Reporting Results**
The results of our bug localization efforts were compiled into a CSV file named "BL-Results.csv" and placed in the root of our GitHub repository.

(5) **Visualization**
We also decided to visualize the results we had using Scatter charts along with a regression line and r-value for every bugs' every metric. After that, we will rank bug datasets based on LD and CodeBLEU metrics. We will also compare bug positions in different ranks using the Spearman Rank Order Correlation. Interpretation of the coefficient value will help determine any correlation's strength, whether positive or negative.

4 EVALUATION

List of research questions:

- **RQ1: Benchmarking Bug Repositories**
  Benchmarking provides an effective way to evaluate different tools [6]. Bug benchmarking is a valuable practice in software engineering that promotes transparency, accountability, and improvement in bug detection and fixing processes.

  It aids developers and researchers in making informed decisions about adopting or enhancing bug detection or fixing tools and methodologies.

  This research focuses on benchmarking bugs in diverse datasets to categorize them based on specific properties. As a proxy for bug complexity, we collect a set of metrics that encapsulate various dimensions of code changes and differences between buggy and patched files.

  The selected metrics include:

(1) **CChange (Number of classes changed/added/deleted to patch the bug):** - Measures the impact on the class structure of the code due to bug fixes.
(2) **MChange (Number of methods changed/added/deleted to patch the bug):** - Quantifies the modification to methods during the bug-fixing process.
(3) **LChange (Number of lines changed/added/deleted to patch the bug):** - Represents the total lines of code altered in the process of fixing a bug.
(4) **LD (Levenshtein edit distance between buggy and patched files):** - Captures the similarity/dissimilarity between the original and patched files using the Levenshtein edit distance.
(5) **CB (Cyclomatic complexity of buggy files):** - Evaluates the complexity of the original (buggy) code using Lizard.
(6) **CP (Cyclomatic complexity of patched files):** - Measures the complexity of the patched code using Lizard.
(7) **CC (Complexity change: |CB-CP|):** - Indicates the absolute change in complexity between the original and patched code.
(8) **CodeBLEU:** - Utilizes the CodeBLEU metric to assess the similarity between buggy and patched files.

For measuring cyclomatic complexity, the Lizard tool is recommended. A simple command can be executed after installation to compute the cyclomatic complexity for each specific Java file in the buggy or patched version. Levenshtein edit distance is computed using the Levenshtein module in Python.

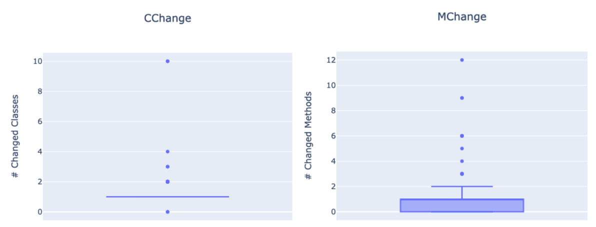
We have drafted a Python script to fetch any metric for any one of the bug belonging to five datasets. It can be used as-

```
    python3 automate.py [bug-dataset] [bug-name] [
     metric]
```

- 'bug-dataset' refers to the dataset containing bugs for analysis. Any one of - Defects4J, Bears, QuixBugs, BugSwarm
- 'bug-name' specifies the name of the bug within the dataset.
- 'metric' allows the selection of a specific metric to be computed. Any one of - CChange, MChange, LChange, LD, CB, CP, CC, CodeBLEU

After generating JSON files for metrics across five projects, we aggregated the data and utilized box plots to visually depict the results. Subsequently, a dynamic visualization dashboard was crafted, accessible through the following execution command:

```
    python3 dashboard.py
```



You can refer the visualizations in the figure above. By employing these metrics and an automated script, we aim to create a

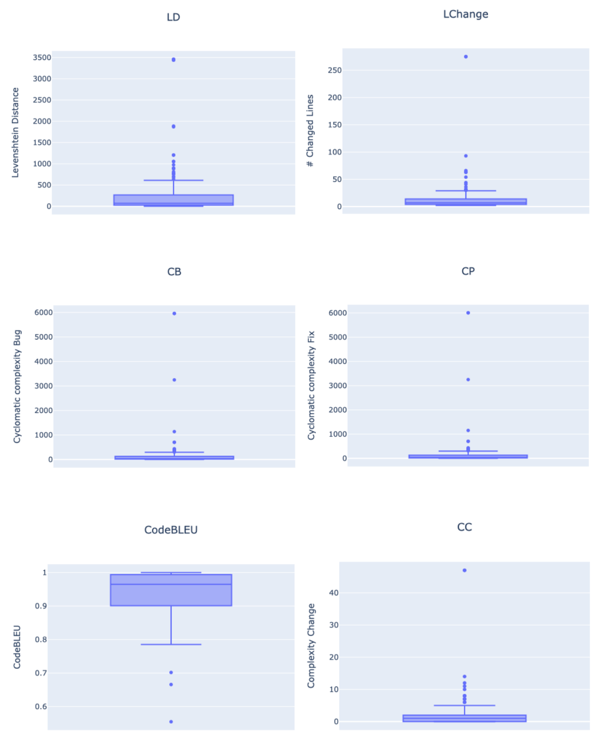


Figure 1: Combined Box Plots Per Metric

comprehensive benchmarking framework for assessing bug complexity.

- RQ2: Evaluating Testing Techniques

This project aimed to evaluate the effectiveness of Randoop and EvoSuite, automated test generation tools, in uncovering bug fixes within our Java projects. We employed both tools to generate tests for the original buggy versions and the patched versions of the code. By analyzing the test results and code coverage information, we can assess the strengths and weaknesses of each technique.

