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

Md Mahim Anjum Haque
Virginia Tech
mahim@vt.edu

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

Ismini Lourentzou
Virginia Tech
ilourentzou@vt.edu

Chris Brown
Virginia Tech
dcbrown@vt.edu

Abstract

Source code repositories consist of large codebases, often containing error-prone programs. The increasing complexity of software has led to a drastic rise in time and costs for identifying and fixing these defects. Various methods exist to automatically generate fixes for buggy code. However, due to the large combinatorial space of possible solutions for a particular bug, there are not many tools and datasets available to evaluate generated code effectively. In this work, we introduce **FIXEVAL**, a benchmark comprising of buggy code submissions to competitive programming problems and their respective fixes. **FIXEVAL** contains a rich test suite to evaluate and assess the correctness of model-generated program fixes, in addition to 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, while execution-based methods evaluate programs through all cases and scenarios specifically designed for that solution. Therefore, we believe **FIXEVAL** provides a step towards real-world automatic bug fixing and model-generated code evaluation. Dataset and models are open-sourced at https://github.com/mahimanzum/FixEval.

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 solved. Automated program repair is an active area of research[1] 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 increasingly applied statistical and neural methods to automate program repair tasks in recent years. 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).

---

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

While a growing number of approaches are being studied to automate program repair, there is limited support for evaluating generated fixes. Prior work, such as Tfix (Berabi et al., 2021), BIFI (Yasunaga and 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-based methods are unable to 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. To assess their quality and functional correctness, a generated fix is considered to be 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). **FIXEVAL** contains solutions to 700 Python and Java problems along with more than $40,000$ test cases for evaluation. 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, deriving **FIXEVAL** from programming competitions provides numerous parallel pairs of buggy and corrected programs, as well as access to corresponding unit tests. 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 **FIXE-VAL**, a novel *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 version of 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 is limited research on the application of language modeling in the field of 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 on 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, while 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,[2] 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. We explore several sampling parameters

---

[2]https://eslint.org/

Table 1: A comparison between FIXEVAL and other existing code repair datasets for machine learning

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

CR and CE indicate code review comments and compilation errors, respectively.

while performing program generation and empirically show that larger language models with good initialization trained on code can improve performance by a large factor, but are still far from solving the task of program repairing.

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 recent benchmarks used to evaluate machine learning techniques for bug fixing is 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 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 and hence, cross-function dependencies cannot be modeled. Further, Bug2Fix also lacks unit tests to check for function correctness. The GitHub-Python dataset (Yasunaga and 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 the 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. However, there are several differences between **FIXEVAL** and these datasets. First, **FIXEVAL** is substantially larger than both datasets consisting of more lines of code (QuixBugs: 1,034; Defects4J: 321,000; FixEval: 54 Million in Java and 61 Million in 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. **FIXEVAL** is more advanced than existing machine learning benchmarks containing introductory programming assignments that fail to capture the representation of large real world bugs and by providing a thorough test suite to evaluate repairs on efficiency and correctness, which is more effective than syntax matching. Further, existing automated program repair datasets focus on domain-specific open source code without taking into consideration additional constraints and contexts that vary from project to project. **FIXEVAL** is the first context-aware program repair evaluation dataset with a comprehensive test suite that also takes runtime and memory consumption into consideration.

Table 2: Example submissions from the FIXEVAL dataset

Problem Statement[5]: 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: $1 \le A, B, T \le 20$, All input values are integers Time Limit: 2 secs, Memory Limit: 1024MB, Problem Difficulty: A

Java Buggy and Fixed Code pair

Java Buggy Code:
```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);
  }
}
```

Java Fixed Code:
```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);
  }
}
```

* Errors are marked in red.

3   Dataset: FIXEVAL

The **FIXEVAL** dataset consists of Java and Python program submissions from CodeNet (Puri et al., 2021), a collection of programs of varying difficulty levels 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 is created for specific problems, enabling rigorous evaluation of program functionality. While competitive programming data is not an exact reflection of real-world professional software development environments, **FIXEVAL** consists of unit tests and takes time and memory requirements into consideration which is common for evaluating software engineers to hire (McDowell, 2019) and crucial for writing efficient code in industrial settings (Mens, 2012).

Dataset Construction   For each user, we consider 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 C.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 pair each wrong submission with the accepted submission and consider that as a single data point consisting of a pair of a buggy and a fixed program. Thus, for a submission path of length n, there exist $n - 1$ collected data points. A sample datapoint can be viewed in Table 2. **FIXEVAL** consists of all such submission paths for the corresponding problems in both Java and Python for all 154k users and 6.5 million submission paths.

