FIXEVAL: EXECUTION-BASED EVALUATION OF PROGRAM FIXES FOR COMPETITIVE PROGRAMMING PROBLEMS

Anonymous authors
Paper under double-blind review

ABSTRACT

The increasing complexity of software has led to a drastic rise in time and costs for identifying and fixing bugs. Various approaches are explored in the literature to generate fixes for buggy code automatically. However, due to the large combinatorial space of possible fixes for a particular bug, few tools and datasets are available to evaluate model generated fixes effectively. In this work, we introduce **FIXEVAL**, a benchmark comprising buggy code submissions to competitive programming problems and their respective fixes. **FIXEVAL** is composed of a rich test suite to evaluate and assess the correctness of model-generated program fixes and further information regarding time and memory constraints and acceptance based on a verdict. We consider two Transformer language models pretrained on programming languages as our baselines 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.[1]

1 INTRODUCTION

Repairing software 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 (Britton et al., 2013) and $70-80\%$ of software engineers' time (National Institute of Standards and Technology, 2002). Current research aims to provide better solutions to automate this process (Le Goues et al., 2019; Gazzola et al., 2017), but this problem is still far from being solved. Automatic program repair is an active area of research[2] that can greatly relieve programmers from the burden of manually fixing bugs in large codebases (Mesbah et al., 2019; Ding et al., 2020; Dinella et al., 2020). Researchers have recently increasingly applied statistical and neural methods to automate program repair tasks. Models such as BART (Lewis et al., 2019) and GPT (Chen et al., 2021) have demonstrated great success in solving problems relevant to code. Techniques to automatically repair programs, and the availability of accurate benchmarks for evaluating them, are useful for enhancing programming productivity (Seo et al., 2014) and reducing development costs (Le Goues et al., 2012).

While many approaches are being studied to automate program repair, more support is needed to evaluate generated fixes. Prior work, such as Tfix (Berabi et al., 2021), BIFI (Yasunaga & Liang, 2021), and CodeBLEU (Ren et al., 2020) rely on token-level matching metrics, such as Exact Match. A limitation of match-based metrics is that they penalize generated fixes if they differ from the reference by a single token, even if the fix is valid. CodeBLEU attempts to mitigate this by aggregating weighted n-gram, data flow, and Abstract Syntax Tree (AST) matches. However, this does not account for the fact that bugs can be fixed using various code instructions, and programs do not have to use the same algorithm, syntax, data flow, or ASTs to solve the same problem. Thus, match-

[1] https://github.com/FixEval/FixEval_official

[2] See https://program-repair.org

based methods cannot effectively evaluate generated code since they cannot account for the large and complex space of program functionality in solutions. A well-defined test suite is, therefore, necessary (Arcuri, 2008; Kim et al., 2013; DeMarco et al., 2014; Ackling et al., 2011) to evaluate a fix's correctness using program behavior instead of syntax. A generated fix is considered functionally correct if it passes a set of unit tests. Overall, there is an increasing need for better evaluation methods and benchmarks to assess code fixes generated by program repair models.

In this work, we introduce **FixEval**, a benchmark dataset consisting of competitive programming submissions from users of AtCoder (Atcoder, 2020) and Aizu Online Judge (Aizu Online Judge, 2004). Competitive programming consists of programmers attempting some of the toughest problems to be solved within a specific time and memory limit. The process of competitive programming comes down to submitting code, receiving a verdict, making educated changes, and repeating until an acceptable solution is reached. Thus using competitive programming submissions to construct FixEval provides data useful for evaluating automated bug fixes, in particiular parallel pairs of buggy and corrected programs with access to corresponding unit tests. **FixEval** contains solutions to 700 Python and Java problems along with more than $40,000$ test cases for evaluation. We demonstrate the effectiveness of our benchmark through an experimental analysis evaluating bug fix generation for state-of-the-art models in program repair.

Contributions: The contributions of our work are summarized as follows: **(1)** We introduce **FixEval**, a *context-aware* dataset that incorporates additional considerations for programs, namely time and space complexity, for evaluating code generated by deep learning models to automatically fix programming bugs. **(2)** We provide a comprehensive comparison of state-of-the-art models on **FixEval** and evaluate their accuracy for repairing buggy code. Further, we open source this dataset and the baseline models used for our evaluation. **(3)** Finally, we experimentally verify the advantages of the proposed execution-based program repair evaluation derived from our introduced test suite.

2 Related Work

Program Repair Automated program repair aims to improve debugging tasks for developers by generating fixed programs from buggy code automatically (Le Goues et al., 2019). Prior work tackles this problem in various ways. One of the most common methods is to model code generation as a machine translation task from a buggy code to a fixed one, e.g., by training a language model on code with various pretraining objectives (Ahmad et al., 2021a). Several researchers have shown language modeling is effective for automating coding tasks, such as program generation (Ahmad et al., 2021a; Wang et al., 2021), program translation between languages (Ahmad et al., 2021b), and program auto-completion (Chen et al., 2021). Nevertheless, there needs to be more research on applying language modeling in automated bug fixing and code repair.

Evaluating Pretrained Language Models Due to the recent success of large-scale language models in many domains (Raffel et al., 2019; Brown et al., 2020; Shoeybi et al., 2019), new techniques have been introduced with different pretraining objectives relevant to code. Models such as BART (Lewis et al., 2019), GPT (Chen et al., 2021), and T5 (Raffel et al., 2019) have been applied to software engineering tasks, demonstrating improvements in automating development tasks such as code generation, translation, bug detection, etc. For example, PLBART (Ahmad et al., 2021a) is a BART model trained on programming corpora with token masking, token deletion, and token infilling training strategies. In contrast, Tfix (Berabi et al., 2021) is a proposed method evaluating T5 (Raffel et al., 2019) by leveraging commits from GitHub repositories to solve bugs detected by ESLint,[3] the most popular static analyzer for JavaScript code. We train a subset of these models on our dataset with various input configurations to evaluate their performance.