★ **Randoop's Focus: Unearthing Bug Fixes** Randoop generated error-revealing tests. These tests specifically targeted the bug manifestations, causing the buggy versions to fail while the patched versions passed successfully. This pinpointed behavior provided strong evidence that the implemented bug fixes effectively addressed the issues present in the original code. Randoop's ability to identify these discrepancies proved valuable in validating the correctness of the patches.

Some key insights we would light to highlight are shown in the visualizations below for bugs that Randoop was able to generate tests that fail on buggy version and pass on patched version-

There were some bugs for which Randoop was not able to generate any error revealing tests. Their information can be seen in

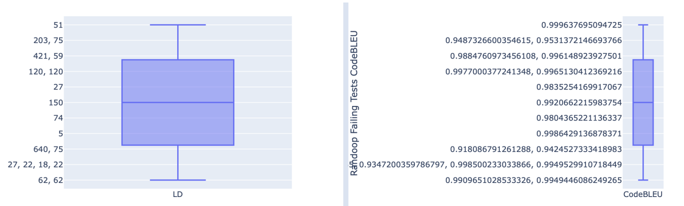


Figure 2: Randoop's Failing Tests on Defects4J

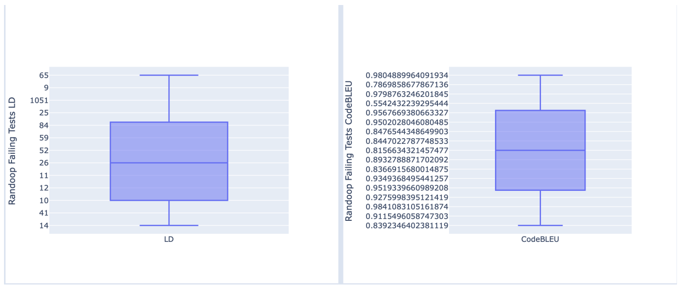


Figure 3: Randoop's Failing Tests on QuixBugs

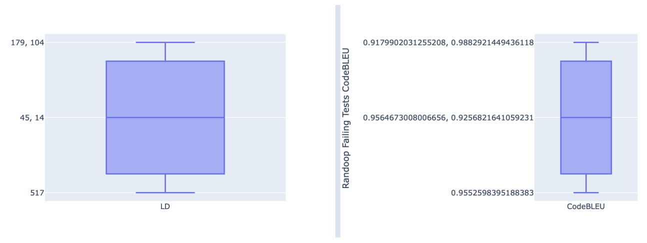


Figure 4: Randoop's Failing Tests on Bears

the visualizations below:

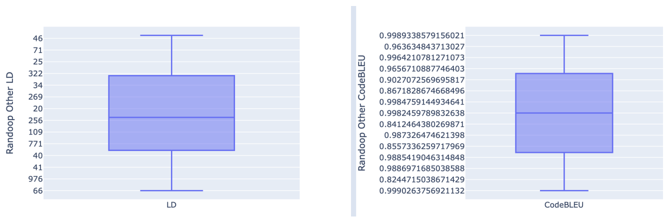


Figure 5: Randoop's Other Tests on Defects4J

For Randoop, we did face some issues were certain bugs were not able to generate any error revealing or regression tests. For example-

(1) **Defects4J:** We were able to generate new test suite for Randoop for almost every bug. For bug Mockito_1,2,3,4 - Patched Version and Mockito_3- buggy version: We faced several compilation issues in unrelated classes and even after retrying multiple times, we did not get any generated tests. In addition, several buggy versions of code did not generate

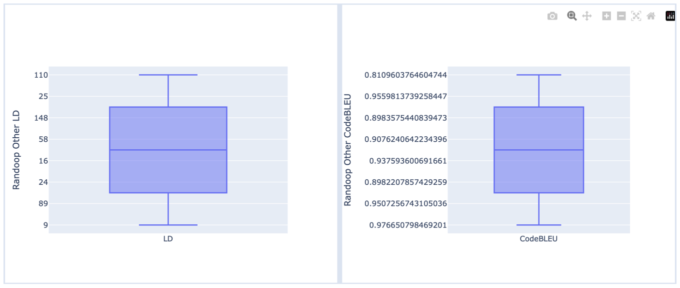


Figure 6: Randoop's Other Tests on QuixBugs

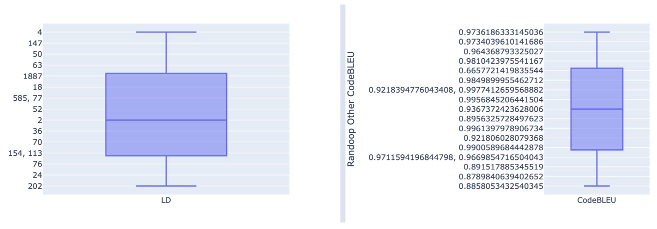


Figure 7: Randoop's Other Tests on Bears

any error revealing tests even after increasing budget. Apart from those few, Randoop was actually able to generate some error revealing tests that matched our criteria. To reproduce our test results, simply run our *run_tests_defects4j.py* script. This script leverages the generated .tar.gz2 file to execute all the tests. We encountered compilation issues after adding tests directly to the test folders. To address this, we opted to run the tests first via our script and then capture their details in tests.txt file. This approach ensured successful test execution despite the compilation challenges.

