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

Md Mahim Anjum Haque

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Computer Science and Applications

Dwayne Christian Brown, Chair
Ismini Lourentzou
Eli Tilevich

October 30, 2023
Blacksburg, Virginia

Keywords: Competitive Programming, Machine Learning, Automatic Program Repair

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

Md Mahim Anjum Haque

(ABSTRACT)

In a software life-cycle Source code repositories serve as vast storage areas for program code, ensuring its maintenance and version control throughout the development process. It is not uncommon for these repositories to house programs with hidden errors, which only manifest under specific input conditions, causing the program to deviate from its intended functionality. The growing intricacy of software design has amplified the time and resources required to pinpoint and rectify these issues. These errors, often unintended by developers, can be challenging to identify and correct. While there are techniques to auto-correct faulty code, the expansive realm of potential solutions for a single bug means there's a scarcity of tools and datasets for effective evaluation of the corrected code. This study presents **FIXEVAL**, a benchmark that includes flawed code entries from competitive coding challenges and their corresponding corrections. **FIXEVAL** offers an extensive test suite that not only gauges the accuracy of fixes generated by models but also allows for the assessment of a program's functional correctness. This suite further sheds light on time, memory limits, and acceptance based on specific outcomes. We utilize cutting-edge language models, trained on coding languages, as our reference point and juxtapose them using match-based (essentially token similarity) and execution-based (focusing on functional assessment) criteria. Our research indicates that while match-based criteria might not truly represent the functional precision of fixes generated by models, execution-based approaches offer a comprehensive evaluation

tailored to the solution. Consequently, we posit that **FIXEVAL** paves the way for practical automated error correction and assessment of code generated by models. Dataset and models for all of our experiments are made publicly available at https://github.com/mahimanzum/FixEval.

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

Md Mahim Anjum Haque

(GENERAL AUDIENCE ABSTRACT)

Think of source code repositories as big digital libraries where computer programs are kept safe and updated. Sometimes, these programs have hidden mistakes that only show up under certain conditions, making the program act differently than planned which we call bugs or errors. As software gets more complex, it takes more time and effort to find and fix these mistakes. Even though there are ways to automatically fix these errors, finding the best solution can be like looking for a needle in a haystack. That's why there aren't many tools to check if the automatic fixes are right. Enter **FIXEVAL**: our new tool that tests and compares faulty computer code from coding competitions and their fixes. It has a set of tests to see how well the fixed code works and gives insights into its performance and results. We used the latest computer language tools to see how well they fix code, comparing them in two ways: by looking at the code's structure and by testing its function. Our findings? Just looking at the code's structure isn't enough; we need to test how it works in action. We believe **FIXEVAL** is a big step forward in making sure automatic code fixes are spot-on. Dataset and models for all of our experiments are made publicly available at https://github.com/mahimanzum/FixEval.

Dedication

Dedication To the teaching and professional

Acknowledgments

Firstly, I extend my deepest gratitude to the Almighty for bestowing upon me the guidance, strength, perseverance, and wellness necessary to complete this thesis. My heart is filled with appreciation for my cherished parents and spouse, who have been my pillars of support and encouragement throughout this journey. I'd also like to express my profound thanks to my academic mentor, Dr. Chris Brown. His invaluable advice, inspiration, and consistent motivation have been instrumental in the realization of this thesis. Immense gratitude is also due to my committee members Dr. Ismini Lourentzou and Dr. Eli Tilevich, whose insights, guidance, and constructive feedback played a pivotal role in shaping this work. Furthermore, I am thankful for the opportunity provided by Virginia Tech to undertake my MS in computer science. A special thanks goes to the entire computer science department at Virginia Tech, including the dedicated faculty, staff, and fellow graduate students, for enriching my understanding and experiences in this field.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Software program repair stands as one of the most challenging and costly aspects of software engineering. Debugging, the process of identifying and rectifying errors, accounts for approximately 50% of the overall software development expenses [9] and consumes about $70-80\%$ of a software engineer's time [33]. Current research aims to provide better solutions to automate this process [15, 25], but this problem is still far from solved and requires much more research to make it efficient. Automated code correction is a growing field of study [1] that can significantly ease the task for developers by reducing the need to manually address errors in extensive code repositories [13, 14, 32] and maintaining the base code repositories. To cut down on the manual effort involved, there has been a rising trend among researchers to use statistical and neural techniques for automating code repair tasks lately. Models like BART [26] and GPT [11] have shown impressive results in addressing coding-related challenges. Methods for auto-correcting programs, coupled with reliable benchmarks to assess them, play a pivotal role in boosting coding efficiency [38] and cutting down on development expenses [23].

