FixEval: Execution-based Evaluation of Program Fixes for Programming Problems

Md Mahim Anjum Haque  
Virginia Tech  
Blacksburg, VA, USA  
mahim@vt.edu

Wasi Uddin Ahmad  
University of California, Los Angeles  
Los Angeles, CA, USA  
wasiahmad@ucla.edu

Ismini Lourentzou  
Virginia Tech  
Blacksburg, VA, USA  
ilourentzou@vt.edu

Chris Brown  
Virginia Tech  
Blacksburg, VA, USA  
dcbrown@vt.edu

Abstract—The complexity of modern software has led to a drastic increase in the time and cost associated with detecting and rectifying software bugs. In response, researchers have explored various methods to automatically generate fixes for buggy code. However, due to the large combinatorial space of possible fixes for any given bug, few tools and datasets are available to evaluate model-generated fixes effectively. To address this issue, we introduce FixEval, a benchmark comprising of buggy code submissions to competitive programming problems and their corresponding fixes. FixEval offers an extensive collection of unit tests to evaluate the correctness of model-generated program fixes and assess further information regarding time, memory constraints, and acceptance based on a verdict. We consider two Transformer language models pretrained on programming languages as our baseline and compare them using match-based and execution-based evaluation metrics. Our experiments show that match-based metrics do not reflect model-generated program fixes accurately. At the same time, execution-based methods evaluate programs through all cases and scenarios designed explicitly for that solution. Therefore, we believe FixEval provides a step towards real-world automatic bug fixing and model-generated code evaluation. The dataset and models are open-sourced.

I. INTRODUCTION

Repairing programs is one of the hardest and most expensive processes in software engineering. Finding and fixing errors, or debugging, takes up nearly 50% of the total software development costs [8] and 70-80% of software engineers’ time [31]. Automation of program repair could enhance programmers’ productivity [36] and reduce software development costs [22]. Current research aims to provide solutions to automate program repair [14], [30]. Automatic program repair is an active area of research that can greatly relieve programmers from the burden of manually fixing bugs in large codebases [29], [13], [12]. Researchers have recently started adapting language models to automate program repair tasks. Models such as PLBART [2] and CodeT5 [40] have demonstrated promise in automating software engineering tasks, including automated program repair.

While many approaches are being studied in the literature, more support is needed to better evaluate automated repair methods. Prior work, such as TFix [7] and BIFI [41], rely on n-gram metrics, e.g., BLEU or Edit Similarity. A limitation of n-gram metrics is that they penalize generated fixes if they differ from the reference, even if the fix is valid. CodeBLEU [35] attempts to mitigate this by aggregating weighted n-gram, data flow, and syntax matches. While CodeBLEU is an improved strategy for evaluation, both n-gram metrics and CodeBLEU do not account for the large and complex space of program functionalities. Therefore, a well-defined test suite is necessary [5], [20], [11], [1] to evaluate a fix’s correctness. A generated fix is considered functionally correct if it passes a set of unit tests. Overall, there is an increasing need for unit-test-driven benchmarks to assess program fixes generated by automatic program repair models.

In this work, we introduce FixEval, a benchmark consisting of solutions to programming problems submitted by users in the AtCoder [6] and Aizu Online Judge [4] platforms. Competitive programming requires programmers to attempt to solve tough problems within a specific time and memory limit. The process of competitive programming boils down to submitting code, receiving a verdict, making educated changes, and repeating these steps until an acceptable solution is reached. Thus, forming parallel examples of a buggy solution (not accepted) and a solution that passes a set of unit tests (accepted) could facilitate program repair evaluation. In FixEval, we accompany buggy and correct program pairs with a suite of unit tests to evaluate the functional correctness of generated bug fixes. FixEval contains solutions to 700 programming challenges in both Python and Java and 25 test cases on average per problem. We demonstrate the effectiveness of FixEval through experimental evaluation and analysis of program repair techniques.

Contributions: The contributions of our work are summarized as follows: (1) We introduce FixEval, a context-aware program repair dataset that, along with unit tests, incorporates additional considerations—namely time and space complexity—to assess the functional correctness of repaired programs. (2) We benchmark the two most popular pretrained sequence-to-sequence language models PLBART and CodeT5 on FixEval. (3) With thorough analysis, we demonstrate the necessity of unit-test-based program repair evaluation.