Here is the list of bugs for which we were able to generate failing tests matching our criteria:
Chart_4, Codec_4, Compress_1, Compress_4, JacksonCore_1, JacksonCore_3, JacksonXml_4, Math_1, Math_4, Time_1, Time_2

(2) **QuixBugs:** We were able to generate new test suite for Randoop 19/20 bugs . For bug TO_BASE - After re-running the command to generate tests multiple times, even after increased the budget, we faced timeout issues. We also were not able to generate tests for SUBSEQUENCES for the buggy version due to dependency issues. We also faced difficulties while generating error revealing tests even after tring a timeout of 1000 seconds.
Here is the list of bugs for which we were able to generate failing tests matching our criteria:
GCD, BUCKETSORT, KNAPSACK, LEVENSHTEIN, FIND_IN_SORTED, LIS, BREADTH_FIRST_SEARCH, POSSIBLE_CHANGE, SHORTEST_PATH_LENGTH, DEPTH_FIRST_SEARCH, FLATTEN, SIEVE, MERGESORT, NEXT_PALINDROME, SHORTEST_PATHS, NEXT_PERMUTATION, REVERSE_LINKED_LIST

(3) **Bears:** We were able to generate randoop tests for all the bugs in the bears dataset except for one, however they were not really efficient in uncovering failures within our classes. From the 20 bugs only 2 (Bears-123 and Bears-141) generated error revealing tests. The rest of the tests generated by the tool passed on both versions of the code and after some manual examination we could see that some of the generated tests were redundant. Following the same procedure as we did with evosuite, we decided to up the allocated resources for test generation to see if we could improve our results. Even after trying multiple run configuration we still encountered situations such as bug Bears-226 where no tests could be generated.

Here is the list of bugs for which we were able to generate failing tests matching our criteria:
Bears-99, Bears-141, Bears-123

(4) **Bugswarm:** Our testing process yielded promising results. We generated tests that uncovered bugs in a significant number of projects, with the number of tests per bug varying greatly. Additionally, we were able to create regression tests for almost all patched versions. There were a few isolated cases where test generation failed for specific bugs, but overall, the findings demonstrate the effectiveness of our approach. But, due to compile issue in unrelated class - "JSONTest", we were not able to run any tests. Also, for some classes we had to generate dependencies for the class under test and then include those in the classpath to the randoop command. For most of the projects Regression tests were generated and for only some error revelaing tests were generated. For project whose failed tests were from test classes those generated no tests for eg: fastjson2-14377005698 class under test was DateTypeTest which did not generate a .class file after compiling hence no tests were generate.

⋆ EvoSuite's Strength: Comprehensive Regression Testing
EvoSuite provided a broader perspective by generating regression test suites. These tests covered various functionalities within the code, ensuring that existing features remained intact after code modifications. While EvoSuite might not directly pinpoint bug fixes like Randoop, its comprehensive test suites offered valuable insights into the overall stability of the patched versions. A high success rate of EvoSuite's tests on the patched versions indicated a lower likelihood of regressions introduced during the bug fixing process.

Similar experiments like Randoop were done with Evosuite as well. Below is the visualization for which Evosuite was able to generate tests that fail on buggy version and pass on patched version-

Here is the last one, bugs for which evosuite was not able to generate tests that matched our filtering criteria.