As research into automating program repair expands, there's a noticeable gap in tools to assess the produced fixes. Earlier studies, like Tfix [8], BIFI [42], and CodeBLEU [37], depend on token-level comparison metrics, for example Exact Match. However, a drawback of these metrics that are mainly based on token matching is their tendency to mark down generated fixes for minor deviations from the reference, even when the correction generated by a model is actually appropriate.

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

This sends a wrong signal to the model as simply a token mismatch such as a different variable name doesn't provide necessary information about the functional incorrectness of the program.

CodeBLEU aims to enhance evaluation accuracy by combining weighted word sequence (n-gram) matches with data flow sequence match and Abstract Syntax Tree (AST) based graph components. Yet, this approach doesn't recognize the inherent versatility of code that multiple algorithms, differing syntaxes, varied data flows, or diverse ASTs can achieve identical outcomes or solve identical problems. Consequently, these match-centric methods fall short when tasked with evaluating generated code. This limitation arises from their inability to fully grasp the expansive and intricate domain of program functionality present in solutions. It's noteworthy to mention here that the Automatic Program Repair (APR) field is constrained by the scarcity of comprehensive benchmarks and datasets that cater to this expansive domain. Thus, turning to a rigorously curated test suite becomes imperative, as cited by various studies [1, 4, 12, 21]. Such suites prioritize evaluating the functional integrity of a fix by focusing on its behavior, sidelining mere syntactical conformity. To gauge the quality and operational accuracy of a generated fix, it's deemed correct if it successfully clears a series of unit tests, crafted by domain specialists specifically for that piece of software. The need for improved evaluation methodologies are increasing everyday. The benchmarks to thoroughly assess and evaluate programs generated by automated models are also important.

In this thesis, we provide **FIXEVAL**, a dataset that consolidates competitive programming code submissions from users on AtCoder [5]. **FIXEVAL** encompasses solutions for 700 problems in Python and Java, accompanied by over $40,000$ test cases for assessment. Competitive programming involves coders tackling some of the most challenging issues, all within designated time and memory constraints. The competitive programming process involves submitting code for a given challenge, receiving a verdict based on the code's performance for that specific challenge, making informed adjustments, and iterating until an acceptable solution that adheres to all constraints is achieved. Hence, sourcing **FIXEVAL** from coding contests offers a wealth of paired instances of

flawed and rectified programs, along with their associated unit tests. We showcase the utility of our benchmark by conducting an experimental review that assesses the bug correction capabilities of leading-edge models in the realm of program repair. In this work, we focus on the following research questions.

RQ1: To what extent do pretrained Transformer models excel in FIXEVAL performance?

RQ2: How do match-based metrics correlate with performance in comparison to execution-based evaluation?

RQ3: Are there any trends on model performance with problem difficulty?

We answered RQ1 in Section 5.1.1 where we compared multiple transformer based models along with some other benchmark models to compare their performance.

We answered RQ2 in Section 5.1.2 Where for a fixed test set and model we evaluated the performance of the generations on different metrics to compare.

Finally we answered RQ3 in 5.2 where we group the tests based on difficulty and evaluate their performance on our proposed metrics and show that there are clear trends on model performance with problem difficulty.

The key takeaways from this thesis can be outlined as follows: **(1)** We present **FIXEVAL**, a unique *context-aware* dataset that factors in aspects like time and space complexity when assessing code produced by deep learning models aimed at auto-correcting programming errors. **(2)** We offer an in-depth analysis of leading models using **FIXEVAL**, gauging their proficiency in mending flawed code. Additionally, we make this dataset and the foundational models from our study publicly available. **(3)** We empirically validate the benefits of the suggested execution-based approach to program repair assessment, stemming from our newly introduced test suite.