II. RELATED WORK

A. Program Repair Benchmarks

A comparison between FixEval and the recent program repair benchmarks is provided in Table I. DeepFix [15] con-
program submissions that vary in size and difficulty. Although lines of code derived from 712K Java and 3.28M Python frames the task as machine translation from a buggy code to tasks in various ways. One of the most common approaches is [14]. Prior works in the literature model bug-fixing experience for developers by generating bug fixes automatically. [14].

Table I: A comparison between FixEval and other existing code repair datasets for machine learning. CR and CE indicate code review comments and compilation errors, respectively.

| Dataset | Test Size | Avg. #Tokens | Input Type | Error Type | Test Cases |
|---------|-----------|--------------|------------|------------|------------|
| DeepFix | 6971      | 203          | Program    | CE Only    | No         |
| Review4Repair | 2961 | 320 + 37 | Program + CR | All | No |
| Bug2Fix | 5835      | ≤ 50, ≤100   | Function   | All        | No         |
| Github-Python | 6545 | 10 - 128 | Program    | CE Only    | Yes        |
| FixEval | 15k       | 30, 243k     | Program    | All        | Yes        |

sists of approximately 7K C programs written by students in an introductory programming course across 93 programming tasks. However, DeepFix only covers compiler errors, does not provide test cases for evaluation, and fails to reflect real-world software applications. Review4Repair [18] contains 55,060 and 2,961 examples of Java patches for training and evaluation, respectively. This work aims to repair code patches with the help of code reviews. Bug2Fix [38] is a popular corpus used in CodeXGLUE [27] that contains examples of the buggy and fixed Java code. However, the examples are at the function level, so cross-function dependencies are not modeled. Bug2Fix also lacks unit tests to check for functional correctness. The GitHub-Python dataset [41] is a collection of 38K buggy and 3M bug-free unparalleled code snippets from GitHub open-source Python projects. The 128 token limit significantly reduces the overall problem complexity. However, the output code is defined as successful if it has no AST errors, which limits the focus only to compiler errors.

Existing benchmarks, including test suites, have also been introduced to support automated program repair research. For instance, datasets such as IntroClass [23] and Refactory [17] consist of student assignments from introductory programming courses and provide unit tests. In contrast, QuixBugs [25] and Defects4J [19] provide real-world buggy programs with test suites. FixEval is substantially larger than both datasets, consisting of more lines of code (QuixBugs: 1,034; Defects4J: 321,000; FixEval: 54M in Java and 61M in Python). As a result, unlike existing benchmarks, FixEval facilitates the training of machine learning models. Moreover, QuixBugs only consists of programs with one-line defects, while Defects4J consists of Java code from five open-sourced programs. In comparison, FixEval contains bugs that span multiple lines of code derived from 712K Java and 3.28M Python program submissions that vary in size and difficulty. Although FixEval was not collected from software repositories, we believe it is more representative for evaluating the capabilities of program repair models than the datasets above.

B. Program Repair Methods

Automated program repair aims to improve the debugging experience for developers by generating bug fixes automatically [14]. Prior works in the literature model bug-fixing tasks in various ways. One of the most common approaches frames the task as machine translation from a buggy code to a fixed one. Several researchers have shown language modeling is effective for automating coding tasks, such as program generation [2], translation [3], and auto-completion [10]. Nevertheless, there needs to be more research on applying language modeling in automated bug fixing and code repair.

C. Evaluating Pretrained Language Models

Due to the recent success of large-scale language models in many domains [34], [9], [37], new techniques have been introduced with pretraining objectives relevant to source code. Models such as BART [24], GPT [10], and T5 [34] have been applied to software engineering tasks, demonstrating improvements in automating development tasks such as code generation, translation, bug detection, etc. For example, PLBART [2] uses denoising autoencoding to pretrain Transformer [39] on programming language corpora. In contrast, TFix [7] is a proposed method evaluating T5 [34] by leveraging commits from GitHub repositories to fix bugs. We train a subset of these models on our dataset to evaluate their performance.

III. FixEval Dataset

The FixEval dataset contains parallel examples of the buggy and correct programs written in Java and Python languages. FixEval examples are collected from CodeNet [33]. CodeNet dataset is a collection of programs written in over 50 languages by participants to solve competitive programming problems at the AtCoder and Aizu Online Judge platforms. FixEval includes open-sourced unit tests for the AtCoder programming problems to facilitate evaluations to assess repaired programs’ functional correctness. Furthermore, FixEval takes the time and memory limit set for each programming problem into consideration while executing repaired programs against the source code.