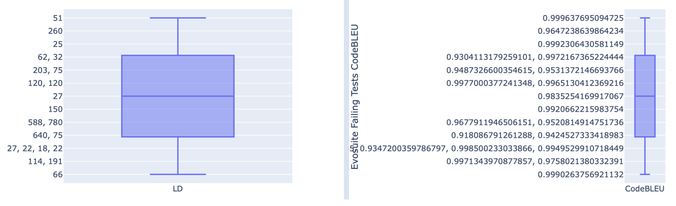


Figure 8: Evosuite Failing Tests on Defects4J

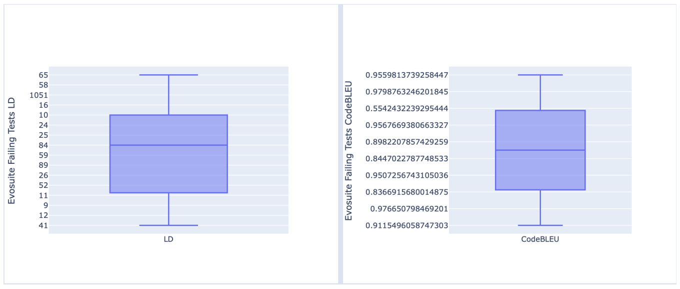


Figure 9: Evosuite Failing Tests on QuixBugs

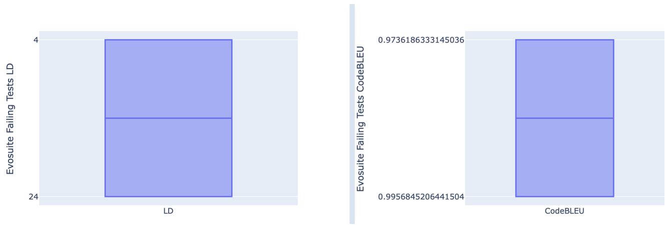


Figure 10: Evosuite's Failing Tests on Bears

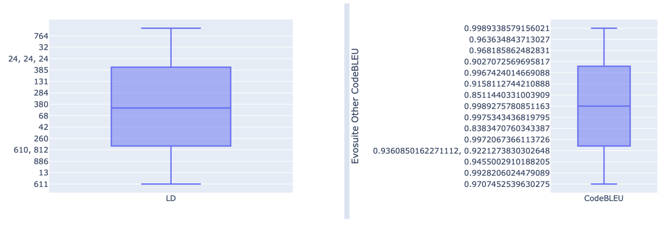


Figure 11: Evosuite's Other Tests on Defects4J

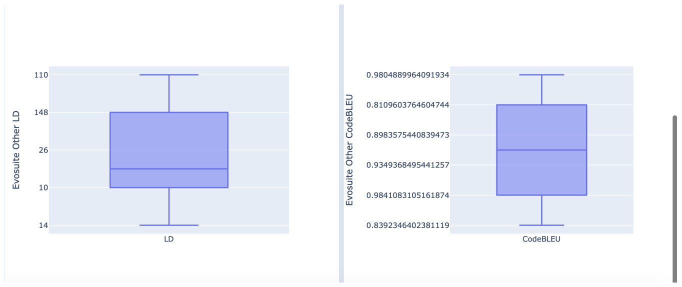


Figure 12: Evosuite's Other Tests on QuixBugs

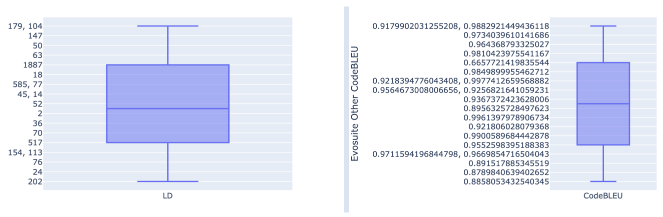


Figure 13: Evosuite's Other Tests on Bears

For Evosuite, we faced less issues while generating and running the tests.

(1) **Defects4J:** We were able to generate Regression tests for Evosuite for each and every bug 68/68. In the first few attempts, even after giving around 240-300 seconds to generate and successfully generating and testing the tests; we were not able to find regression tests that passed on patched version and failed on buggy version. But, we decided to give it one more try and generated evosuite tests using defects4j's inbuild function- after using this we were able to generate several regression tests that passed on patched version and failed on buggy version.
To reproduce our test results, simply run our script *run_tests_defects4j.py*. This script leverages the generated .tar.gz2 file to execute all the tests. We encountered compilation issues after adding tests directly to the test folders. To address this, we opted to run the tests first via our script and then capture their details in tests.txt file. This approach ensured successful test execution despite the compilation challenges.

Here is the list of bugs for which we were able to generate failing tests matching our criteria:
Chart_1, Chart_4, Compress_1, Compress_3, Compress_4, Csv_2, Gson_2, Gson_4, JacksonCore_1, JacksonCore_2, JacksonDatabind_2, JacksonDatabind_3, JacksonDatabind_4, JacksonXml_4, Jsoup_4, JxPath_1, JxPath_3, Lang_5, Math_1, Math_3, Math_4, Mockito_2, Time_1, Time_2, Time_4

(2) **QuixBugs:** We were able to generate Regression tests for Evosuite for each and every bug 20/20. Evosuite did not generate many tests that matched our criteria for test selection-passed on patched version and failed on buggy version. We were able to get some tests for a few bugs only.
Here is the list of bugs for which we were able to generate failing tests matching our criteria:
BUCKETSORT, LEVENSHTEIN, QUICKSORT, FIND_IN_SORTED, BREADTH_FIRST_SEARCH, POSSIBLE_CHANGE, LCS_LENGTH, SHORTEST_PATH_LENGTH, DEPTH_FIRST_SEARCH, FLATTEN, TO_BASE, SIEVE, NEXT_PALINDROME, IS_VALID_PARENTHESIZATION, SHORTEST_PATHS, SHUNTING_YARD, NEXT_PERMUTATION, REVERSE_LINKED_LIST, GET_FACTORS

(3) **Bears:** With Evosuite we were able to generate new test suites for of 17/20 bugs. For this subset of tests, one thing we would like to highlight is that we did encounter a handful of tests that would fail in the buggy version and pass on the patched version, however we were not able to generate any tests in the patched version that failed in the buggy version. Tests generated in the patched version ended up passing in both versions for all of the generated tests. For the remainder of the bugs, since we were not able to generate new tests with our base configuration, we decided to increase the allocated resources. After multiple iterations reaching up to 6 cores, and an allocated time per class of 60 minutes, we still saw no new tests being generated. We created an empty test class as a placeholder in order to have it show in our coverage reports
Here is the list of bugs for which we were able to generate failing tests matching our criteria:
Bears-143, Bears-246

(4) **BugSwarm:** We were able to generate tests for a significant amount of bugs. For classes whose fix patch was for a test class, we failed to generate any tests for it using Evosuite after trying multiple ways to do so. For project whose failed tests were from test classes those generated no tests for eg: fastjson2-14377005698 class under test was DateTypeTest which did not generate a .class file after compiling hence no tests were generate. For some classes we got out of memory errors as well(fastjson2-13792916908,fastjson2-14206220675,fastjson2-14206220786).For classes like shiro-16456433682,projects-java-fundamentals-exercises-7068728014, no tests were generated due absent artifacts.None of the generated tests failed on the bugs.

Both Randoop and EvoSuite offer distinct advantages in automated test generation. Randoop shines in directly identifying bug fixes through error-revealing tests, while EvoSuite provides a broader safety net with comprehensive regression testing.

⋆ **Coverage Reports**
Code coverage reports provided insights into the portions of code that the tests actually execute.

While we successfully generated coverage reports for individual test sets within Defects4J, including Buggy-Randoop, Patched-Randoop, Buggy-Evosuite, and Patched-Evosuite, attempts to generate reports for the entire test suite encountered persistent compilation errors. These errors occurred even after incorporating the necessary Maven and EvoSuite dependencies. Despite seeking assistance from Campuswire, we were unable to identify a solution that addressed the compilation issues and allowed us to generate reports for the full test suite. In order to generate coverage reports for one bug, we deleted the auto-generated tests and generated reports with the original tests only.