We find that users submit 90 programs on average from an individual account. We de-duplicate the submissions using Jaccard similarity to remove multiple submissions for a specific problem. We use the $javalang$[3] tokenizer for Java and the $tokenizer$[4] of the standard library for Python. As shown in Table 3, we create 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. We also verify that the problem difficulty distribution stays the same in the test set and further ensure that there is an even difficulty of problems partitioned in all splits by stratified sampling. Finally, we make sure that the test and validation data have all the test cases required for evaluating the programs.

---

[3]https://github.com/c2nes/javalang
[4]https://docs.python.org/3/library/tokenize.html
[5]https://atcoder.jp/contests/abc125/tasks/abc125_a

Table 3: Statistics of the **FIXEVAL** dataset.

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

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[6] site. To construct our dataset test suite, we match the problem names from the CodeNet problem metadata with the AtCoder published website. Then, we match each problem with the corresponding input and output files provided with the test cases. We also clean 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 keep 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, which consequently results in many combinatorial outputs to be equivalently valid. We remove such problems and test cases from our evaluation pipeline as they require more complex algorithms to receive an Accepted Verdict. More details are available in Appendix C.1.

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 of the test cases is manually generated by domain experts (e.g., problem setters for competitive programming challenges) to ensure the functional correctness of the submitted programs. Furthermore, each of the test cases specializes in testing a program in individual extremes. For example, some cases stress-test the memory used, while others test the time taken by the program in a worst-case scenario.

Dataset Difficulty Our **FIXEVAL** dataset has an average of 331 tokens for Java and 236 tokens for Python programs, which is larger than the existing program repair datasets shown in Table 1. Due to the vast combinatorial search spaces of programs, it is difficult to find a correct solution by chance without a deep understanding of the task. Each of the problems come with their predefined difficulty measure set by the official AtCoder website[7]. We retain the same order containing 5 difficulty categories A, B, C, D, E where A is the easiest and E is the hardest problem. The distribution of problem difficulty from the overall dataset is presented in Figure 1.
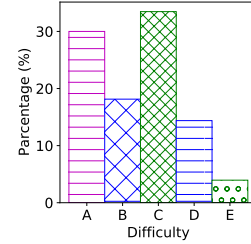

Figure 1: Test set Difficulty

4 Experiment Setup

We fine-tune state-of-the-art Transformer language models pre-trained on programming languages using our proposed dataset and evaluate on the program repair task. We feed the buggy and reference (correct) program into the encoder and decoder of the Transformer model, respectively. In addition, we perform experiments by appending the verdict to the buggy program to examine whether the verdict message helps the model to correct the errors in the buggy program.

4.1 Baselines and 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 both unimodal (code only) and bimodal (code text pairs) data for pretraining.

---

[6]https://atcoder.jp/posts/21
[7]https://atcoder.jp/

Apart from the two pretrained 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 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 and Hutter, 2019), $5 \times e^{-5}$ learning rate, early stopping with patience set to 3 and 100 warm-up steps.

4.2 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.

4.2.1 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).

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 ground truth 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 the ground truth reference data-flows.

4.2.2 Execution-based Metrics

In program repair tasks, typically the input and output have high lexical overlapping. As a result, match-based metrics, such as, BLEU and CodeBLEU, may not estimate the true accuracy of model-generated program fixes. On the other hand, a program can be fixed in multiple ways that differ from the reference program. Therefore, we argue that the Exact Match metric is not ideal for program repair evaluation. Tfix (Berabi et al., 2021) introduces error removal to assess if a program fix is compilable. However, compilability does not imply functional correctness. Therefore, we introduce execution-based evaluation that alleviates these limitations. We collect a comprehensive suite of test cases and evaluate functional correctness of model generated program fixes by executing them to check if they pass the unit tests.

Evaluating all generated programs on execution for all available test cases is a memory intensive and time-consuming process. To reduce time complexity, we randomly selected two datapoints per test problem from the test dataset, with similar distribution of error verdicts, to make sure 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 the purpose of 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 $k \leq n$

---

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

Table 4: Match-based results of program repair language models, trained and tested on Java or Python code pairs from our proposed FIXEVAL corpus.

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

EM (Exact Match), **SM** (Syntax Match), **DM** (Dataflow Match), **CB** (CodeBLEU), and **CA** (Compilation Accuracy).

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 a strict measure as a code repair is considered unsuccessful if there exists a single failed test case.

Test Case Average (TCA@k): We follow Hendrycks et al. (2021) to compute the average fraction 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

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

where eval is the function evaluating a code fix in a test case by matching the output with the intended result. Oftentimes, solutions can successfully pass a subset of the test cases but may not cover every corner case. This allows for a less stringent model evaluation, as strict accuracy may currently obscure model improvements. Thus, we consider test case average as soft accuracy and report results for varying number of generations k.