Program Repair Benchmarks There are several existing examples of benchmarks to help researchers evaluate deep learning techniques for automatically fixing bugs. A comparison of **FixEval** to these recent benchmarks are available in Table 1. DeepFix (Gupta et al., 2017) consists 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 (Huq et al., 2022) contains $55,060$ training data along with $2,961$ test data for Java patches. This work aims to repair

[3] https://eslint.org/

	DeepFix	**Review4Repair**	**Bug2Fix**	**Github-Python**	**FIXEVAL**
Language	C	Java	Java	Python	Java, Python
Dataset Test Size	6971	2961	5835, 6545	15k	43k, 243k
Avg. #Tokens	203	320 + 37	$\leq$ 50, $\leq$100	10 - 128	331, 236
Input Type	Program	Program + CR	Function	Program	Program
Error Type	CE Only	All	All	CE Only	All
Test Cases	No	No	No	No	Yes

Table 1: A comparison between **FIXEVAL** and other existing code repair datasets for machine learning. CR and CE indicate code review comments and compilation errors, respectively.

code patches with the help of code reviews, making match-based methods the only way to evaluate performance. Incorporating review comments as conditional input results in high linguistic variation, making the learning process more difficult and requiring more training examples. Bug2Fix (Tufano et al., 2019a) is a popular corpus used in CodeXGLUE (Lu et al., 2021) that contains buggy and fixed Java code. However, the dataset is only stored at the function level, so cross-function dependencies cannot be modeled. Further, Bug2Fix also lacks unit tests to check for functional correctness. The GitHub-Python dataset (Yasunaga & Liang, 2021) is a collection of 38K buggy and 3M correct 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 program repair benchmarks incorporating test suites have also been introduced to support automated program repair research. For instance, datasets such as IntroClass (Le Goues et al., 2015) and Refactory (Hu et al., 2019) consist of student assignments from introductory programming courses and provide unit tests. However, these benchmarks lack relevance to real-world software. QuixBugs (Lin et al., 2017) and Defects4J (Just et al., 2014) both provide more relevant 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: 54 Million for Java and 61 Million for Python). This allows large-scale training and testing of machine learning techniques for automated program repair. QuixBugs only contains 40 "small" programs and Tufano et al. (2019b) note that the limited size of Defects4J restricts its usage as training data with machine learning models. Further, our benchmark is more diverse and representative of software in practice. QuixBugs only consists of programs with one-line defects, whereas Defects4J consists of Java code from only five open source programs. On the other hand, **FIXEVAL** contains bugs that span multiple lines of code derived from 712K Java and 3.28 Million Python program submissions that vary in size and difficulty.

This paper aims to fill the gaps and limitations of existing datasets by providing better benchmarks for deep learning models in automated program repair research. Existing benchmarks contain introductory programming assignments that do not capture the representation of large real-world bugs. In addition, these automated program repair datasets focus on domain-specific open-source code without considering additional constraints and contexts that vary from project to project. In contrast, **FIXEVAL** provides a thorough test suite to evaluate repairs on efficiency and correctness. To the best of our knowledge **FIXEVAL** is the first context-aware program repair evaluation dataset with a comprehensive test suite that also considers runtime and memory consumption.

3 DATASET: **FIXEVAL**

The **FIXEVAL** dataset consists of Java and Python program submissions from CodeNet (Puri et al., 2021), a collection of programs submitted by competitive programmers to different online judges for evaluation. We enrich the dataset with test suites for the programs in the validation and test set, where each problem has multiple test cases to evaluate the correctness of program submissions. Each test was created for specific problems, enabling rigorous evaluation of program functionality. While competitive programming is not an exact reflection of real-world professional software development environments, **FIXEVAL** consists of programs with varying difficulty and takes time and memory requirements into consideration, a common practice for evaluating software engineers to hire (McDowell, 2019) and crucial for writing efficient code in industrial settings (Mens, 2012).