We argue that while competitive programming is not an exact reflection of real-world professional software development environments, by taking time and memory requirements into consideration, FixEval extorts the practice of requiring software engineers to write efficient code in industrial settings [28]. Since FixEval contains programs with different lengths and difficulties and is written by many programmers, FixEval as a benchmark would foster future work on automatic program repairing.
Problem Statement: A biscuit making machine produces $B$ biscuits at the following moments: $A$ seconds, $2A$ seconds, $3A$ seconds and each subsequent multiple of $A$ seconds after activation. Find the total number of biscuits produced within $T + 0.5$ seconds after activation.

Constraints: $1 \leq A, B, T \leq 20$, All input values are integers

Time Limit: 2 secs; Memory Limit: 1024MB; Problem Difficulty: A

---

Buggy Program in Java

```java
import java.util.*;

public class Main {

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int A = sc.nextInt();
        int B = sc.nextInt();
        int T = sc.nextInt();

        int S = T / A;
        System.out.println(s * b);
    }
}
```

Fixed Program in Java

```java
import java.util.*;

public class Main {

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int A = sc.nextInt();
        int B = sc.nextInt();
        int T = sc.nextInt();

        int S = T / A;
        System.out.println(s * b);
    }
}
```

---

**A. Dataset Construction**

In the CodeNet dataset, every program is associated with a user_id and submission_date. We group the submitted programs by user_id and then chronologically sort them. We refer to such chronological user submissions for a particular problem as a submission path. Each submission is associated with status, i.e., acceptance or error type. We referred to the submission status as a verdict for the submission. If the submitted code passes all the hidden test cases, the verdict is Accepted (AC). Otherwise, submissions may receive a verdict from among 12 different options, the most frequent being: (i) Wrong Answer (WA), i.e., failed one or more test cases; (ii) Time Limit Exceeded (TLE), i.e., the program did not run within the intended time limit; (iii) Compilation Error (CE), i.e., the program did not compile; and (iv) Runtime Error (RE), i.e., program execution was not successful [33]. For a user solving a particular problem, a sample submission path could be [WA, WA, TLE, AC] representing three failed submissions consisting of two incorrect codes and one inefficient implementation before arriving at the correct solution.

We pair each unaccepted submission (i.e., a buggy code) from a user with the accepted submission (i.e., a bug-free code) to form an example (data point) in FixEval. A sample example can be viewed in Figure 1. FixEval consists of examples in Java and Python created from all 154K users’ submissions. We de-duplicate the programs submitted for each problem using Jaccard similarity. We used the javalang4 tokenizer for Java and the tokenize5 standard library for Python. As shown in Table II, we create stratified dataset splits based on problems to ensure a clear partition (80-10-10) in the train, test, and validation, with no overlapping problems or submissions across splits. We ensure all the test and validation data examples include test cases.

---

**B. Test Suite Collection**

We download all the publicly available test cases by AtCoder site. The test cases are manually created by domain experts (e.g., problem setters for the programming challenges) to cover all possible cases and ensure the functional correctness of the submitted programs. We also acknowledge the potential threat to validity as all the unit tests weren’t manually validated. The test cases are organized into individual input and output files. The input files are directly used as input to a program. Correspondingly, the output of the executed program is compared with the output files. To construct the FixEval test suite, we match the problem names from AtCoder with names available as problem metadata in the CodeNet dataset. We manually remove some problems from FixEval. The problems are constraint satisfaction problems where the main goal is to satisfy conditions based on rules or designed constraints. Consequently, we also remove problems for which many combinatorial outputs are equivalently valid. To tackle programs with numerical output, we assume a program is accepted if the difference between the program output and the test case output is below a certain precision threshold. All evaluations use the most frequent precision cutoff ($10^{-5}$). On average, FixEval validation and test set contain 24 and 25 test cases per problem.

---

**C. Task Difficulty**

The average length of the Java and Python programs in FixEval are 331 and 236 tokens, respectively. While the length of a program does not indicate the level of difficulty of repairing the program, we argue that since FixEval programs’ are larger than the existing program repair datasets (as shown in Table I), FixEval is a challenging benchmark for automatic program repair models.