We were successful in generating all 6 coverage reports for almost all the Bugs in QuixBugs and Bears which includes Buggy-Randoop, Patched-Randoop, Buggy-Evosuite, Patched-Evosuite, Buggy-All and Patched-All. Even for BugSwarm, we successfully generated coverage reports for most of the bugs, we faced several unrelated compilation errors for other bugs.
In conclusion, code coverage reports provide valuable data for evaluating the effectiveness of automated testing techniques. By incorporating these reports into our analysis, we gained a deeper understanding of how Randoop and EvoSuite interacted with the code, ultimately leading to a more comprehensive evaluation of their strengths and weaknesses in uncovering bug fixes.

⋆ **Details for reproducing results:**

(1) **Defects4J:**
In order to generate Randoop tests for Defects4J, We used a script called *generateTestsViaDefects.py*, which will just need the path to *gen_tests.py* file from defects4j repository in local machine and path to the dataset. To run these tests script *run_tests_defects.py* (needs path to dataset set in the dataset_path variable.) was used.
There is one more script - *defects4j_testgeneration.py* which is obsolete now, but it can be used to generate evosuite randoop tests via commandline, without using defects4j's commands.
In order to generate coverage reports, *run_coverage_defects.py*, which just needs path to dataset set in the dataset_path variable. This script uses defects4j's inbuild coverage tool "Cobertura".

(2) **QuixBugs, Bears, BugSwarm:**
We had a script to generate tests which have been uploaded in root of the project, which worked well but had some issues with compilation of the entire code later. We decided to generate tests, run them and then generate coverage reports for every bug manually, as it was less time consuming.

- RQ3: Evaluating Bug Localization Techniques
We successfully completed bug localization as per strategies explained in Section 3.3. Here's a table that shows Average Rank (AR) and First Rank (FR) for each bug.
To gain deeper insights into the performance of our performed bug localization, we conducted a comprehensive evaluation by benchmarking the collected bugs in Milestone 2. We established connections between various metrics and bug localization results through extensive plotting and analysis.

Specifically, we plotted scatter charts to visualize the relationship between LD (Levenshtein Distance) metric values and AR (Average Rank) as well as FR (First Rank). Each plot was accompanied by a regression line and r-value to provide a quantitative understanding of the relationship between the metrics and bug localization accuracy. The visualizations where generated using dash-plotly, using as a data source a single dataframe. The dataframe was the result of a join between our csv file containing the AR/FR metrics with our benchmark results containing the LD/CodeBleu metrics

Table 1: Bug Localization Results

Bug Name	Bug ID	AR	FR
Defects4J	Csv_2	4	4
Defects4J	Csv_1	6	6
Defects4J	Compress_1	89	89
Defects4J	Csv_3	25	25
Defects4J	Csv_4	1	1
Defects4J	Codec_2	9	9
Defects4J	Codec_3	206	206
QuixBugs	FLATTEN	5.5	5
QuixBugs	LCS_LENGTH	1	1
QuixBugs	SHUNTING_YARD	15	15
QuixBugs	IS_VALID_PARENTHESIZATION	2	2
QuixBugs	SUBSEQUENCES	9	9
QuixBugs	TO_BASE	1	1
QuixBugs	GET_FACTORS	2	2
Bears	Bears-141	4.5	2
Bears	Bears-143	65	64
Bears	Bears-137	1	1
Bears	Bears-131	34	1
Bears	Bears-130	1	1
Bears	Bears-21	9.625	1
Bears	Bears-222	26.333	23
BugSwarm	commons-lang-224267191	21.57142857	19
BugSwarm	owlapi-158989792	11	11
BugSwarm	byte-buddy-140517155	24.73333333	23
BugSwarm	mp4parser-107859078	100.50	1
BugSwarm	mp4parser-133036862	1	1

Once we had these plots, To understand more about how our metrics relate to each other, we ranked the bugs based on each metric. Then, we compared how the bugs ranked differently in each list using something called the Spearman Rank Order Correlation. This helps us see if there's any connection between the rankings, and if so, how strong it is. For example, if the coefficient value is close to 1, it means there's a strong positive connection between the rankings. If it's close to -1, it means there's a strong negative connection. If it's closer to 0, the connection is weaker.

(1) **Defects4J:** Figure 14 shows the scatter plot for Defects4J
Also, here is the table indicating Spearman Rank Order Correlation for Defects4J:

Table 2: Defects4J Evaluation Metrics

Metric	Correlation	R-Value
LD_AR	-0.5714	0.15
LD_FR	-0.5714	0.15
CodeBLEU_AR	0.1786	0.1
CodeBLEU_FR	0.1786	0.1

For this dataset we observed similar values for AR and FR in every bug as the number of statements involved in the patch