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 4 presents the results of all compared models on the match-based metrics described in subsection 4.2.1. 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 terms of all match-based measures except for Exact Match (EM). This is due to the fact that the submitted code pairs will most likely have changes to the program after

Table 5: Execution-based evaluation metrics on the sampled evaluation FIXEVAL

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	TCA
Java	0.0	37.95
Python	0.0	41.55

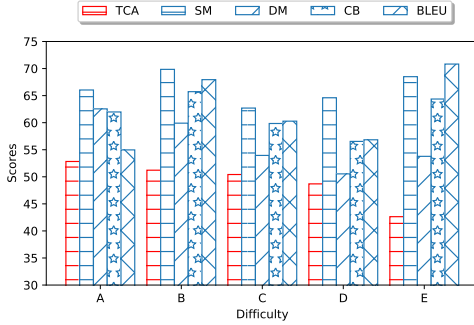(b) Execution-based Results for Naive Copy

Figure 2: Comparison of BLEU and Test Case Average (TCA) with varying problem difficulty.
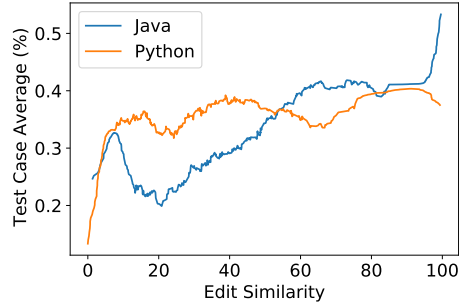

Figure 3: TCA increases as edit similarity between buggy and reference code increases for both Java and Python.

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 the extensive pre-training tasks that help CodeT5 in learning useful patterns from programming languages. Further, we observe slight performance increase of 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 perform analysis on 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 the top pretrained Transformer language model, CodeT5, with execution-based metrics and compare with the match-based metrics across varying problem difficulty levels. We report the execution-based results for CodeT5 in Table 5a for the sampled test set. While Naive Copy performed the best for match-based metrics, we observe that it performs the worst in both of the execution-based evaluation metrics (see Table 5b). This suggests that TCA and pass@k are better indicators for functional program correctness and evaluate models better than match-based metrics. Further, Figure 2 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, and hence lead 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.

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 data. Examples of model-generated code are available in Appendix C.

Effect of 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 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.

Effect of Problem Difficulty   We analyze the effect of increasing problem difficulty. In Figure 4, 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).
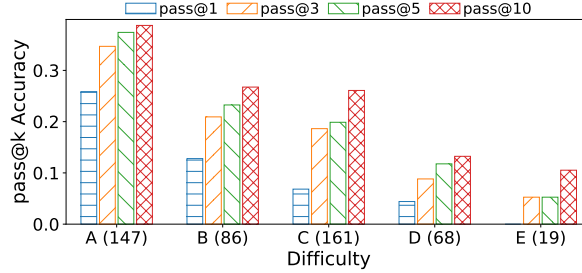

Figure 4: Effect of increasing problem difficulty in pass@k model performance.

Effect of 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 even over the entire algorithm.
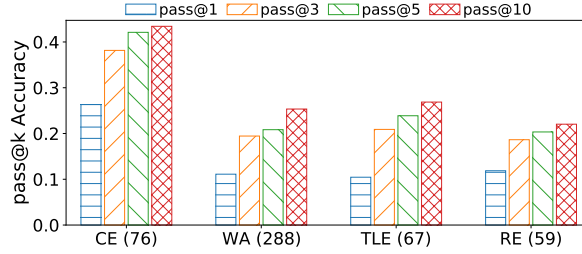

Figure 5: pass@k accuracy when varying verdict type. **CE** (Compilation Error), **WA** (Wrong Answer), **TLE** (Time Limit Exceeded), **RE** (Runtime Error).

Effect of Decoding Algorithms   We use our best performing model to 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 probability and temperature empirically set to 0.95 and 0.7, respectively. Figure 6 shows that beam search decoding achieves the best performance. We notice that beam search decoding usually performs better than both greedy and sampling. We also experiment with varying temperature from 0.2 to 1.2 while sampling and observe negligible 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 in turn results in the model becoming more confident on its predictions and hence temperature does not result in substantial model output changes.
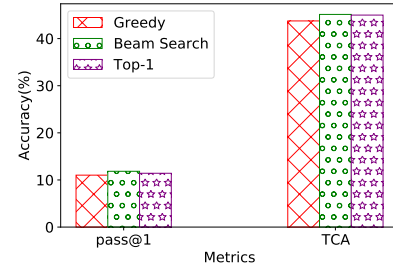

Figure 6: Accuracy sensitivity for strict (pass@1) and soft (top-1 TCA) accuracy for greedy, beam search and top-k sampling decoding algorithms.