Problem Statement[4]: 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);
  }
}
```

Figure 1: Example submissions from the FixEval dataset. Buggy and fixed statements are marked in red and green, respectively.

Dataset Construction For each user, we considered the chronologically ordered submission path for each of the problems solved. If the code passes all of the hidden test cases, then the result, also termed as the *verdict* for the code, is considered *Accepted (AC)*. Otherwise, programs 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 (Puri et al., 2021). A full list of possible verdicts for submitted programs is available in the Appendix B.2. Each submission is associated with a verdict. For a user trying to solve a particular problem, a sample submission path could be [WA, WA, TLE, AC] representing 3 failed attempts consisting of two incorrect programs and one inefficient algorithm, before arriving at the correct solution.

We paired each wrong submission with the accepted submission and consider that as a single data point (example) consisting of a pair of a buggy and a fixed program. A sample example can be viewed in Figure 1. **FIXEVAL** consists of submission paths for the corresponding problems in Java and Python for all 154k users and 6.5 million submission paths. On average, users submitted 90 programs from individual accounts. We de-duplicated the submissions using Jaccard similarity to remove multiple submissions for a specific problem. We used the $javalang$[5] tokenizer for Java and the $tokenizer$[6] standard library for Python. As shown in Table 2, we created stratified dataset splits based on problems to ensure that there is a clear partition (80-10-10) in train, test, and validation splits, with no overlapping problems or submissions across splits. Finally, we ensured that all the test and validation data examples include test cases.

Test Suite Collection We download all test cases that are used for evaluating the submitted programs from the open source test pool (Atcoder, 2020) shared by the official AtCoder[7] site. To construct our dataset test suite, we matched the problem names from the CodeNet problem metadata with the AtCoder published website. Then, we matched each problem with the corresponding input and output files provided with the test cases. We also cleaned the test case data manually. For example, there exist programs with different precision cutoffs for numerical output. In other words, the solution of the program is accepted if the difference between the output and the target is below a certain precision threshold. For simplicity, we kept the most frequent precision cutoff, 10^{-8}, across all programs. Also, there are constraint satisfaction problems where the main goal is to satisfy conditions based on rules or design constraints. Consequently, many combinatorial outputs are equivalently valid. We removed such problems and test cases from our evaluation pipeline. More details are available in Appendix B.1.

[5] https://github.com/c2nes/javalang
[6] https://docs.python.org/3/library/tokenize.html
[7] https://atcoder.jp/posts/21

Language	Problem Count				Example Count			
	Train	Valid	Test	Total	Train	Valid	Test	Total
Java	2,160	279	279	2,718	156k	44k	45k	245k
Python	1,951	244	244	2,439	567k	301k	243k	1,111k

Table 2: Statistics of the FixEval dataset.

Finally, our validation set contains an average of 24 test cases per problem, and our test set contains an average of 25 test cases per problem. Each test case is manually generated by domain experts (e.g., problem setters for competitive programming challenges) to ensure the functional correctness of the submitted programs.

Problem Difficulty While the length of a problem solution does not indicate a problem's difficulty, we still emphasize that **FixEval** is composed of Java and Python programs that consist of 331 and 236 tokens, respectively, on average. As shown in Table 1, **FixEval** programs' are larger than the existing program repair datasets. Therefore, **FixEval** is a challenging benchmark for automatic program repair models as they need to tackle longer programs and generate fixes accordingly. In AtCoder, contest problems come with a Task Label (e.g., A, B, C, D, E) and we conjecture that the label indicates the difficulty of the problem. In **FixEval**, we retain the task labels and verify that there is an even difficulty of problems partitioned in all splits by stratified sampling. The distribution of problem difficulty from the overall dataset is presented in Figure 2.

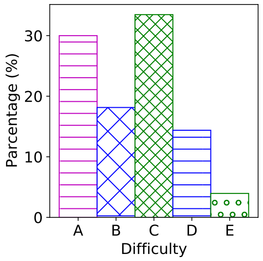


Figure 2: Test set difficulty.

4 Experiment Setup

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

PLBART (Ahmad et al., 2021a) is a BART (Lewis et al., 2019) model trained on programming language corpora using three learning strategies: token masking, token deletion, and token infilling.

CodeT5 (Wang et al., 2021) is a T5 model (Raffel et al., 2019) 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.

Apart from the two baseline models, we consider **Naive Copy** as a baseline. The input buggy code is copied to the output in this approach. Since there is a significant overlap between the buggy code and its fix, this baseline shows the minimum a model could achieve in match-based metrics.

Setup We finetune PLBART and CodeT5 on **FixEval** Java and Python programs using the base variant of both the models, shared by the respective authors and test on our **FixEval** dataset using beam search with a beam size of 5 and batch size of 32. We train with AdamW optimizer (Loshchilov & Hutter, 2019), $5 \times e^{-5}$ learning rate, early stopping with patience set to 3 and 100 warm-up steps.

4.1 Evaluation Metrics

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

4.1.1 Match-based Metrics

Exact Match (EM) evaluates whether a generated program fix exactly matches the reference.

BLEU computes the match-based overlap between a model-generated fix and the reference. We use corpus-level BLEU score (Papineni et al., 2002).

[7] https://atcoder.jp/contests/abc125/tasks/abc125_a

CodeBLEU (CB) (Ren et al., 2020) is designed to measure the quality of a code with respect to a reference. 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 programs that are compilable. We use off-the-shelf compilers, i.e., `javac` for Java and `py_compile`[8] for Python.

Syntax Match (SM) represents the percentage of the sub-trees extracted from the candidate program's Abstract Syntax Tree (AST) that match the subtrees in the reference programs' AST.

Dataflow Match (DM) (Ren et al., 2020) is the ratio of the number of matched candidate dataflows and the total number of reference dataflows.

4.1.2 Execution-based Metrics

In program repair tasks, the input and output typically have high lexical overlapping. However, match-based metrics may not estimate the actual functional correctness of model-generated program fixes. Further, a program can be fixed in multiple ways that differ from the reference program. Therefore, match-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. To reduce time complexity, we randomly selected two data points per problem from the test split, with a similar distribution of error verdicts, to ensure the evaluation data follows the actual distribution of the total 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 generate top-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 by Kulal et al. (2019) and Hendrycks et al. (2021). For self-containment, we provide the descriptions in the following paragraphs.