Chapter 2

Related Works

2.1 Program Repair

Automated program repair (APR) seeks to enhance developers' debugging tasks by autonomously generating corrected programs from erroneous code [25]. It is a pivotal advancement in software engineering, aiming to streamline the debugging process by autonomously detecting and rectifying errors within codebases. Conceptually, APR can be likened to machine translation in natural language processing. Just as linguistic phrases are translated from one language to another, APR endeavors to 'translate' erroneous code segments into their corrected counterparts. The underlying mechanism for this translation process is rooted in advanced machine learning models, specifically language models, which are trained on vast datasets comprising both faulty and rectified code. These models, through rigorous training, acquire the capability to discern patterns and anomalies in code structures, thereby facilitating the generation of accurate fixes. [2]. Numerous studies have demonstrated the effectiveness of language modeling in automating coding tasks, such as program generation [2, 41], program translation between languages [3], and program auto-completion [11]. However, there has been relatively little research focused on applying language modeling to the realm of automated bug fixing and code repair which is one of the most time consuming and laborious tasks for developers.

2.2 Evaluating Pretrained Language Models

Thanks to the recent achievements of large-scale language models in various domains [10, 36, 39], new approaches have emerged, introducing different pretraining objectives tailored to code-related tasks. Software engineering efforts have embraced models like BART [26], GPT [11], and T5 [36], showcasing advancements in automating tasks such as code creation, translation, bug identification, and more, as per their specific metrics. For instance, PLBART [2] is a BART-based model tailored for programming content, utilizing strategies like token masking, deletion, and infilling. On the other hand, Tfix [8] assesses the T5 model [36] by tapping into GitHub commits to address bugs identified by ESLint,[1], a leading static analysis tool for JavaScript. In our study, we train select models, including CodeT5 [41] and PLBART [2], on our dataset using diverse input setups to gauge their efficacy. We delve into multiple sampling criteria during code generation and empirically demonstrate that while larger, well-initialized models trained on code can significantly boost performance, they haven't yet mastered the art of program repair.

2.3 Program Repair Benchmarks

Benchmarks play a crucial role in aiding researchers to assess the efficacy of deep learning methods in auto-correcting code errors. Numerous benchmarks exist to assist researchers in assessing deep learning methods for automated bug fixing. A side-by-side comparison of **FIXEVAL** with recent benchmarks tailored for evaluating machine learning approaches to bug rectification can be found in Table 2.1. DeepFix [16] is comprised of roughly 7K C programs, crafted by novices during an entry-level programming course, spanning over 93 coding tasks. Yet, DeepFix is limited to compiler errors, lacks evaluative test cases, and doesn't mirror real-world software scenarios. On

[1] https://eslint.org/

Table 2.1: A comparison of FIXEVAL with other existing datasets for machine learning-based code repair

	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)	≤ 50, ≤ 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 stands for code review comments, while CE denotes compilation errors.

the other hand, Review4Repair [19] boasts $55,060$ training entries and an additional $2,961$ test entries for Java patches. Its primary objective is to mend code patches leveraging code reviews, which confines its evaluation to match-based techniques. The inclusion of review feedback as a conditional input introduces a broad linguistic diversity, complicating the learning trajectory and necessitating a larger training dataset.

Bug2Fix [40], a notable dataset featured in CodeXGLUE [29], offers both flawed and rectified Java code. Yet, its data is confined to the function level, preventing the modeling of inter-function dependencies. Moreover, Bug2Fix doesn't provide unit tests to validate function accuracy. The GitHub-Python dataset [42] assembles 38K erroneous and 3M accurate unmatched code fragments from GitHub's Python based open-source projects. While its 128-token constraint simplifies the problem's intricacy, the output code is deemed successful only if devoid of AST errors, narrowing the scope exclusively to compiler issues.