Impact of Verdict Message   We analyze the test example cases that were 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, whereas the model

9

with verdict information as input is able to pinpoint the exact location of the error and make the code more consistent. We provide a relevant examples in Appendix C.3 Table 6.

We study performance trends with respect to edit similarity, problem difficulty, and evaluation verdict to show that some bug fixing tasks are trivial or easy while many of them are challenging. Therefore, we encourage future work to consider all aforementioned aspects while performing evaluation. Choice of decoding strategy produces marginal differences; therefore, it is not a crucial factor in improving bug-fixing models. We hope that our study on using verdict messages will motivate future works to study feedback-based (e.g., feedback from an oracle) approaches to improve bug fixing models.

6 Limitations

There are several limitations to the work presented in this research. 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.

7 Conclusion

We introduce **FIXEVAL**, a context-aware dataset to improve bug fix model creation and evaluation. In contrast to prior benchmarks that evaluate models with open-source GitHub repositories or programming assignments, we provide a new evaluation corpus that is able to capture the acceptability, accuracy, and efficiency of model-generated code. We assess the performance of state-of-the-art models on this dataset and showcase that traditional evaluation metrics are sub-optimal evaluation methods 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, and can also be used to evaluate the automation of different 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 take into consideration realistic code requirements.

References

Ackling, T., Alexander, B., Grunert, I., 2011. Evolving patches for software repair, in: Proceedings of the 13th annual conference on Genetic and evolutionary computation, pp. 1427–1434.

Ahmad, W., Chakraborty, S., Ray, B., Chang, K.W., 2021a. 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, Association for Computational Linguistics, Online. pp. 2655–2668. URL: https://aclanthology.org/2021.naacl-main.211, doi:10.18653/v1/2021.naacl-main.211.

Ahmad, W.U., Tushar, M.G.R., Chakraborty, S., Chang, K.W., 2021b. Avatar: A parallel corpus for java-python program translation. arXiv preprint arXiv:2108.11590 .

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

Arcuri, A., 2008. On the automation of fixing software bugs, in: Companion of the 30th International Conference on Software Engineering, Association for Computing Machinery, New York, NY, USA. p. 1003–1006. URL: https://doi.org/10.1145/1370175.1370223, doi:10.1145/1370175.1370223.

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

Berabi, B., He, J., Raychev, V., Vechev, M., 2021. Tfix: Learning to fix coding errors with a text-to-text transformer, in: International Conference on Machine Learning, PMLR. pp. 780–791.

Britton, T., Jeng, L., Carver, G., Cheak, P., 2013. Reversible debugging software "quantify the time and cost saved using reversible debuggers" .

Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J.D., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., et al., 2020. Language models are few-shot learners. Advances in neural information processing systems 33, 1877–1901.

Chen, M., Tworek, J., Jun, H., Yuan, Q., Pinto, H.P.d.O., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., et al., 2021. Evaluating large language models trained on code. arXiv preprint arXiv:2107.03374 .

DeMarco, F., Xuan, J., Le Berre, D., Monperrus, M., 2014. 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.

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

Ding, Y., Ray, B., Devanbu, P., Hellendoorn, V.J., 2020. Patching as translation: the data and the metaphor, in: 2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE), IEEE. pp. 275–286.

Gazzola, L., Micucci, D., Mariani, L., 2017. Automatic software repair: A survey. IEEE Transactions on Software Engineering 45, 34–67.

Gupta, R., Pal, S., Kanade, A., Shevade, S., 2017. Deepfix: Fixing common c language errors by deep learning, in: Thirty-First AAAI Conference on Artificial Intelligence.

Hendrycks, D., Basart, S., Kadavath, S., Mazeika, M., Arora, A., Guo, E., Burns, C., Puranik, S., He, H., Song, D., et al., 2021. Measuring coding challenge competence with apps. arXiv preprint arXiv:2105.09938 .

Hu, Y., Ahmed, U.Z., Mechtaev, S., Leong, B., Roychoudhury, A., 2019. Re-factoring based program repair applied to programming assignments, in: 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 388–398. doi:10.1109/ASE.2019.00044.

Huq, F., Hasan, M., Haque, M.M.A., Mahbub, S., Iqbal, A., Ahmed, T., 2022. Review4repair: Code review aided automatic program repairing. Information and Software Technology 143, 106765.

Just, R., Jalali, D., Ernst, M.D., 2014. Defects4j: A database of existing faults to enable controlled testing studies for java programs, in: Proceedings of the 2014 International Symposium on Software Testing and Analysis, pp. 437–440.

Kim, D., Nam, J., Song, J., Kim, S., 2013. Automatic patch generation learned from human-written patches, in: 2013 35th International Conference on Software Engineering (ICSE), IEEE. pp. 802–811.