Pass@k: Kulal et al. (2019) evaluate functional correctness using the pass@k metric, 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 high variance. Hence, we follow Chen et al. (2021) to evaluate pass@k, i.e., we generate $k \leq n$ samples per task (in this paper, $n = 10$ 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:

$$\text{pass@}k := \mathop{\mathbb{E}}_{\mathcal{D}_{test}} \left[1 - \frac{\binom{n-c}{k}}{\binom{n}{k}} \right],$$

where $\mathcal{D}_{test}$ 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): We follow Hendrycks et al. (2021) to compute the average number of test cases passed. 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 $\langle code_p^i \rangle$, 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

$$\frac{1}{|P|} \sum_{p \in P} \frac{1}{k} \sum_{i=1}^{k} \frac{1}{|C_p|} \sum_{c=1}^{|C_p|} 1 \left\{ \text{eval} \left(\left\langle \text{code}_p^i \right\rangle, x_{p,c} \right) = y_{p,c} \right\}, \quad (1)$$

where eval is the function evaluating a code fix in a test case by matching the output with the intended result. Often, solutions can successfully pass a subset of the test cases but may only cover some corner cases. This allows for a less stringent model evaluation, as strict accuracy may obscure model improvements. Thus, we consider the test case average as soft accuracy and report results for a varying number of generations k.

[8] https://docs.python.org/3/library/py_compile.html

Method	Language	Verdict	BLEU	EM	SM	DM	CB	CA
Naive Copy	Java	✗	**80.28**	0.0	**84.22**	**53.64**	**75.43**	**89.93**
	Python	✗	**68.55**	0.0	**70.12**	**60.51**	**68.47**	**96.56**
PLBART	Java	✗	58.49	0.45	66.92	43.08	57.23	31.36
	Java	✓	59.84	1.46	68.01	44.99	58.62	33.04
	Python	✗	61.89	2.32	64.32	48.81	61.13	91.16
	Python	✓	62.25	2.46	63.31	49.73	62.21	92.21
CodeT5	Java	✗	62.31	**2.96**	74.01	52.30	63.37	63.03
	Java	✓	62.54	2.45	73.93	53.29	63.71	64.23
	Python	✗	64.92	2.74	68.79	56.21	63.53	92.80
	Python	✓	64.67	**2.97**	68.45	56.04	63.28	92.70

Table 3: Match-based results of program repair language models, trained and tested on Java or Python code pairs from our proposed **FixEval** corpus. **EM** (Exact Match), **SM** (Syntax Match), **DM** (Dataflow Match), **CB** (CodeBLEU), and **CA** (Compilation Accuracy).

5 Results

We aim to address the following questions through our preliminary experiments and analysis: **(1)** How well do pretrained Transformer models perform on **FixEval**? and **(2)** How match-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 the research community.

5.1 Pretrained Model Performance

To answer our first research question, we calculated the match-based metrics for our baseline methods, Naive Copy, PLBART, and CodeT5. Table 3 presents the results of all compared models on the match-based metrics described in subsection 4.1.1. 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 match-based measures except for Exact Match (EM). This is because the submitted code pairs will most likely change the program after receiving anything other than an "Accepted" verdict. Between the compared language models, CodeT5 and PLBART, CodeT5 performs better than PLBART across all metrics and programming languages. We believe this is due to their novel identifier-aware pre-training objective that helps CodeT5 learn useful patterns from programming languages.

Further, we observe a marginal performance increase in our baseline models with verdict information as conditional input for Java programs. However, there was no such correlation for Python. We hypothesize that the verbosity of Java has a positive effect when the model is trained with the verdict information, but since Python is not as verbose as Java, the effect may not be the same. We further analyze the impact of verdicts, reporting our results in Section 5.3.

5.2 Match-based vs Execution-based Evaluation Metrics

We compare match-based and execution-based evaluation metrics to check whether both correlate with model performance. To answer our second research question, we evaluate CodeT5 with execution-based metrics and report the results in Table 4a for the sampled test set. While Naive Copy performed the best for match-based metrics (see Table 3), we observe that it performs the worst in both of the execution-based evaluation metrics (see Table 4b). This suggests that TCA and pass@k are better indicators for functional program correctness and evaluate models better than match-based metrics. Further, Figure 3a demonstrates that, with increasing difficulty, TCA decreases, whereas the match-based metrics have no such clear correlation. This means that problems with increasing difficulty become harder to fix, leading to low TCA scores. We speculate that TCA and match-based metrics (BLEU, DM, SM, and CB) do not behave similarly because high match-based similarity does not necessarily indicate program correctness.

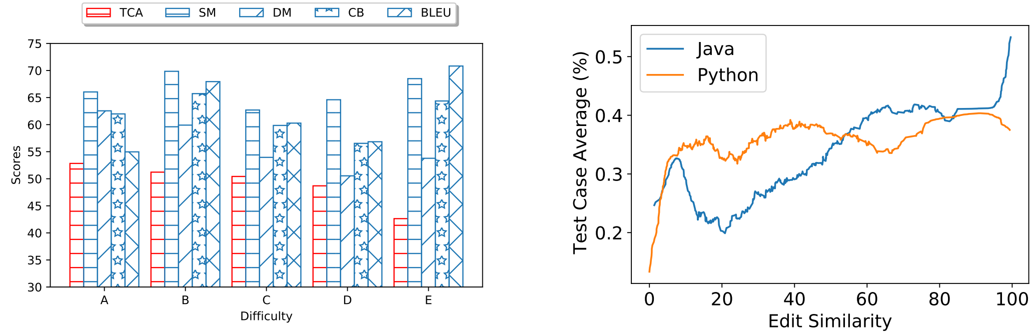
Language	Verdict	pass@k				top-k TCA			
		$k = 1$	$k = 3$	$k = 5$	$k = 10$	$k = 1$	$k = 3$	$k = 5$	$k = 10$
Java	✗	8.65	15.62	19.63	24.44	41.00	34.00	32.70	29.60
Java	✓	**10.94**	**18.77**	**22.66**	**27.96**	**44.99**	**38.80**	**35.87**	**32.90**
Python	✗	6.86	13.07	16.27	20.51	**50.20**	**41.20**	**38.50**	**35.20**
Python	✓	**7.32**	**13.94**	**17.47**	**22.63**	48.75	41.16	38.37	34.88

(a) Execution-based Results for CodeT5.

Language	pass@1	top-1 TCA
Java	0.0	37.95
Python	0.0	41.55

(b) Execution-based Results for Naive Copy.

Table 4: Execution-based evaluation metrics on the sampled evaluation **FIXEVAL**.



(a) Comparison of BLEU and Test Case Average (TCA) with varying problem difficulty.

(b) TCA increases as edit similarity between buggy and reference code increases.

5.3 Ablation Analysis

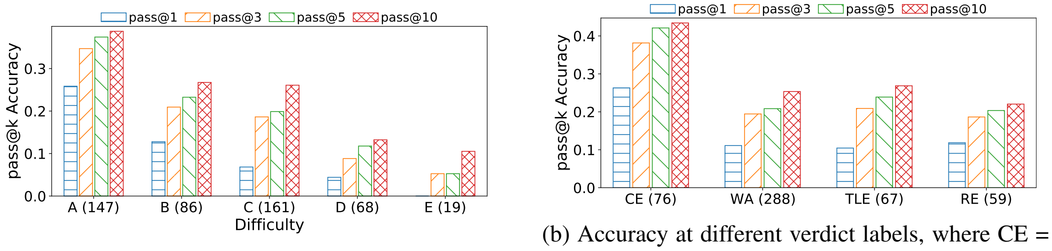
We further perform ablation analyses on **FIXEVAL** to understand the effects of several components, e.g., verdict information, decoding algorithms, etc. The following analyses are based on the CodeT5 with verdicts on sampled Java examples. Qualitative examples are provided in Appendix B.

Correlation to Edit Similarity We analyze model performance based on edit similarity, assuming lower edit similarity between the buggy and fixed code indicates a more difficult problem to fix that error. We sort all data points on our evaluation set based on the buggy and reference (fixed) code's edit similarity and plot the test case average of our best model for both Java and Python. In Figure 3b, 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.

Correlation to Problem Difficulty We analyze the effect of problem difficulty as described in Section 3. From Figure 4a, we observe that as difficulty increases, i.e., problems become harder to solve (difficulty is increasing from A to E), the model performance degrades. At the same time, accuracy increases as we generate more programs for the same input code (pass@1 to pass@10).

Correlation to Evaluation Verdict We analyze the effect of verdict type on performance. Figure 4b 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 even over the entire algorithm.

Effect of Decoding Algorithms We generate candidate fixed programs with various decoding strategies: (i) greedy, (ii) beam search with beam size 10, and (iii) top-k sampling with top-k proba-



(a) Accuracy at different difficulty levels. Task labels A to E indicates increasing difficulty.

(b) Accuracy at different verdict labels, where CE = compilation error, WA = wrong answer, TLE = time limit exceeded, and RE = runtime error.

Figure 4: Pass@k accuracy breakdown based on criteria (a) and (b).

bility and temperature empirically set to 0.95 and 0.7, respectively. Figure 5 shows that beam search decoding usually performs better than greedy and sampling. We also 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 becoming more confident in its predictions. Hence, the temperature doesn't result in substantial model output changes.