Several program repair benchmarks equipped with test suites have been developed to bolster research in automated program repair. Datasets like IntroClass [24] and Refactory [18], for example, are built from student tasks in beginner programming courses and come with unit tests. Yet, their relevance to real-world software is limited. QuixBugs [27] and Defects4J [20] offer more pertinent buggy programs paired with test suites. However, there are notable distinctions between **FIXEVAL** and these datasets. Firstly, **FIXEVAL** overshadows both in terms of size, boasting more lines

of code (Defects4J: 321,000; QuixBugs: 1,034; FixEval: 54 Million in Java and 61 Million in Python). This magnitude allows for comprehensive training and assessment of machine learning techniques for automated code repair. QuixBugs, on the other hand, comprises only 40 "compact" programs, and as noted in a study by Tufano et al. (2019), Defects4J has a limited size, which restricts its usefulness as training data for machine learning models. Additionally, our benchmark offers a broader and more accurate reflection of real-world software. While QuixBugs is limited to one-line defect programs, Defects4J is sourced from merely five open-source Java applications. On the contrary, **FIXEVAL** comprises multi-line defects extracted from an extensive dataset consisting of 712,000 Java and 3.28 million Python submissions. Each of these submissions exhibits a wide range of defect lengths and complexities.

This thesis aims to rectify the limitations of existing datasets by providing more advanced benchmarks for deep learning models in the field of automated program repair research. **FIXEVAL** stands out from conventional machine learning benchmarks, which often comprise basic programming tasks and don't truly reflect the intricacies of substantial real-world bugs. Our benchmark prioritizes the inclusion of a comprehensive test suite, evaluating repairs based on both efficiency and correctness, which proves to be more effective than simple syntax comparison. Furthermore, in contrast to many existing datasets for automated program repair that focus solely on domain-specific open-source code, our dataset takes into account the diverse constraints and contexts that vary between different projects. **FIXEVAL** pioneers as a context-sensitive program repair evaluation dataset, complete with an exhaustive test suite that also weighs in on runtime and memory usage. Distinct from datasets that narrowly focus on domain-specific open-source code, **FIXEVAL** incorporates competitive programming data. Here, the objective is to derive a viable solution that meets specific time and memory complexity criteria.

Chapter 3

Dataset

3.1 Dataset: FIXEVAL

The **FIXEVAL** dataset is comprised of Java and Python program submissions extracted from CodeNet [35]. CodeNet is an assortment of programs, ranging in complexity, submitted by competitive coders to various online evaluation platforms. We have improved this dataset by including test suites for the programs within the validation and test portions. Each of these problems is accompanied by several test cases to assess the accuracy of the program submissions. These test cases usually stay hidden to test the submitted programs on correctness and efficiency on the back-end. Every test is tailored to distinct problems, ensuring a thorough assessment of program capabilities. Although competitive programming data doesn't perfectly mirror professional software development scenarios in the real world, **FIXEVAL** incorporates unit tests and factors in both time and memory constraints. Such considerations are typical when assessing potential software engineer hires [30] and are vital for crafting efficient code in industry contexts [31]. This reflects real-world professional software development, as programmers typically need to write the most efficient code version given time and memory constraints, in addition to creating unit tests for software testing to validate program behavior and avoid degrading user experiences.