Kulal, S., Pasupat, P., Chandra, K., Lee, M., Padon, O., Aiken, A., Liang, P.S., 2019. Spoc: Search-based pseudocode to code. Advances in Neural Information Processing Systems 32.

Le Goues, C., Dewey-Vogt, M., Forrest, S., Weimer, W., 2012. 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. doi:10.1109/ICSE.2012.6227211.

Le Goues, C., Holtschulte, N., Smith, E.K., Brun, Y., Devanbu, P., Forrest, S., Weimer, W., 2015. The ManyBugs and IntroClass benchmarks for automated repair of C programs. IEEE Transactions on Software Engineering (TSE) 41, 1236–1256. doi:10.1109/TSE.2015.2454513. DOI: 10.1109/TSE.2015.2454513.

Le Goues, C., Pradel, M., Roychoudhury, A., 2019. Automated program repair. Communications of the ACM 62, 56–65.

Lewis, M., Liu, Y., Goyal, N., Ghazvininejad, M., Mohamed, A., Levy, O., Stoyanov, V., Zettlemoyer, L., 2019. Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. arXiv preprint arXiv:1910.13461 .

Lin, D., Koppel, J., Chen, A., Solar-Lezama, A., 2017. 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.

Loshchilov, I., Hutter, F., 2019. " decoupled weight decay regularization", 7th international conference on learning representations, iclr. New Orleans, LA, USA, May , 2019.

Lu, S., Guo, D., Ren, S., Huang, J., Svyatkovskiy, A., Blanco, A., Clement, C., Drain, D., Jiang, D., Tang, D., et al., 2021. Codexglue: A machine learning benchmark dataset for code understanding and generation. arXiv preprint arXiv:2102.04664 .

McDowell, G.L., 2019. Cracking the Coding Interview: 189 Programming Questions and Solutions. CareerCup.

Mens, T., 2012. On the complexity of software systems. Computer 45, 79–81.

Mesbah, A., Rice, A., Johnston, E., Glorioso, N., Aftandilian, E., 2019. 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.

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

Papineni, K., Roukos, S., Ward, T., Zhu, W.J., 2002. 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.

Puri, R., Kung, D.S., Janssen, G., Zhang, W., Domeniconi, G., Zolotov, V., Dolby, J., Chen, J., Choudhury, M., Decker, L., et al., 2021. Project codenet: a large-scale ai for code dataset for learning a diversity of coding tasks. ArXiv. Available at https://arxiv. org/abs 2105.

Raffel, C., Shazeer, N., Roberts, A., Lee, K., Narang, S., Matena, M., Zhou, Y., Li, W., Liu, P.J., 2019. Exploring the limits of transfer learning with a unified text-to-text transformer. arXiv preprint arXiv:1910.10683 .

Ren, S., Guo, D., Lu, S., Zhou, L., Liu, S., Tang, D., Sundaresan, N., Zhou, M., Blanco, A., Ma, S., 2020. Codebleu: a method for automatic evaluation of code synthesis. arXiv preprint arXiv:2009.10297 .

Seo, H., Sadowski, C., Elbaum, S., Aftandilian, E., Bowdidge, R., 2014. Programmers' build errors: a case study (at google), in: Proceedings of the 36th International Conference on Software Engineering, pp. 724–734.

Shoeybi, M., Patwary, M., Puri, R., LeGresley, P., Casper, J., Catanzaro, B., 2019. Megatron-lm: Training multi-billion parameter language models using model parallelism. arXiv preprint arXiv:1909.08053 .

Tufano, M., Pantiuchina, J., Watson, C., Bavota, G., Poshyvanyk, D., 2019a. On learning meaningful code changes via neural machine translation, in: 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), IEEE. pp. 25–36.

Tufano, M., Watson, C., Bavota, G., Penta, M.D., White, M., Poshyvanyk, D., 2019b. An empirical study on learning bug-fixing patches in the wild via neural machine translation. ACM Transactions on Software Engineering and Methodology (TOSEM) 28, 1–29.

Wang, Y., Wang, W., Joty, S., Hoi, S.C., 2021. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. arXiv preprint arXiv:2109.00859 .

Yasunaga, M., Liang, P., 2021. Break-it-fix-it: Unsupervised learning for program repair, in: International Conference on Machine Learning (ICML).

A   Checklist

A.1   For all authors...

(a) Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope?   [Yes] Please see Sections 2, 3, and 4.

(b) Did you describe the limitations of your work?   [Yes] Please see the paragraph on Limitations in the supplementary materials.

(c) Did you discuss any potential negative societal impacts of your work?   [N/A]. We believe that this dataset does not have any negative societal impact. On the contrary, it improves how we think about evaluating code in the field of AI.