Effect of Modeling Verdict We analyze the test example cases successfully repaired only when the model had access to the verdict information. We observe that verdict information is crucial in fixing some instances of buggy code. When we input the same code to program repair models both with and without the verdict, 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 as input can pinpoint the exact location of the error and make the code more consistent. We provide relevant examples in Appendix B, Figure 6.

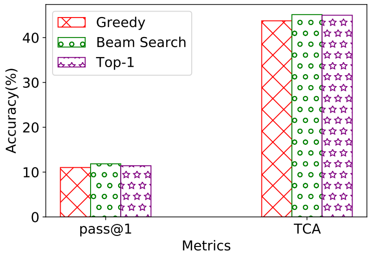


Figure 5: Accuracy sensitivity for strict (pass@1) and soft (top-1 TCA) accuracy for greedy, beam search and top-k sampling decoding algorithms.

Summary of Findings and Our Recommendations We study performance trends concerning edit similarity, problem 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. Choice of decoding strategy produces marginal differences; therefore, it is not a crucial factor in improving bug-fixing models. Our study using verdict will motivate future works to study feedback-based (e.g., feedback from an oracle) approaches to improve bug fixing models.

6 Conclusion

We introduce **FixEval**, a context-aware dataset to improve bug fix model creation and evaluation. In contrast to previous benchmarks that evaluate models with open-source GitHub repositories or programming assignments, we provide a new evaluation corpus that can capture model-generated code's acceptability, accuracy, and efficiency. We assess the performance of state-of-the-art models on this dataset and showcase that traditional evaluation metrics are sub-optimal compared to execution-based metrics derived from test suites that capture contextual program repair requirements often found in practice. Our **FixEval** dataset facilitates several other potential future directions and applications. It can also be used to evaluate the automation of 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 easily extended to other programming languages, such as C++ and JavaScript. We hope that **FixEval** will spur the development of more sophisticated program repair language models that consider realistic program requirements.

Ethics Statement

This work uses language models, for which the risks and potential harms are discussed in Bender & Koller (2020), Brown et al. (2020), Bender et al. (2021) and others. The dataset consists of computer programs that are submitted to online judging sites for competitive programming problems, along with their accompanying test cases and metadata. Any information related to the programmer who wrote these programs has been anonymized. As with all labeled datasets, undesirable biases may be encoded in data. Due to the many technical and practical complexities involved, training generalizable models with no biases cannot be guaranteed in most machine learning applications (though we certainly hope the newly introduced dataset would help assess some of these issues). The rich metadata and diversity of **FixEval** enables model evaluation for several other software engineering tasks. Specifically, **FixEval** can be used to evaluate code completion, code editing, code search, verdict-conditioned code repair, verdict prediction, and chain edit suggestion and can also be extended to other programming languages beyond Java and Python, such as C++ and JavaScript.

Reproducibility

We open-sourced all pretrained models at https://github.com/FixEval/FixEval_official, along with both raw and preprocessed datasets. The repository contains all the necessary scripts to process the dataset from scratch and train the model. The `README.md` file provides detailed explanations on how to run each preprocessing step and train the models. We also checkpoint each section (detailed preprocessing, training, evaluation steps) so that users can quickly reproduce and verify the experimental results.

References

Thomas Ackling, Bradley Alexander, and Ian Grunert. Evolving patches for software repair. In *Proceedings of the 13th annual conference on Genetic and evolutionary computation*, pp. 1427–1434, 2011.

Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. Unified pre-training for program understanding and generation. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pp. 2655–2668, Online, June 2021a. Association for Computational Linguistics. doi: 10.18653/v1/2021.naacl-main.211. URL https://aclanthology.org/2021.naacl-main.211.

Wasi Uddin Ahmad, Md Golam Rahman Tushar, Saikat Chakraborty, and Kai-Wei Chang. Avatar: A parallel corpus for java-python program translation. *arXiv preprint arXiv:2108.11590*, 2021b.

Aizu Online Judge, 2004. https://judge.u-aizu.ac.jp/onlinejudge.

Andrea Arcuri. On the automation of fixing software bugs. In *Companion of the 30th International Conference on Software Engineering*, ICSE Companion '08, pp. 1003–1006, New York, NY, USA, 2008. Association for Computing Machinery. ISBN 9781605580791. doi: 10.1145/1370175.1370223. URL https://doi.org/10.1145/1370175.1370223.

Atcoder. Atcoder opensourced test cases. https://www.dropbox.com/sh/nx3tnilzqz7df8a/AAAYlTq2tiEHl5hsESw6-yfLa?dl=0, 2020.

Emily M Bender and Alexander Koller. Climbing towards nlu: On meaning, form, and understanding in the age of data. In *Proceedings of the 58th annual meeting of the association for computational linguistics*, pp. 5185–5198, 2020.

Emily M Bender, Timnit Gebru, Angelina McMillan-Major, and Shmargaret Shmitchell. On the dangers of stochastic parrots: Can language models be too big? In *Proceedings of the 2021 ACM Conference on Fairness, Accountability, and Transparency*, pp. 610–623, 2021.

Berkay Berabi, Jingxuan He, Veselin Raychev, and Martin Vechev. Tfix: Learning to fix coding errors with a text-to-text transformer. In *International Conference on Machine Learning*, pp. 780–791. PMLR, 2021.

Tom Britton, Lisa Jeng, Graham Carver, and Paul Cheak. Reversible debugging software "quantify the time and cost saved using reversible debuggers". 2013.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

Favio DeMarco, Jifeng Xuan, Daniel Le Berre, and Martin 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*, pp. 30–39, 2014.

Elizabeth Dinella, Hanjun Dai, Ziyang Li, Mayur Naik, Le Song, and Ke Wang. Hoppity: Learning graph transformations to detect and fix bugs in programs. In *International Conference on Learning Representations (ICLR)*, 2020.

Yangruibo Ding, Baishakhi Ray, Premkumar Devanbu, and Vincent J Hellendoorn. Patching as translation: the data and the metaphor. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 275–286. IEEE, 2020.

Luca Gazzola, Daniela Micucci, and Leonardo Mariani. Automatic software repair: A survey. *IEEE Transactions on Software Engineering*, 45(1):34–67, 2017.

Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish Shevade. Deepfix: Fixing common c language errors by deep learning. In *Thirty-First AAAI Conference on Artificial Intelligence*, 2017.

Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, et al. Measuring coding challenge competence with apps. *arXiv preprint arXiv:2105.09938*, 2021.

Yang Hu, Umair Z. Ahmed, Sergey Mechtaev, Ben Leong, and Abhik Roychoudhury. Refactoring based program repair applied to programming assignments. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 388–398, 2019. doi: 10.1109/ASE.2019.00044.

Faria Huq, Masum Hasan, Md Mahim Anjum Haque, Sazan Mahbub, Anindya Iqbal, and Toufique Ahmed. Review4repair: Code review aided automatic program repairing. *Information and Software Technology*, 143:106765, 2022.