Furthermore, we conjecture that the task labels assigned to AtCoder contest problems indicate problems’ difficulty (labels

---

3From CodeNet, we extracted total 0.5M submission paths.
4https://github.com/c2nes/javalang
5https://docs.python.org/3/library/tokenize.html
| Language | Problem Count | Example Count |
|----------|---------------|---------------|
|          | Train | Valid | Test | Total | Train | Valid | Test | Total |
| Java     | 2,160 | 279   | 279  | 2,718 | 156k  | 44k   | 43k  | 245k  |
| Python   | 1,951 | 244   | 244  | 2,439 | 567k  | 301k  | 243k | 1,111k |

**TABLE II: FixEval dataset statistics.**

Fig. 2: Distribution of task difficulty (labels A to E indicate increasing difficulty) for the AtCoder problems belong to the test split in **FixEval**.

A to E represent increasing difficulty). In **FixEval**, we retain the task labels and ensure the distribution of the labels across splits is even. The distribution of the task labels that we refer to as task difficulty for the test split is presented in Figure 2.

IV. Experiment Setup

A. Evaluation Metrics

To understand how accurately models perform on **FixEval**, we evaluate both conventional n-gram-based metrics and our proposed execution-based metrics, explained next.

1) **N-gram-based Metrics**

- **Exact Match (EM)** indicates the percentage of the generated program fixes exactly match the reference programs.
- **BLEU** computes n-gram overlap between a model-generated fix and the reference. We use corpus-level BLEU scores [32].
- **Syntax Match (SM)** represents the percentage of the subtrees extracted from the Abstract Syntax Tree (AST) of a generated program that matches the subtrees in the AST of the reference programs.
- **Dataflow Match (DM)** indicates the ratio of the number of matched candidate dataflows and the total number of reference dataflow [35].
- **CodeBLEU (CB)** is designed to measure the quality of a code with respect to a reference [35]. Compared to BLEU, CodeBLEU also considers logical correctness based on an Abstract Syntax Tree (AST) in conjunction with data flow structure and grammatical similarity.
- **Compilation Accuracy (CA)** indicates the percentage of generated program fixes that can be run without any compilation error. We use off-the-shelf compilers `javac` and `py_compile`\(^7\) for Java and Python languages, respectively.

7[https://docs.python.org/3/library/py_compile.html](https://docs.python.org/3/library/py_compile.html)

2) **Execution-based Metrics**

In program repair tasks, the input and output typically have high lexical overlapping. However, n-gram-based metrics may not accurately indicate the functional correctness of model-generated program fixes. Further, a program can be fixed in multiple ways that differ from the reference program. Therefore, n-gram-based metrics may not be ideal for program repair evaluation. Thus, we also evaluate **FixEval** with execution-based metrics to alleviate these limitations.

Evaluating all generated programs on execution for all available test cases is memory-intensive and time-consuming. Therefore, we randomly select two program pairs per problem from the test split and evaluate them on all the available test cases. We select the data points with a similar distribution of the verdicts to ensure the evaluation data follows the actual distribution of the test data for different verdicts (AC, WA, TLE, etc.). Since our goal is not to exhaustively evaluate all models but to showcase the efficacy of the proposed dataset, we only evaluate CodeT5, the current state-of-the-art, on relevant tasks. We input the buggy code appended with the verdict information as input to the model and generate 10 outputs using beam search decoding. Then, we evaluate the output programs by running our test suite that simulates how online judges evaluate submitted programs. Our execution-based evaluation metrics, pass\(k\) and TCA\@k\), were introduced in recent works [21], [16]. For self-containment, we provide the descriptions as follows:

- **Pass\(k\)** [21] evaluates functional correctness where \(k\) code samples are generated per problem. A problem is considered solved if any sample passes all the unit tests, and the total fraction of problems solved is reported. However, this computation of pass\(k\) can have a high variance. Hence, we follow [10] to compute pass\(k\), i.e., we generate \(k \leq n\) samples per task (in this work, \(n = 10\) and \(k \leq 10\)), count the number of correct samples \(c \leq n\) that pass all unit tests and calculate the unbiased estimator of pass\(k\) as follows.

\[
\text{pass}\@k := \mathbb{E}_{D_{test}} \left[ 1 - \left( \frac{n-c}{k} \right) \right],
\]

where \(D_{test}\) denotes the **FixEval** test set. Note that this is a strict measure, as a code repair is considered unsuccessful if a single failed test case exists.