(d) Have you read the ethics review guidelines and ensured that your paper conforms to them? [Yes] The dataset does no harm to living beings and does not raise any security and economic concerns, human rights and surveillance issues, damages to the environment, or deception and damage people's livelihood. We have anonymized each submitter's user id and tried filtering offensive words. We have also followed the term of service of the website from which we constructed the dataset.

A.2   If you are including theoretical results...

(a) Did you state the full set of assumptions of all theoretical results?   [N/A]

(b) Did you include complete proofs of all theoretical results?   [N/A]

A.3   If you ran experiments (e.g. for benchmarks)...

(a) Did you include the code, data, and instructions needed to reproduce the main experimental results (either in the supplemental material or as a URL)?   [Yes] The source code and instructions of the experiments are available at `https://github.com/mahimanzum/FixEval`.

(b) Did you specify all the training details (e.g., data splits, hyperparameters, how they were chosen)?   [Yes] Please see Section 4 and the supplementary materials.

(c) Did you report error bars (e.g., with respect to the random seed after running experiments multiple times)?   [No]

(d) Did you include the total amount of compute and the type of resources used (e.g., type of GPUs, internal cluster, or cloud provider)?   [Yes] Please see the supplementary materials.

A.4   If you are using existing assets (e.g., code, data, models) or curating/releasing new assets...

(a) If your work uses existing assets, did you cite the creators?   [Yes]

(b) Did you mention the license of the assets?   [N/A] There is no license or restrictions.

(c) Did you include any new assets either in the supplemental material or as a URL?   [Yes] The new assets are available at `https://github.com/mahimanzum/FixEval`

(d) Did you discuss whether and how consent was obtained from people whose data you're using/curating?   [Yes] We have looked into the terms of service and ensured that the code samples can be used for research purposes. We also contacted the respective communities.

(e) Did you discuss whether the data you are using/curating contains personally identifiable information or offensive content?   [N/A] The sources we curated the data from already checks for non-identifiable information, and we also made certain of it.

A.5 If you used crowdsourcing or conducted research with human subjects...

(a) Did you include the full text of instructions given to participants and screenshots, if applicable? [N/A] We did not use crowdsourcing or conduct research with human subjects.

(b) Did you describe any potential participant risks, with links to Institutional Review Board (IRB) approvals, if applicable? [N/A] We did not use crowdsourcing or conduct research with human subjects.

(c) Did you include the estimated hourly wage paid to participants and the total amount spent on participant compensation? [N/A] We did not use crowdsourcing or conduct research with human subjects.

B Datasheet

B.1 Motivation

1. For what purpose was the dataset created? The FixEval benchmark provides a large dataset of bug fixes for Python and Java code along with test suites for programs. Current methods for evaluating program repair models are sub-optimal, and we aim to introduce FixEval as a corpus to minimize the gap between automated program repair research and practice in code repairs during software development.

2. Who created this dataset (e.g. which team, research group) and on behalf of which entity (e.g. company, institution, organization)? The FixEval dataset is created by a team of researchers at Virginia Tech and UCLA.

3. What support was needed to make this dataset? FixEval is a research projectsupported by Virginia Tech.

B.2 Composition

1. What are the instances?(that is, examples; e.g., documents, images, people, countries) Are there multiple types of instances? (e.g., movies, users, ratings; people, interactions between them; nodes, edges) The dataset consists of computer programs that are submissions to online judging sites for competitive programming problems, along with their accompanying test cases and metadata. FixEval does not have multiple types of instances.

2. How many instances are there in total (of each type, if appropriate)? Please refer to Table 1 for a detailed breakdown.

3. What data does each instance consist of ? "Raw" data (e.g., unprocessed text or images)? Features/attributes? The data are Python and Java source code files. The character encoding of each file is UTF-8.

4. Is there a label or target associated with each instance? If so, please provide a description. [Yes] Each instance of data (file) has associated metadata that may be interpreted as labels. Each datapoint consists of a pair of buggy and reference (corrected) code versions in json format. Please refer to Table 2 and Appendix C for some examples.

5. Is any information missing from individual instances? If so, please provide a description, explaining why this information is missing (e.g., because it was unavailable). [No]

6. Are relationships between individual instances made explicit (e.g., users' movie ratings, social network links)? If so, please describe how these relationships are made explicit. Relationships between instances are explicitly available in the provided metadata.

7. Are there recommended data splits (e.g., training, development/validation, testing)? If so, please provide a description of these splits, explaining the rationale behind them. [Yes] Data splits are done explicitly with a set of predefined criteria. Please refer to Section 3 for details.