René Just, Darioush Jalali, and Michael 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*, pp. 437–440, 2014.

Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. Automatic patch generation learned from human-written patches. In *2013 35th International Conference on Software Engineering (ICSE)*, pp. 802–811. IEEE, 2013.

Sumith Kulal, Panupong Pasupat, Kartik Chandra, Mina Lee, Oded Padon, Alex Aiken, and Percy S Liang. Spoc: Search-based pseudocode to code. *Advances in Neural Information Processing Systems*, 32, 2019.

Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *2012 34th International Conference on Software Engineering (ICSE)*, pp. 3–13, 2012. doi: 10.1109/ICSE.2012.6227211.

Claire Le Goues, Neal Holtschulte, Edward K. Smith, Yuriy Brun, Premkumar Devanbu, Stephanie Forrest, and Westley Weimer. The ManyBugs and IntroClass benchmarks for automated repair of C programs. *IEEE Transactions on Software Engineering (TSE)*, 41(12):1236–1256, December 2015. ISSN 0098-5589. doi: 10.1109/TSE.2015.2454513. DOI: 10.1109/TSE.2015.2454513.

Claire Le Goues, Michael Pradel, and Abhik Roychoudhury. Automated program repair. *Communications of the ACM*, 62(12):56–65, 2019.

Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Ves Stoyanov, and Luke Zettlemoyer. Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. *arXiv preprint arXiv:1910.13461*, 2019.

Derrick Lin, James Koppel, Angela Chen, and Armando Solar-Lezama. Quixbugs: A multi-lingual program repair benchmark set based on the quixey challenge. In *Proceedings Companion of the 2017 ACM SIGPLAN international conference on systems, programming, languages, and applications: software for humanity*, pp. 55–56, 2017.

I Loshchilov and F Hutter. Decoupled weight decay regularization, 7th international conference on learning representations, iclr. *New Orleans, LA, USA, May*, (6-9):2019, 2019.

Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, et al. Codexglue: A machine learning benchmark dataset for code understanding and generation. *arXiv preprint arXiv:2102.04664*, 2021.

Gayle Laakmann McDowell. *Cracking the Coding Interview: 189 Programming Questions and Solutions*. CareerCup, 2019.

Tom Mens. On the complexity of software systems. *Computer*, 45(08):79–81, 2012.

Ali Mesbah, Andrew Rice, Emily Johnston, Nick Glorioso, and Edward Aftandilian. Deepdelta: learning to repair compilation errors. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 925–936, 2019.

National Institute of Standards and Technology. The economic impacts of inadequate infrastructure for software testing. *U.S. Department of Commerce Technology Administration*, 2002.

Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, pp. 311–318, 2002.

Ruchir Puri, David S Kung, Geert Janssen, Wei Zhang, Giacomo Domeniconi, Vladmir Zolotov, Julian Dolby, Jie Chen, Mihir Choudhury, Lindsey Decker, et al. Project codenet: a large-scale ai for code dataset for learning a diversity of coding tasks. *ArXiv. Available at https://arxiv. org/abs*, 2105, 2021.

Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *arXiv preprint arXiv:1910.10683*, 2019.

Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. Codebleu: a method for automatic evaluation of code synthesis. *arXiv preprint arXiv:2009.10297*, 2020.

Hyunmin Seo, Caitlin Sadowski, Sebastian Elbaum, Edward Aftandilian, and Robert Bowdidge. Programmers' build errors: a case study (at google). In *Proceedings of the 36th International Conference on Software Engineering*, pp. 724–734, 2014.

Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053*, 2019.

Michele Tufano, Jevgenija Pantiuchina, Cody Watson, Gabriele Bavota, and Denys Poshyvanyk. On learning meaningful code changes via neural machine translation. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pp. 25–36. IEEE, 2019a.

Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. An empirical study on learning bug-fixing patches in the wild via neural machine translation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 28(4):1–29, 2019b.

Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv preprint arXiv:2109.00859*, 2021.

Michihiro Yasunaga and Percy Liang. Break-it-fix-it: Unsupervised learning for program repair. In *International Conference on Machine Learning (ICML)*, 2021.

A LIMITATIONS

There are a few limitations of the dataset presented in this work. In our experiments, we manually analyze the mistakes made by models with and without access to verdict information. We find that common model mistakes include additions of code segments that are unnecessary to fix a given bug. We believe this is due to insufficient information about the bug and the problem. Another limitation of this work is that, while the code bug fix pair dataset can be extended for other programming languages, code and test suite expansion for more problems is solely dependent upon AtCoder availability. Also, the execution-based evaluation metric calculation requires substantial amount of time to evaluate a single program. To reduce computation complexity, we select two datapoints at random per problem from the test dataset. Nevertheless, parallelism and high performance computing can be utilized towards making execution-based evaluation faster and more efficient for large-scale data. In terms of research insights, our dataset incorporates Java and Python bugs, and may not generalize to other programming languages. Further research is necessary to evaluate the impact of FixEval for analyzing deep learning models for additional languages. Additionally, competitive programs submitted online may not accurately reflect real-world software bugs from professional developers. More work is necessary to develop benchmarks that simulate authentic software programs to evaluate deep learning models for automated program repair.

B FURTHER DETAILS OF OUR ANALYSIS

B.1 DESCRIPTION OF TEST SUITE COLLECTION

On the AtCoder website we found the names of each contest.[9] Contests are named in the format of ABC123, AGC123, or ARC123 to indicate the AtCoder Beginner Contest, AtCoder Grand Contest, or AtCoder Regular Contest, followed by the specific contest number (i.e. 123 in this example) to create a unique identifier for a given problem. The name and contest number are also stored on CodeNet, where we also get the exact same name and the contest number with problems sorted by difficulty. We then match these to retreive the test cases in addition to input and expected output values from the DropBox link to create the test suite for **FIXEVAL**.

B.2 LIST OF VERDICTS

The following is a list of all the possible verdict outcomes for submitted competitive programs with a brief description:

- **Accepted (AC):** Passed all test cases.
- **Wrong Answer (WA):** Failed one or more test cases.
- **Compile Error (CE):** Program did not compile.
- **Runtime Error (RE):** Program execution was not successful.
- **Presentation Error (PE):** Output is correct, but it is not formatted in the proper way.
- **Time Limit Exceeded (TLE):** The program did not run within the intended time limit.
- **Memory Limit Exceeded (MLE):** The program did not run within the intended memory limit.
- **Output Limit Exceeded (OLE):** Program tried to write too much information.
- **Waiting for Judging (WJ):** Judge is busy.
- **Waiting for Re-judging (WR):** Waiting for Judge to run the tests again.
- **Judge Not Available (JNA):** Error encountered by Judge.
- **Internal Error (IE):** Judge encountered an error or the problem setter's configuration is incorrect.

[9] https://atcoder.jp/contests/

Buggy Program (verdict: Wrong Answer)