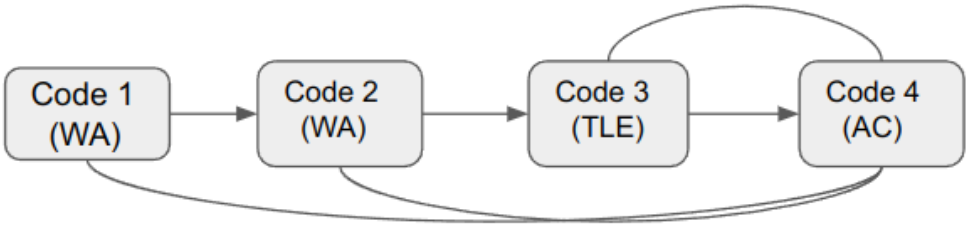

Figure 3.1: Creation of datapoints

3.2 Dataset Construction:

For every user, we examine the sequence of submissions for each tackled problem based on their chronological order. If the code clears all hidden test cases, its outcome or *verdict* is labeled as *Accepted (AC)*. If the code doesn't pass all of the hidden test cases, it may receive one of 12 different verdicts, with the most frequent ones being: *(i) Wrong Answer (WA)*, indicating a failure in one or more test cases in the test suit. *(ii) Time Limit Exceeded (TLE)*, meaning the code didn't complete in that specified limit of time. *(iii) Compilation Error (CE)*, indicating the code couldn't be compiled. *(iv) Runtime Error (RE)*, denoting that the codes execution was unsuccessful [35]. Every submitted code is linked to a verdict. For instance, a user's attempt sequence for a specific problem might look like [WA, WA, TLE, AC], as depicted in Figure 3.1. This represents three unsuccessful tries, comprising two faulty codes and one suboptimal algorithm, before reaching the correct solution.

We match all incorrect submission with its corresponding accepted version, treating this pairing as a singular data point that represents both a flawed and a rectified program. So, for a submission sequence of length n, we gather $n-1$ data points. Some illustrative data points are showcased in Figure 3.2. **FIXEVAL** encompasses all these submission sequences related to specific problems in both Java and Python, spanning all 154k users and their 6.5 million submission trajectories.

On average, users submit 90 programs from a single account. To eliminate redundancy, we employ

Table 3.1: 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

Jaccard similarity, ensuring that multiple submissions for a particular problem are removed. For Java, we utilize the *javalang*[1] tokenizer, while for Python, we use the standard library's *tokenizer*[2].

As detailed in Table 3.1, we establish dataset divisions based on problems, ensuring a distinct separation (80-10-10) for training, testing, and validation sets. This guarantees no overlap of problems or submissions between these divisions. We also confirm that the distribution of problem difficulty remains consistent in the test set. Additionally, we ensure an equitable distribution of problem challenges across all divisions through stratified sampling. Lastly, we ascertain that both the test and validation datasets are equipped with all necessary test cases for program evaluation.

3.3 Test Suite Collection:

We source all test cases, which evaluate the program submissions, from the public test pool [5] provided by the official AtCoder[5] platform. To construct the test suite for our dataset, we match each of the problems name in the metadata of the problems on CodeNet with the available ones on the AtCoder's published website. Subsequently, we pair each problem with its respective input and output files from the test cases. Additionally, we undertake manual cleaning of the test case data. For instance, some programs have varying precision thresholds for numerical outputs, meaning a program's solution is deemed correct if the disparity between its output and the target remains

[1] https://github.com/c2nes/javalang
[2] https://docs.python.org/3/library/tokenize.html
[3] https://atcoder.jp/contests/abc125/tasks/abc125_a
[4] https://atcoder.jp/contests/abc142/tasks/abc142_b
[5] https://atcoder.jp/posts/21

within a specified precision range. There are also constraint-based problems where the primary objective is to meet conditions dictated by rules or design parameters, leading to multiple, equally valid combinatorial outputs. We exclude such problems and their test cases from our evaluation process, given their need for intricate algorithms to achieve an 'Accepted' verdict.

In addition to our test suite collection, our validation set averages 24 test cases for each problem, while our test set averages 25 test cases per problem. These test cases are meticulously crafted by domain specialists, such as those who set competitive programming challenges, to verify the functional accuracy of the submitted programs. Moreover, each test case is designed to evaluate a program under specific extreme conditions. For instance, certain cases focus on pushing the memory usage limits, while others assess the program's performance under the most challenging time constraints.

3.4 Dataset Difficulty:

The **FIXEVAL** dataset comprises Java programs with an average of 331 tokens and Python programs with an average of 236 tokens. This size surpasses that of existing program repair datasets, as indicated in Table 2.1.

Given the extensive combinatorial possibilities inherent to programs, pinpointing the correct solution without a profound grasp of the task is challenging for any machine learning based automated models. We collected the metadata as well that's associated with each problem which also contains the algorithmic and implementation difficulty class that was assigned by the domain experts who creates these problems. We use that difficulty rating, as determined by the official AtCoder website[6]. We maintain this classification, which spans 5 difficulty levels: A through E, with A being the simplest and E the most challenging. The count of the problems by difficulty goes as follows

[6] https://atcoder.jp/

(A:690, B:391, C:760, D:345, E:110) The distribution of these difficulty levels within our dataset is illustrated in Figure 3.3.