- **Test Case Average (TCA@k)** [16] indicates the average number of test cases passed. Often, solutions successfully pass a subset of the test cases but fail to pass tests that cover corner cases. TCA@k allows for a less stringent model evaluation,
as Pass\(k\) accuracy may obscure model improvements. Concretely, let \(P\) be the set of problems in the test set and \(|P|\) be the number of problems in \(P\). Let the code fixes generated to solve problem \(p \in P\) be denoted as \((\text{code}_{i}^{p})\), where \(i\) denotes the index of generated fix and \(k\) is the total number of generated fixes. Furthermore, let the set of test cases for problem \(p\) be \(\{(x_{p,c}, y_{p,c})\}_{c=1}^{C_{p}}\), where \(x_{p,c}\) and \(y_{p,c}\) are the input, output pair and \(C_{p}\) is the number of available test case pairs for that problem. Then, the test case average for \(k\) generated fixes (TCA@\(k\)) is

\[
\frac{1}{|P|} \sum_{p \in P} \sum_{i=1}^{k} \sum_{c=1}^{C_{p}} \frac{1}{|C_{p}|} \sum_{c=1}^{C_{p}} 1 \{ \text{eval} ((\text{code}_{i}^{p}), x_{p,c}) = y_{p,c} \}, \tag{2}
\]

where eval is the function evaluating a code fix in a test case by matching the output with the intended result.

\subsection*{B. Baselines}

We consider the following two Transformer language models as the baseline methods:

- **PLBART** [2] is a BART [24] model trained on programming language corpora using three learning strategies: token masking, token deletion, and token infilling.

- **CodeT5** [40] is a T5 model [34] pretrained on programming languages via multiple objectives, such as span prediction and identifier tagging prediction. CodeT5 uses unimodal (code only) and bimodal (code text pairs) data for pretraining.

In addition, we consider **Naive Copy** as a baseline - the input buggy code is copied to the output (fixed code). Since typically there is a significant overlap between the buggy code and its fix, this baseline shows the minimum a model could achieve in n-gram-based evaluation metrics.

\subsection*{Implementation:}

We finetune the base variants of PLBART and CodeT5 released by the respective authors on FixEval programs. We finetune them with AdamW optimizer [26] with 32 batch size, \(5 \times e^{-5}\) learning rate, early stopping with patience set to 3 and 100 warm-up steps. For inference, we use beam decoding with a beam size of 5.

\section*{V. Results}

We aim to address the following questions through our experiments and analyses: (1) How well do pretrained Transformer models perform on FixEval? and (2) How do n-gram-based metrics track performance relative to execution-based evaluation? Our results validate the need for better program repair evaluation practices, demonstrating that FixEval can fill a critical need in this line of research.

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|c|c|c|c|c|c|}
\hline
Method & Language & Verdict & BLEU & EM & SM & DM & CB & CA \\
\hline
Naive Copy & Java & \xmark & 80.28 & 0.0 & 84.22 & 53.64 & 75.43 & 89.93 \\
& Python & \xmark & 68.55 & 0.0 & 70.12 & 60.51 & 68.47 & 96.56 \\
\hline
PLBART & Java & \xmark & 58.49 & 0.45 & 66.92 & 43.08 & 57.23 & 31.36 \\
& Java & \cmark & 59.84 & 1.46 & 68.01 & 44.99 & 58.62 & 33.04 \\
& Python & \xmark & 61.89 & 2.32 & 64.32 & 48.81 & 61.13 & 91.16 \\
& Python & \cmark & 62.25 & 2.46 & 63.31 & 49.73 & 62.21 & 92.21 \\
\hline
CodeT5 & Java & \xmark & 62.31 & 2.96 & 74.01 & 52.30 & 63.37 & 63.03 \\
& Java & \cmark & 62.54 & 2.45 & 73.93 & 53.29 & 63.71 & 64.23 \\
& Python & \xmark & 64.92 & 2.74 & 68.79 & 56.21 & 63.53 & 92.80 \\
& Python & \cmark & 64.67 & 2.97 & 68.45 & 56.04 & 63.28 & 92.70 \\
\hline
\end{tabular}
\caption{N-gram-based evaluation performances on FixEval. EM (Exact Match), SM (Syntax Match), DM (Dataflow Match), CB (CodeBLEU), and CA (Compilation Accuracy).}
\end{table}