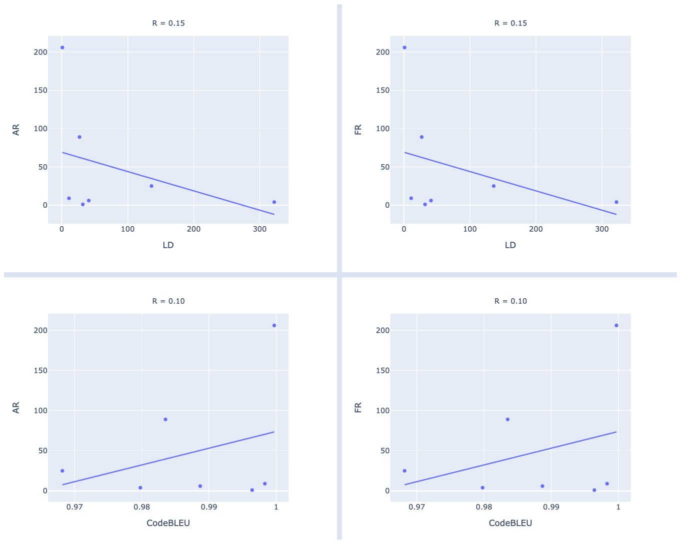


Figure 14: Defects4J Visualization

was low. Comparing LD and both AR/FR we concluded there was a negative correlation as when one metric increases the other would decrease and vice versa. With a coefficient value of -0.5714 this would be deemed a negative, relatively strong correlation. This is what we expected since better bug localization techniques should in turn result in a problematic statement to be more prominent than the rest. When it comes to CodeBleu and AR/FR, we observed a coefficient value of 0.1786 which is a weak, positive correlation. This is also expected, as with better localization accuracy we would be able to properly determine the statements involved in the bug and rank them higher.

(2) **QuixBugs:**
Figure 15 shows the scatter plot for QuixBugs
Also, here is the table indicating Spearman Rank Order Correlation for QuixBugs

Table 3: QuixBugs Evaluation Metrics

Metric	Correlation	R-Value
LD_AR	0.3762	0.14
LD_FR	0.3762	0.14
CodeBLEU_AR	-0.4546	0.19
CodeBLEU_FR	-0.4546	0.21

In our analysis of the QuixBugs dataset, we found some interesting correlations between the metrics LD (Levenshtein Distance) and CodeBLEU with AR (Average Rank) and FR (First Rank). For LD and both AR/FR, we noticed a positive correlation with coefficient values of approximately 0.3762. This indicates that as the LD metric increases, both AR and FR tend to increase as well. This suggests that bugs with higher LD values are more likely to have higher ranks in

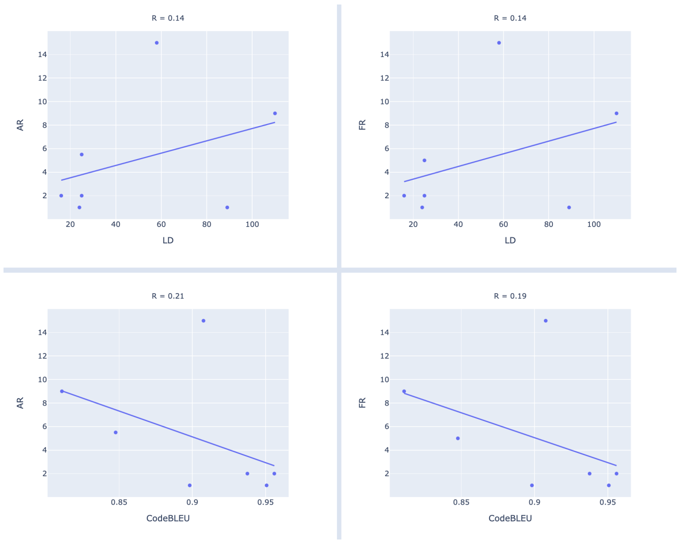


Figure 15: QuixBugs Visualization

terms of both average and first occurrence.

Conversely, when examining CodeBLEU with AR/FR, we observed a negative correlation with coefficient values around -0.4546. This implies that as the CodeBLEU metric increases, both AR and FR tend to decrease. This aligns with our expectations, as higher CodeBLEU scores suggest better localization accuracy, resulting in lower ranks for both average and first occurrence. Overall, these findings provide valuable insights into the relationships between bug localization metrics and ranking accuracy in the QuixBugs dataset. Compared to the other 3 examined datasets, QuixBugs presented the results that best fit the motivations of bug localization techniques. All measured correlations indicate that the content of the bugs in the dataset were ideal for the tested techniques and would result in faster debugging and more efficient development cycles.

(3) **Bears:**
Figure 16 shows the scatter plot for Bears
Also, here is the table indicating Spearman Rank Order Correlation for Bears and corresponding R-values also.

Table 4: Bears Evaluation Metrics

Metric	Correlation	R-Value
LD_AR	0.2319	0.03
LD_FR	-0.1518	0.08
CodeBLEU_AR	0.6377	0.16
CodeBLEU_FR	0.9411	0.25

In our examination of the Bears dataset, we uncovered noteworthy correlations between the metrics LD (Levenshtein Distance) and CodeBLEU with AR (Average Rank) and FR (First Rank). For LD and AR, we found a positive correlation with a coefficient value of approximately 0.2319, indicating that as the LD metric increases, the average rank tends to increase as well. However, for LD and FR, we observed a negative correlation with a coefficient value of about -0.1518. This suggests that as the LD metric increases, the likelihood of a bug being ranked first decreases slightly.