Problem Statement[3]: 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);
  }
}
```

Problem Statement[4]: N friends of Takahashi has come to a theme park. To ride the most popular roller coaster in the park, you must be at least K centimeters tall. The i-th friend is h_i centimeters tall. How many of the Takahashi's friends can ride the roller coaster?
Constraints: $1 \leq N \leq 10^5$, $1 \leq K \leq 500$, $1 \leq h_i \leq 500$.
All input values are integers
Time Limit: 2 secs, Memory Limit : 1024MB, Problem Difficulty: B

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)
```

Figure 3.2: Sample submissions from the FixEval dataset. Problematic and corrected statements are highlighted in red and green, respectively.

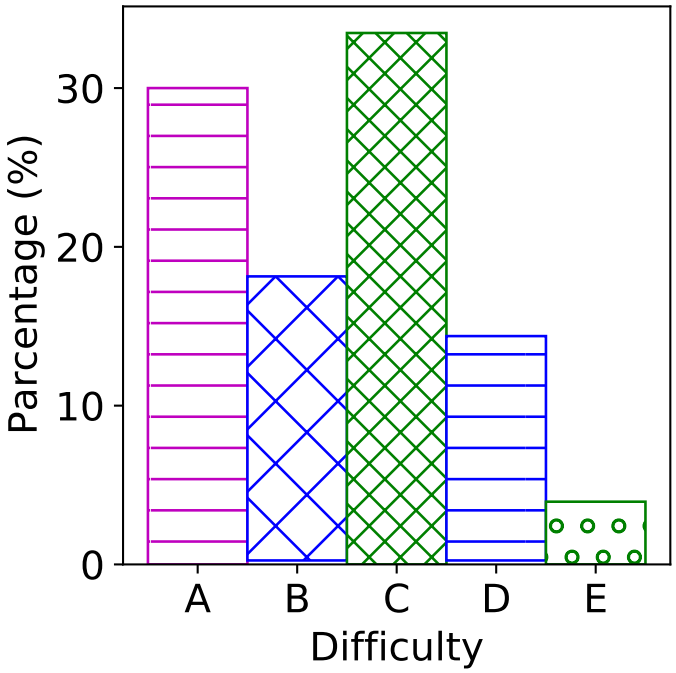

Figure 3.3: Test set Difficulty

Chapter 4

Experiment

4.1 Experiment

We evaluate two Transformer-based language models as our foundational methods: PLBART [2] and CodeT5 [41]. In addition to these models, we also employ a baseline approach termed **Naive Copy**, where the input erroneous code is directly replicated to the output. Given the considerable similarity between the faulty code and its correction, this baseline illustrates the lowest performance a model might attain using match-based metrics.