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|c|c|c|c|c|c|c|}
\hline
Method & Language & Verdict & \multicolumn{4}{c|}{pass@\(k\)} & \multicolumn{4}{c|}{top-\(k\) TCA} \\
& & & \(k = 1\) & \(k = 3\) & \(k = 5\) & \(k = 10\) & \(k = 3\) & \(k = 5\) & \(k = 10\) \\
\hline
Naive Copy & Java & \xmark & 0.0 & - & - & - & 37.95 & - & - \\
& Python & \xmark & 0.0 & - & - & - & 41.55 & - & - \\
\hline
PLBART & Java & \xmark & 7.51 & 14.21 & 17.32 & 24.14 & 39.89 & 33.02 & 31.88 & 29.56 \\
& Java & \cmark & 8.43 & 17.65 & 21.51 & 27.17 & 43.87 & 37.78 & 34.71 & 32.78 \\
& Python & \xmark & 6.15 & 12.59 & 15.74 & 19.98 & 49.81 & 40.79 & 37.63 & 34.43 \\
& Python & \cmark & 7.01 & 13.12 & 16.97 & 21.91 & 48.05 & 40.81 & 37.89 & 34.01 \\
\hline
CodeT5 & Java & \xmark & 8.65 & 15.62 & 19.63 & 24.44 & 41.00 & 34.00 & 32.70 & 29.60 \\
& Java & \cmark & 10.94 & 18.77 & 22.66 & 27.96 & 44.99 & 38.80 & 35.87 & 32.90 \\
& Python & \xmark & 6.86 & 13.07 & 16.27 & 20.51 & 50.20 & 41.20 & 38.50 & 35.20 \\
& Python & \cmark & 7.32 & 13.94 & 17.47 & 22.63 & 48.75 & 41.16 & 38.37 & 34.88 \\
\hline
\end{tabular}
\caption{Execution-based evaluation performance on FixEval.}
\end{table}
Fig. 3: TCA increases as edit similarity between buggy and reference code increases for both Java and Python.

A. N-gram-based Evaluation

Table III presents the n-gram-based evaluation results of all the baseline methods. The Verdict column indicates the use of verdict information as conditional input when generating a candidate program fix. We observe that Naive Copy performs the best in all n-gram-based measures except for Exact Match (EM). This is because buggy and fixed code pairs typically overlap significantly. Between the compared models, CodeT5 performs better than PLBART across all metrics and programming languages. We anticipate this is due to their identifier-aware pretraining that helps CodeT5 learn useful patterns.

Further, we observe a marginal performance increase in the baseline models with verdict information as conditional input for Java programs. However, there is no such indication for Python. We hypothesize that the lengthy description of the causes of test failures in Java positively impacts performance. We further analyze the impact of verdicts in Section VI-E.

B. Execution-based Evaluation

We compare n-gram-based and execution-based evaluations to check whether they correlate. The execution-based evaluation results are presented in Table IV. Naive Copy performed the best according to the n-gram-based metrics, and its performance in the execution-based evaluation demonstrates the limitations of n-gram-based evaluation. This suggests TCA and pass@k are better indicators for functional program correctness and evaluate models better than n-gram-based metrics.

Further, we observe a marginal performance increase in the baseline models with verdict information as conditional input for Java programs. However, there is no such indication for Python. We hypothesize that the lengthy description of the causes of test failures in Java positively impacts performance. We further analyze the impact of verdicts in Section VI-E.

B. Execution-based Evaluation

We further analyze model performances on \textsc{FixEval} to understand the effects of several components, e.g., verdict information, decoding algorithms, etc. All analyses are based on the CodeT5 model on sampled examples.

A. Correlation to Edit Similarity

We analyze model performance based on edit similarity, assuming lower edit similarity between the buggy and fixed code indicates more difficulty in fixing the code. We sort all the data points on the \textsc{FixEval} test set based on the buggy and reference (fixed) codes’ edit similarity and plot the test case average of the CodeT5 model for both Java and Python. In Figure 3, we observe a mostly upward trending curve for Java, which indicates a positive correlation between edit similarity and model performance. However, no such correlation is evident for Python.