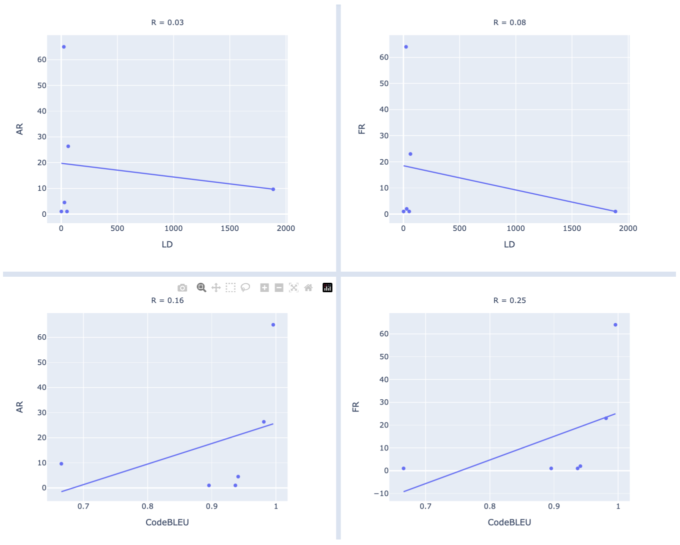


Figure 16: Bears Visualization

On the other hand, when analyzing CodeBLEU with AR/FR, we identified strong correlations. Specifically, we found a positive correlation between CodeBLEU and AR with a coefficient value around 0.6377, indicating that higher CodeBLEU scores correspond to higher average ranks. Similarly, we observed a very strong positive correlation between CodeBLEU and FR, with a coefficient value close to 0.9411. This indicates that bugs with higher CodeBLEU scores are significantly more likely to be ranked first. These findings offer valuable insights into the relationship between bug localization metrics and ranking accuracy in the Bears dataset, shedding light on the effectiveness of bug localization techniques in this context.

(4) **BugSwarm:**
Figure 17 shows the scatter plot for BugSwarm Also, here is the table indicating Spearman Rank Order Correlation for BugSwarm
In our analysis of the BugSwarm dataset, we observed interesting correlations between the metrics LD (Levenshtein Distance) and CodeBLEU with AR (Average Rank) and FR (First Rank). For LD and AR, we found a positive correlation with a coefficient value of 0.3, indicating that as the LD metric increases, the average rank tends to increase as well. Similarly, for LD and FR, we observed a positive correlation with a coefficient value of approximately 0.3591, suggesting that bugs with higher LD values are more likely to be ranked

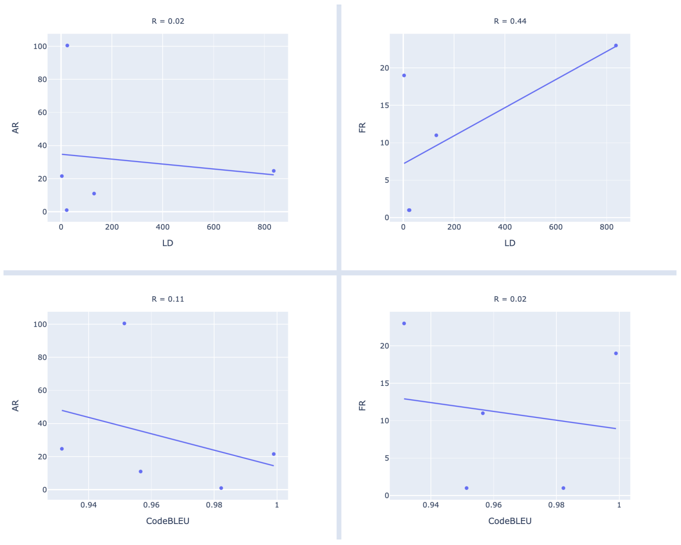


Figure 17: BugSwarm Visualization

Table 5: BugSwarm Evaluation Metrics

Metric	Correlation	R-Value
LD_AR	0.3	0.02
LD_FR	0.3591	0.44
CodeBLEU_AR	-0.6	0.11
CodeBLEU_FR	-0.2052	0.02

first.

Conversely, when examining CodeBLEU with AR/FR, we identified negative correlations. Specifically, we found a negative correlation between CodeBLEU and AR with a coefficient value of -0.6, implying that as the CodeBLEU metric increases, the average rank tends to decrease. Similarly, for CodeBLEU and FR, we observed a negative correlation with a coefficient value around -0.2052, indicating that bugs with higher CodeBLEU scores are less likely to be ranked first. These findings offer valuable insights into the relationship between bug localization metrics and ranking accuracy in the BugSwarm dataset, highlighting areas for potential improvement in bug localization techniques.

Now, we will rank the bugs based on each metric and compare the position of bugs in different ranks.
By analysing the gathered results we have concluded that the used bug localization techniques are most efficient in the QuixBugs dataset. This conclusion was reached due to the strong correlations observed between the Levenshtein Distance and CodeBLEU metrics with the Average and First Rank values. The dataset presents the expected positive correlation between LD and AR/FR as well as negative correlation between CodeBLEU and AR/FR.

Table 6: Ranking

Rank	LD_AR	LD_FR
1	Defects4J	Defects4J
2	QuixBugs	QuixBugs
3	BugSwarm	BugSwarm
4	Bears	Bears

Rank	CodeBLEU_AR	CodeBLEU_FR
1	Bears	Bears
2	BugSwarm	QuixBugs
3	QuixBugs	BugSwarm
4	Defects4J	Defects4J

The obtained coeffcient values reflect how with greater similarities between the localized and actual buggy code correspond to more accurate ranking. As to why this is the case, our team believes that it comes down to the nature of the codebases included in the dataset. The classes in question are less complex compared to other contending datasets, with each being covered by just a handful of lines and little to no branching. The structured and algorithmic nature of the bugs in QuixBugs likely lends itself well to the pattern-recognition capabilities of automated tools, enabling more effective pinpointing and ranking of relevant buggy code.

Following QuixBugs, Bears, Defects4J, and BugSwarm demonstrate better performance in the same heirarchy.

5 CHALLENGES

In following section, we will discuss we challenges we encountered during the project bug dataset-wise.

(1) **Defects4J**
Despite encountering numerous challenges, we managed to address nearly all, if not all, of them by its conclusion. Here is a list of challenges we had faced:
- Despite successfully generating tests for all bugs using Evosuite, the auto-generated tests did not meet our criteria. Initially, we couldn't find a single test that failed on the buggy version but passed on the patched version. After several attempts we eventually found Evosuite tests that matched our criteria.
- Even after multiple attempts, Randoop failed to produce error-revealing tests for all bugs in the buggy version. It was able to generate error revealing tests for a few bugs only.
- During milestone 3, after renaming the auto-generated Randoop and Evosuite tests and integrating them into the original test locations, we faced compile issues. However, we later resolved these by adding the required dependencies, enabling successful compilation of the entire codebase.