```java
import java.util.*;
import java.lang.*;
public class Main {
  public static void main(String[] args){
    Scanner sc = new Scanner(System.in);
    int a = sc.nextInt();
    int b = sc.nextInt();
    long ans = a*b/gcd(a, b);
    System.out.println(ans);
    sc.close();
  }
  public static long gcd(long m,long n){
    if (m < n) return gcd(n, m);
    if (n==0) return m;
    return gcd(n, m % n);
  }
}
```

Fixed Program

```java
import java.util.*;
import java.lang.*;
public class Main{
  public static void main(String[] args){
    Scanner sc = new Scanner(System.in);
    long a = sc.nextInt();
    long b = sc.nextInt();
    long ans = a*b/gcd(a, b);
    System.out.println(ans);
    sc.close();
  }
  public static long gcd(long m,long n){
    if (m < n) return gcd(n, m);
    if (n==0) return m;
    return gcd(n, m % n);
  }
}
```

Buggy Program (verdict: Compilation Error)

```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();
    if((A - B) % 2 == 0){
      System.out.println((A + B)/2);
    }
    else {
      System.out.println("IMPOSSIBLE");
    }
  }
}
```

Fixed Program

```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();
    if((a-b) % 2 == 0) {
      System.out.println((a+b)/2);
    }
    else {
      System.out.println("IMPOSSIBLE");
    }
  }
}
```

Figure 6: Java examples from the sampled FIXEVAL test set that were only successfully solved with the verdict information as conditional model input

B.3 Java Examples Solved only by the model with verdict in Figure 6

Model learned to cast the output

Buggy Program in Python