B. Correlation to Task Difficulty

We analyze the effect of task difficulty as described in Section III-C. From Figure 4, we observe that the model performance degrades as difficulty increases (increasing difficulty from label A to E). This verifies our hypothesis that fixing a program that solves a challenging problem is harder.

C. Correlation to Evaluation Verdict

We analyze the effect of verdict type on performance. Figure 5 shows that compilation errors (CE) are the easiest to solve as these mostly deal with syntactical changes to correct a program, whereas runtime errors (RE) or time limit exceeded errors (TLE) are much harder to fix since these indicate semantically incorrect code that requires multiple changes, sometimes the main logic of the algorithm.

D. Effect of Decoding Algorithms

We use CodeT5 to generate fixed programs with various decoding strategies: (i) greedy, (ii) beam search with beam size 10, and (iii) temperature-scaled nucleus sampling ($p = 0.95$ and $t = 0.7$). Figure 6 shows that beam search usually performs better than greedy decoding and sampling. We experiment with varying sampling temperatures from 0.2 to 1.2 and observe minor performance changes. We believe this is due to the nature of the problem, as the fixed program remains mostly similar to the buggy version, which results in the model remaining confident in its predictions. Hence, the temperature scaling doesn’t result in substantial distribution changes.
we observe that the model without verdict information attempts to add unnecessary but syntactically correct code snippets that cannot fix the actual error. In contrast, the model with verdict information can pinpoint the exact location of the error and make the code more consistent. We provide relevant examples in Figure 7.

F. Summary of Analysis

We study performance trends concerning edit similarity, task difficulty, and evaluation verdicts to show that some bug-fixing tasks are trivial while many are challenging. Therefore, we encourage future work to consider all aforementioned aspects while performing evaluations. The choice of decoding strategy produces marginal differences; therefore, it is not a crucial factor in improving bug-fixing models. We encourage future works to study feedback-based (e.g., feedback from an oracle) approaches to improve bug-fixing models.

G. Limitations

We acknowledge that fixing programming problem solutions does reflect real-world software bugs from professional developers. Therefore, more effort is necessary to develop benchmarks that simulate bug fixing of real-world software programs to evaluate automated program repair models.

VII. CONCLUSION

We introduce FixEval, a dataset of program fixes to facilitate bug-fixing model development and evaluation. We benchmark two pretrained language models on FixEval and showcase that traditional evaluation metrics are sub-optimal compared to execution-based metrics that capture contextual program repair requirements often found in practice.

We believe FixEval could help facilitate several software engineering tasks, such as code completion, code editing, code search, verdict-conditioned code repair, verdict prediction, and chain edit suggestion tasks. In the future, since the provided test cases are language-independent, our work can be extended to other programming languages.