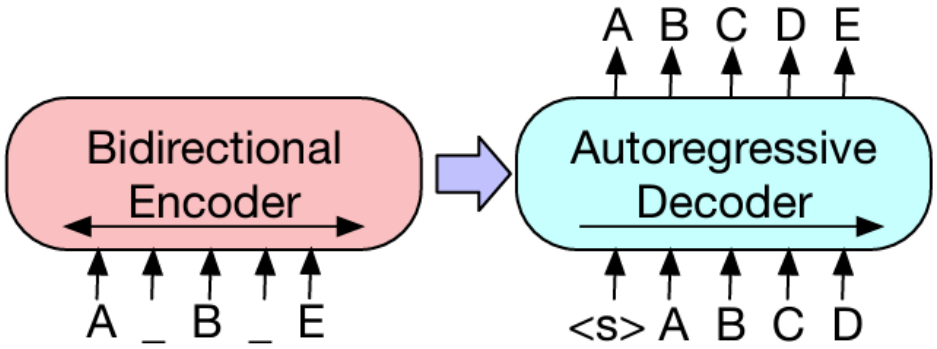

Figure 4.1: Illustration of BART Model

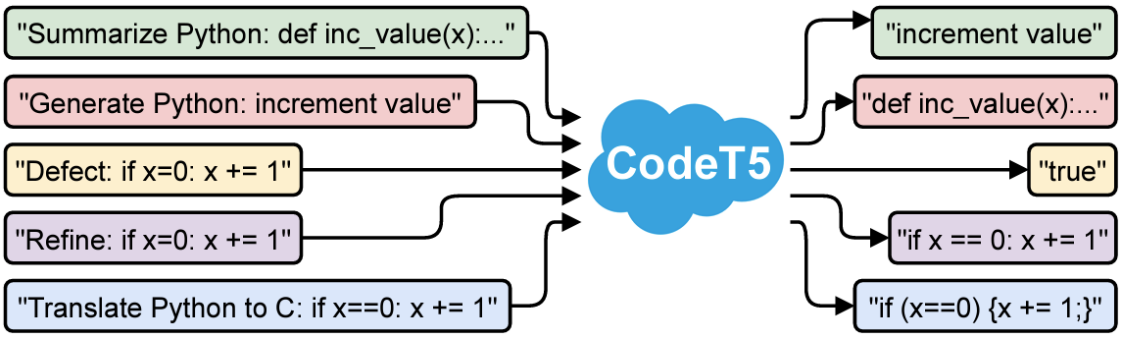

Figure 4.2: Illustration depicting CodeT5 for tasks related to code comprehension and generation.

4.1.1 PLBART

PLBART[2] is a model built on the BART architecture [26], maintaining the same structure as depicted in Figure 4.1. However, it's distinctively trained on programming language datasets using three specific learning techniques: token masking, token deletion, and token infilling. PLBART uses denoising sequence-to-sequence pretraining to utilize unlabeled data in Programming Language and Natural Language. Such pre-training lets PLBART reason about language syntax and semantics. At the same time, PLBART learns to generate language coherently.

4.1.2 CodeT5

CodeT5[41] is a large code language model model built upon the T5 architecture [36] and is specifically pretrained for programming languages. It employs various objectives, including span prediction and identifier tagging prediction, as illustrated in Figure 4.2. The model utilizes both unimodal (solely code) and bimodal (code paired with text) data, as detailed in Figure 4.3. Its primary goal is to establish versatile representations for both programming languages (PL) and natural language (NL) by pretraining on unlabeled source code. Initially, CodeT5 undergoes pretraining with objectives like span prediction, identifier prediction, and identifier tagging across both data types. It's subsequently fine-tuned using dual generation training on bimodal data, enhancing the syn-

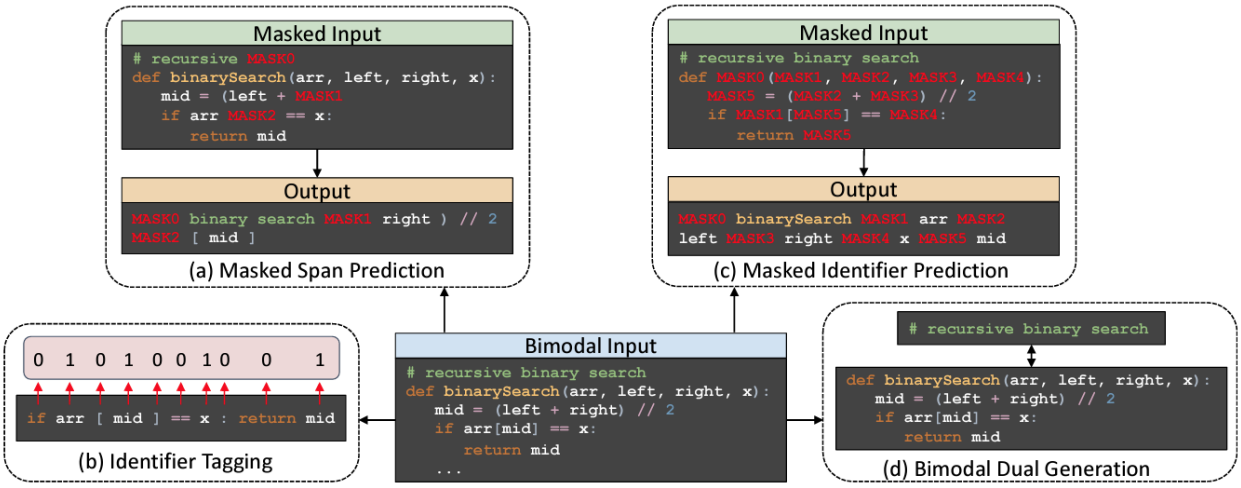

Figure 4.3: CodeT5 Pretraining Tasks

ergy between programming and natural language. CodeT5 is the current state-of-the-art for Code translation from language to language, Code refinement, Clone detection and Defect prediction.