```python
n = int(input())
n = print(int(n-1+1)*(n-1)/2)
```

Fixed Program in Python

```python
n = int(input())
n = print(int((n-1+1)*(n-1)/2))
```

Model learned add sort function correctly

Buggy Program in Python

```python
n = int(input())
a = list(map(int,input().split())).sort()
print(a[-1]-a[0])
```

Fixed Program in Python

```python
n = int(input())
a = list(map(int,input.split()))
a.sort()
print(a[-1]-a[0])
```

Model Learned to change the comparison sign

Buggy Program in Python

```python
N, K = map(int, input().split())
h = input().split()
c = 0
for i in range(N):
  if int(h[i]) > k:
    c += 1
print(c)
```

Fixed Program in Python

```python
N, K = map(int, input().split())
h = input().split()
c = 0
for i in range(N):
  if int(h[i]) > k:
    c += 1
print(c)
```

Model Learned to change loop range for corrcetion

Buggy Program in Python

```python
from collections import defaultdict
N = int(input())
d = defaultdict(int)
for n in range(N):
  n = str(n)
  a = n[0]
  b = n[- 1]
  d[(a,b)] += 1
s = 0
for i in range(1, 10):
  for j in range(10):
    s += d[(str(j), str(i))] * d[(str(i),
     str(j))]
print(s)
```

Fixed Program in Python

```python
from collections import defaultdict
N = int(input())
d = defaultdict(int)
for n in range(N+1):
  n = str(n)
  a = n[0]
  b = n[- 1]
  d[(a,b)]+=1
s = 0
for i in range(1,10):
  for j in range(10):
    s += d[(str(j), str(i))] * d[(str(i),
     str(j))]
print(s)
```

Figure 7: Some Python examples which were also successfully generated by our model when we pass the buggy code to it. The errors are marked with red

B.4 Generated Python code examples with the CodeT5 model in Figure 7

Model learned to copy entire line out of scope

Buggy Program (Wrong Answer)

```java
import java.util.*;
public class Main {
  public static void main(String[] args){
    Scanner sc = new Scanner(System.in);
    int D = sc.nextInt();
    int N = sc.nextInt();
    int a;
    a = (int) Math.pow(100, D )*N;
    if (N == 100){
      a = (int) Math.pow(100, D)*(N+1);
      System.out.println(a);
    }
  }
}
```

Fixed Program

```java
import java.util.*;
public class Main {
  public static void main(String[] args){
    Scanner sc = new Scanner(System.in);
    int D = sc.nextInt();
    int N = sc.nextInt();
    int a;
    a = (int) Math.pow(100, D )*N;
    if (N == 100){
      a = (int) Math.pow(100, D)*(N+1);
    }
    System.out.println(a);
  }
}
```

Model learned change logic from ”or” to ”and”

Buggy Program (Compilation Error)

```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();
    if ( a <= 8 || b <= 8 ) {
      System.out.println(”Yay!”);
    }
    else {
      System.out.println(”:(”);
    }
  }
}
```

Fixed Program

```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();
    if ( a <= 8 && b <= 8 ) {
      System.out.println(”Yay!”);
    }
    else {
      System.out.println(”:(”);
    }
  }
}
```

Model Learned to change return statement to a print statement

Buggy Program (Compilation Error)

```java
import java.util.*;
public class Main {
  public static void main(String[] args){
    Scanner sc = new Scanner(System.in);
    int a = sc.nextInt();
    int a2 = a * a;
    int a3 = a2 * a;
    return a + a2 + a3;
  }
}
```

Fixed Program

```java
import java.util.*;
public class Main {
  public static void main(String[] args){
    Scanner sc = new Scanner(System.in);
    int a = sc.nextInt();
    int a2 = a * a;
    int a3 = a2 * a;
    System.out.println( a + a2 + a3 );
  }
}
```

Model Learned to change syntax and compilation error

Buggy Program (Compilation Error)

```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

```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);
  }
}
```

Figure 8: Java examples from the sampled FIXEVAL test set that were only successfully solved with the verdict information as conditional model input

B.5 Generated Java code examples with the CodeT5 model in Figure 8