8. Are there any errors, sources of noise, or redundancies in the dataset? If so, please provide a description. [No]

9. Is the dataset self-contained, or does it link to or otherwise rely on external resources (e.g., websites, tweets, other datasets)? If it links to or relies on external resources, a) are there guarantees that they will exist, and remain constant, over time; b) are there official archival versions of the complete dataset (i.e., including the external resources as they existed at the time the dataset was created); c) are there any restrictions (e.g., licenses, fees) associated with any of the external resources that might apply to a future user? Please provide descriptions of all external resources and any restrictions associated with them, as well as links or other access points, as appropriate. a. [Yes] CodeNet was published in this NeurIPS track in 2021 b. [Yes] c. [No]

10. Does the dataset contain data that might be considered confidential (e.g. data that is protected by legal privilege or by doctor-patient confidentiality, data that includes the content of individuals' non-public communications)? If so, please provide a description. [No]

11. Does the dataset contain data that, if viewed directly, might be offensive, insulting, threatening, or might otherwise cause anxiety? If so, please describe why. [No]

12. Does the dataset relate to people? [No]

B.3 Collection Process

1. How was the data associated with each instance acquired? Was the data directly observable (e.g., raw text, movie ratings), reported by subjects (e.g., survey responses), or indirectly inferred/derived from other data (e.g., part-of-speech tags, model-based guesses for age or language)? If data was reported by subjects or indirectly inferred/derived from other data, was the data validated/verified? If so, please describe how. The data is acquired from CodeNet (https://github.com/IBM/Project_CodeNet) and a DropBox containing test cases from At-Coder (https://atcoder.jp/posts/21).

2. What mechanisms or procedures were used to collect the data (e.g., hardware apparatus or sensor, manual human curation, software program, software API)? How were these mechanisms or procedures validated? Software programs. No verification beyond manual inspection was applied to the data.

3. If the dataset is a sample from a larger set, what was the sampling strategy (e.g., deterministic, probabilistic with specific sampling probabilities)? No specific strategy: as much data as was available with our data collection criteria.

4. Who was involved in the data collection process (e.g., students, crowdworkers, contractors) and how were they compensated (e.g., how much were crowdworkers paid)? Only the authors were involved with the data collection process. There were no third-party participants in the data collection.

5. Over what timeframe was the data collected? Does this timeframe match the creation timeframe of the data associated with the instances (e.g., recent crawl of old news articles)? If not, please describe the timeframe in which the data associated with the instances was created. The data was collected in 2022. Code samples in the dataset go back to approximately a decade ago (2012).

6. Were any ethical review processes conducted (e.g. by an institutional review board)? If so, please provide a description of these review processes, including the outcomes, as well as a link or other access point to any supporting documentation. [No]

7. Does the dataset relate to people? If not, you may skip the remainder of the questions in this section. [No] Only as far as the fact that the data instances are created and written by people.

8. Did you collect the data from the individuals in question directly, or obtain it via third parties or other sources (e.g. websites)? The data was collected indirectly from code available in CodeNet (https://github.com/IBM/Project_CodeNet) and AtCoder (https://atcoder.jp/posts/21).

9. Did the individuals in question consent to the collection and use of their data? If so, please describe (or show with screenshots or other information) how consent was requested and provided, and provide a link or other access point to, or otherwise reproduce, the exact language to which the individuals consented. They did consent directly to the respective online judging sites that we used as source. See e.g. https://onlinejudge.u-aizu.ac.jp/term_of_use.

B.4   Data Preprocessing/Cleaning

1. Was any preprocessing/cleaning/labeling of the data done (e.g., discretization or bucketing, tokenization, part-of-speech tagging, SIFT feature extraction, removal of instances, processing of missing values)? If so, please provide a description. If not, you may skip the remainder of the questions in this section. Minor processing of the data instances ware performed mainly to make the overall pipeline simpler. [No] Primarily manual filtering.

2. Was the "raw" data saved in addition to the preprocessed/cleaned/labeled data (e.g., to support unanticipated future uses)? If so, please provide a link or other access point to the "raw" data. The raw data is saved and available along with the processed data to validate the results. Please refer to the following link: https://github.com/mahimanzum/FixEval.

3. Is the software used to preprocess/clean/label the instances available? If so, please provide a link or other access point. All of the pre-processing, cleaning, and model training code for both Python and Java along with the processed data and trained models are available at https://github.com/mahimanzum/FixEval.

Does this dataset collection/processing procedure achieve the motivation for creating the dataset stated in the first section of this datasheet? If not, what are the limitations? [Yes] This dataset and its derived benchmark datasets offer the scale, quality and diversity to drive research in applying AI techniques to code.

B.5   Uses

1. Has the dataset been used for any tasks already? If so, please provide a description. [No]

2. Is there a repository that links to any or all papers or systems that use the dataset? If so, please provide a link or other access point. [Yes] https://github.com/mahimanzum/FixEval

3. What (other) tasks could the dataset be used for? The rich metadata and diversity of FixEval enables it to be used to evaluate models for many other software engineering tasks. 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.