(2) **QuixBugs**
QuixBugs being a small dataset, we did not encounter a lot

of challenges, additionally, we were able to address almost all of the challenges by the end of the project. Here is a list of challenges we faced:

- We encountered issues generating error-revealing tests using Evosuite on the buggy version of the codebase. Despite running Evosuite for an extended duration of 2 hours, we were unable to produce the desired tests.
- We faced challenges integrating Clover with Gradle to generate coverage reports. Despite attempting various integration methods, including command-line approaches, IntelliJ and Eclipse plugins, as well as Gradle-specific plugins, we were unable to generate the desired reports. As an alternative, we opted to utilize Jacoco to produce Clover reports, which proved successful.
- Since the Clover reports for Quixbugs were generated using Jacoco, we encountered limitations in obtaining detailed insights, particularly regarding which tests hit specific lines in the test files, as offered by Clover. To address this gap, we devised a workaround by generating Clover reports for each test individually. This enabled us to calculate the number of passing and failing tests hitting each line, subsequently aggregating this data to derive comprehensive results.

(3) **Bears**
The Bears dataset proved to be a challenge due to its large size and bug diversity. Initially we had selected a subset of bugs randomly to use as test subjects, but as we created and tested scripts to automate their compilations, test generation and coverage reports, we realized that we needed to change our approach. When it came to test generation, we were initially unsuccessful at generating Evosuite tests for most bugs and had to experiment with multiple resource configurations to obtain the desired results.

- Instead of completely random bug subsets, we decided to curate the pool of available bugs by eliminated this who had very specific requirements or dependencies. For example we eliminated any bug that required a java version that was not compatible with either Java 17 or 18. Anothe example were bugs that required call to third party services such as a Spotify api.
- Since no Evosuite tests were being generated for our bugs, we ran some tests on a handful of bugs and ran Evosuite on them in parallel, each with a different configuration. We managed to generate tests for most bugs by increasing the allocated time by 10 minutes. For all the remaining bugs we linearly increased the time (capped at 60 mins) until we manage to obtain results.
- For coverage generation we encountered some compilation issues when running all tests including the ones generated by the automated testing tools. We had to update our pom file to include all necessary dependencies to ensure they all ran properly.

(4) **BugSwarm**
Out of all other bug repositories, We faced numerous challenges while working with BugSwarm which are listed below:

- While generating tests for BugSwarm using Randoop and Evosuite, we faced several issues. For a significant number of bugs in our dataset, the bug fix was in the test file itself. Test classes, unlike regular classes, do not generate a ".class" file after compilation, making it impossible to obtain generated tests through Randoop and Evosuite.
- Code coverage analysis tools rely on the presence of compiled ".class" files to determine which lines of code have been executed during testing. Due to test classes not generating ".class" file, we are not able to generate coverage reports for these bugs.
- The calculation of suspiciousness scores required coverage reports. However, because coverage reports were unavailable, as previously mentioned, we had to choose new bugs for Bug localization in Milestone 4, necessitating significant rework. Nonetheless, we managed to complete it at the end for five bugs.

6 CONCLUSION

This project examined how well automated techniques can find the root cause of bugs in real software. We used tools like Randoop and Evosuite to generate tests that target bug fixes, then gave code statements a "suspiciousness score" based on how often failing tests ran them. By comparing these scores to the actual bug locations, we evaluated how accurate our approach was on datasets like Defects4J, Bears, QuixBugs and BugSwarm. The results show that these techniques have promise for speeding up bug finding in real-world projects. They can highlight potentially faulty code, saving developers time and effort.

REFERENCES

[1] René Just, Darioush Jalali, and Michael D. Ernst. 2014. Defects4J: a database of existing faults to enable controlled testing studies for Java programs. In Proceedings of the 2014 International Symposium on Software Testing and Analysis (ISSTA 2014). Association for Computing Machinery, New York, NY, USA, 437–440. https://doi.org/10.1145/2610384.2628055

[2] D. A. Tomassi et al., "BugSwarm: Mining and Continuously Growing a Dataset of Reproducible Failures and Fixes," 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), Montreal, QC, Canada, 2019, pp. 339-349, doi: 10.1109/ICSE.2019.00048.

[3] Madeiral, F., Urli, S., Maia, M., & Monperrus, M. (2019, February). Bears: An extensible java bug benchmark for automatic program repair studies. In 2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER) (pp. 468-478). IEEE.

[4] Rafael-Michael Karampatsis and Charles Sutton. 2020. How Often Do Single-Statement Bugs Occur? The ManySStuBs4J Dataset. In Proceedings of the 17th International Conference on Mining Software Repositories (MSR '20). Association for Computing Machinery, New York, NY, USA, 573–577. https://doi.org/10.1145/3379597.3387491

[5] H. Ye, M. Martinez, T. Durieux and M. Monperrus, "A Comprehensive Study of Automatic Program Repair on the QuixBugs Benchmark," 2019 IEEE 1st International Workshop on Intelligent Bug Fixing (IBF), Hangzhou, China, 2019, pp. 1-10, doi: 10.1109/IBF.2019.8665475.

[6] Lu, Shan & Li, Zhenmin & Qin, Feng & Tan, Lin & Zhou, Pin & Zhou, Yuanyuan, BugBench: Benchmarks for Evaluating Bug Detection Tools (2005)