REFERENCES

[1] T. Ackling, B. Alexander, and I. Grunert. Evolving patches for software regression. In Proceedings of the 12th International Conference on Genetic and Evolutionary Computation, GECCO ’11, page 1427–1434, New York, NY, USA, 2011. Association for Computing Machinery.
[2] W. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang. Unified pre-training for program understanding and generation. In Proceedings of the 2021 Conference of NAACL: HLT, pages 2655–2668, June 2021.
[3] W. U. Ahmad, M. G. R. Tushar, S. Chakraborty, and K.-W. Chang. Avatar: A parallel corpus for java-python program translation. arXiv preprint arXiv:2108.11590, 2021.
[4] Aizu Online Judge. 2004. https://judge.u-aizu.ac.jp/onlinejudge.
[5] A. Arcuri. On the automation of fixing software bugs. In Companion of the 30th International Conference on Software Engineering, ICSE Companion ’08, page 1003–1006, New York, NY, USA, 2008.
[6] Atcoder. Atcoder opened-source test cases. https://www.dropbox.com/sh/nx31lzq7m59a/AAAY1q2iHEHxOSw6-yL3aWg6l?dl=0, 2020.
[7] B. Berabi, J. He, V. Raychev, and M. Vechev. Tifix: Learning to fix coding errors with a text-to-text transformer. In M. Meila and T. Zhang, editors, Proceedings of the 38th ICML, volume 139 of Proceedings of Machine Learning Research, pages 780–791. PMLR, 18–24 Jul 2021.
[8] T. Britton, L. Jing, G. Carver, P. Cheak, and T. Katzenellenbogen. Reversible debugging software-quantify the time and cost saved using reversible debuggers. University Cambridge: Cambridge, UK, 2013.
[9] T. Brown, B. Mann, N. Ryder, M. Subbiah, et al. Language models are few-shot learners. In H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, and H. Lin, editors, Advances in Neural Information Processing Systems, volume 33, pages 1877–1901. Curran Associates, Inc., 2020.
[10] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. d. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, et al. Evaluating large language models trained on code. arXiv:2107.03374, 2021.
[11] F. DeMarco, J. Xuan, D. Le Berre, and M. Monperrus. Automatic repair of buggy if conditions and missing preconditions with smt. In Proceedings of the 6th International Workshop on Constraints in Software Testing, Verification, and Analysis, CSTVA 2014, page 30–39, New York, NY, USA, 2014. Association for Computing Machinery.
[12] 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, 2020.
[13] Y. Ding, B. Ray, P. Devanbu, and V. J. Hellendoorn. Patching as translation: The data and the metaphor. In Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering, ASE ’20, page 275–286, New York, NY, USA, 2020.
[14] C. L. Goues, M. Pradel, and A. Roychoudhury. Automated program repair. Commun. ACM, 62(12):56–65, nov 2019.
[15] R. Gupta, S. Pal, A. Kanade, and S. Shevade. Deepfix: Fixing common c language errors by deep learning. In Thirty-First AAAI Conference on Artificial Intelligence, volume 31, Feb. 2017.
[16] D. Hendrycks, S. Basart, S. Kadavath, M. Mazeika, A. Arora, E. Guo, C. Burns, S. Puranik, H. He, D. Song, and J. Steinhardt. Measuring coding challenge competence with APPS. In Thirty-fifth NeurIPS Datasets and Benchmarks Track (Round 2), 2021.
[17] Y. Hu, U. Z. Ahmed, S. Mchtaev, B. Leong, and A. Roychoudhury. Refactoring-based program repair applied to programming assignments. In Proceedings of the 34th ASE, page 388–398, 2019.
[18] F. Huq, M. Hasan, M. M. A. Haque, S. Mahbub, A. Iqbal, and T. Ahmed. Review-repair: Code review aided automatic program repairing. Information and Software Technology, 143:106765, 2022.
[19] R. Just, D. Jalali, and M. D. Ernst. 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, page 437–440, New York, NY, USA, 2014. Association for Computing Machinery.
[20] D. Kim, J. Nam, J. Song, and S. Kim. Automatic patch generation learned from human-written patches. In Proceedings of the 2013 ICSE, page 802–811. IEEE Press, 2013.
[21] S. Kulal, P. Pasupat, K. Chandra, M. Lee, O. Padon, A. Aiken, and P. S. Liang. Speo: Search-based pseudo-code to code. In NeurIPS, volume 32. Curran Associates, Inc., 2019.
[22] C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer. A systematic study of automated program repair: Fixing 55 out of 105 bugs for 88 each. In 34th ICSE, pages 3–13, 2012.

Fig. 6: Pass@1 and TCA@1 accuracy for greedy decoding, beam search, and nucleus sampling.

E. Effect of Modeling Verdict

We analyze the test examples repaired correctly when the model has access to the verdict information. We observe that verdict information helps in fixing some instances of buggy code. We further observe that the model without verdict information attempts to add unnecessary but syntactically correct code snippets that cannot fix the actual error. In contrast, the model with verdict information can pinpoint the exact location of the error and make the code more consistent. We provide relevant examples in Figure 7.
public class Main {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int a = sc.nextInt();
        int b = sc.nextInt();
        if ((a - b) % 2 == 0) {
            System.out.println((a + b) / 2);
        } else {
            System.out.println("IMPOSSIBLE");
        }
    }
}

gcd(n, m) = \begin{cases} 
    \gcd(n, m), & \text{if } (m < n) \\
    \gcd(n, m), & \text{if } (n == 0) \\
    \gcd(n, m % n), & \text{if } (n == 0) \\
    \gcd(n, m), & \text{if } (m < n) \\
    \gcd(n, m % n), & \text{if } (n == 0) \\
    \gcd(n, m), & \text{if } (m < n) \\
    \gcd(n, m % n), & \text{if } (n == 0) \\
    \end{cases}