4. Is there anything about the composition of the dataset or the way it was collected and preprocessed/cleaned/labeled that might impact future uses? For example, is there anything that a future user might need to know to avoid uses that could result in unfair treatment of individuals or groups (e.g. stereotyping, quality of service issues) or other undesirable harms (e.g. financial harms, legal risks) If so, please provide a description. Is there anything a future user could do to mitigate these undesirable harms? [No]

17

5. Are there tasks for which the dataset should not be used? If so, please provide a description. [No]

B.6 Dataset Distribution

1. Will the dataset be distributed to third parties outside of the entity (e.g. company, institution, organization) on behalf of which the dataset was created? If so, please provide a description. [Yes] The dataset will be distributed to the general public.

2. When will the dataset be released/first distributed? What license (if any) is it distributed under? The dataset is released and available without a license at https://github.com/mahimanzum/FixEval.

3. How will the dataset be distributed (e.g. tarball on website, API, GitHub)? Does the dataset have a digital object identifier (DOI)? The dataset is made available as a downloadable gzipped tar file here: https://github.com/mahimanzum/FixEval. There is no DOI yet.

4. Will the dataset be distributed under a copyright or other intellectual property (IP) license, and/or under applicable terms of use (ToU)? If so, please describe this license and/or ToU, and provide a link or other access point to, or otherwise reproduce, any relevant licensing terms or ToU, as well as any fees associated with these restrictions. The dataset is made available at https://github.com/mahimanzum/FixEval without any restrictions.

5. Have any third parties imposed IP-based or other restrictions on the data associated with the instances? If so, please describe these restrictions, and provide a link or other access point to, or otherwise reproduce, any relevant licensing terms, as well as any fees associated with these restrictions. [No]

6. Do any export controls or other regulatory restrictions apply to the dataset or to individual instances? If so, please describe these restrictions, and provide a link or other access point to, or otherwise reproduce, any supporting documentation. [No] These code samples are solutions to competitive programming problems at varying levels of difficulty and should not be subject to export control.

B.7 Dataset Maintenance

1. Who is supporting/hosting/maintaining the dataset? The Computer Science Department at Virginia Polytechnic Institute and State University.

2. How can the owner/curator/manager of the dataset be contacted (e.g. email address)? The users can create an issue on our GitHub repository or directly contact any of the listed authors.

3. Is there an erratum? If so, please provide a link or other access point. [No]

4. Will the dataset be updated (e.g. to correct labeling errors, add new instances, delete instances)? If so, please describe how often, by whom, and how updates will be communicated to users (e.g. mailing list, GitHub)? [Yes] There are plans to add more test cases, as AtCoder and Aizu publish more competitive programming submissions in the next six months to a year's time frame. Any updates will communicated through the GitHub repository.

5. If others want to extend/augment/build on this dataset, is there a mechanism for them to do so? If so, please provide a description. Will these contributions be validated/verified? If so, please describe how. If not, why not? Is there a process for communicating/distributing these contributions to other users? If so, please provide a description. There is no such mechanism yet, but the data is available in both processed and unprocessed format so it can be extended to other formats easily.

C Further Details of our Analysis

C.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**.

C.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/

C.3 **Java Examples Solved only by the model with verdict in Table 6**

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

From the verdict information, the model learned to change the type to be consistent with `long`

Buggy Input Code:
```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)
  }
}
Verdict:  Wrong Answer
```

Model generated Fixed Code:
```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)
  }
}
```

From the verdict information, the model learned to change variable names that exist in the code

Buggy Input Code:
```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")
    }
  }
}
Verdict:  Compilation Error
```

Model generated Fixed Code:
```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")
    }
  }
}
```

C.4 Generated Python code examples with the CodeT5 model in Table 7

Table 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

Model learned to cast the output

Buggy Input Code:	Model generated Fixed Code:

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

Model learned add sort function correctly

Buggy Input Code:	Model generated Fixed Code:

``python
n = int(input())
a = list(map(int,input\
   ().split())).sort()
print(a[-1]-a[0])
``
	``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 Input Code:	Model generated Fixed Code:

``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)
``
	``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 Input Code:	Model generated Fixed Code:

``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)
``
	``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)
	``

C.5 Generated Java code examples with the CodeT5 model in Table 8

Table 8: Some Java examples which were also successfully generated by our model when we pass the buggy code to it. The errors are marked with red

---

Model learned to copy entire line out of scope

Buggy Input Code:	**Model generated Fixed Code:**

```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);
    }
}
```

```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 Input Code:	**Model generated Fixed Code:**

```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(":(");
    }
  }
}
```

```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 Input Code:	**Model generated Fixed Code:**

```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;
  }
}
```

```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 Input Code:	**Model generated Fixed Code:**

```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);
 }
}
```

```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);
 }
}
```