4.2 Setup

In our experimentation, we employed two state-of-the-art models, PLBART and CodeT5, both of which have demonstrated proficiency in code-related tasks. These models were fine-tuned to enhance their performance specifically for Java and Python programs. The base variants of both models were utilized, ensuring a consistent starting point for our experiments. These base models were generously made available by the original authors, allowing for a direct and fair comparison with their published results.

The computational backbone for our fine-tuning process was a V100 GPU, a high-performance graphics processing unit known for its capability to handle intensive machine learning tasks. The models were trained continuously on this GPU for a duration of 48 hours, ensuring ample time for the models to adapt and optimize their parameters for the task at hand.

For the evaluation phase, we utilized the **FIXEVAL** dataset, a curated collection of Java and Python programs designed to assess the models' capability in code repair tasks. The testing procedure employed a beam search strategy, a method that explores multiple potential solutions simultaneously to find the most optimal one. We set the beam size to 5, meaning the model would consider five potential solutions at each step. Additionally, a batch size of 32 was chosen, allowing for parallel processing of multiple data points, thereby enhancing computational efficiency.

The training process was governed by the AdamW [28] optimizer, a variant of the popular Adam optimization algorithm, but with weight decay regularization. This optimizer is known for its effectiveness in training deep neural networks. We set the learning rate to $5 \times e^{-5}$, a value that determines the step size during model optimization. A smaller learning rate ensures a more meticulous search in the parameter space, while a larger one accelerates convergence but risks overshooting the optimal solution.

To prevent overfitting and ensure the model generalizes well to unseen data, we employed early stopping with a patience level set to 3. This means that if the model's performance on the validation set did not improve for three consecutive evaluations, the training would halt. This strategy ensures computational resources are not wasted on training that no longer yields significant improvements. Lastly, we incorporated a warm-up phase of 100 steps at the beginning of the training. During this phase, the learning rate gradually increases, allowing the model to adapt smoothly to the training data before full-scale optimization begins.

4.3 Evaluation Metrics

To gauge the performance of models on FIXEVAL, we assess them using both traditional match-based metrics and our newly introduced execution-based metrics, which we'll detail in the following section.

4.3.1 Match-based Metrics:

Match-based metrics are a set of evaluation measures employed to assess the quality and effectiveness of program repair models based on some sort ot match. These metrics focus on comparing the tokens or some sort of structural similarity of the generated program fix to a reference or ground truth solution. They are necessary in program repair research as they provide a quantitative means to gauge how closely the model-generated fixes align with the desired correct solutions.

The primary purpose of match-based metrics is to measure the degree of similarity or overlap between the model-generated code and the reference code. By doing so, they offer valuable insights into the accuracy and correctness of the generated fixes. Each match-based metric has its unique way of quantifying this similarity, whether it's through exact replication, syntactic matching, or the consideration of logical correctness and data flow structure.

In the context of our research, we utilize a range of match-based metrics, including Exact Match (EM), BLEU, CodeBLEU (CB), Compilation Accuracy (CA), Syntax Match (SM), and Dataflow Match (DM). These metrics collectively provide a comprehensive evaluation of the program repair models' performance.

Exact Match (EM)

Exact Match assesses if the produced program fix is an exact replica of the reference.

BLEU

BLEU measures the overlap between a fix generated by the model and the reference, using the corpus-level BLEU score [34].

CodeBLEU (CB)

CodeBLEU [37] evaluates the quality of code in relation to a reference. Unlike BLEU, CodeBLEU incorporates logical correctness using an Abstract Syntax Tree (AST) and takes into account data flow structure and grammatical resemblance.

Compilation Accuracy (CA)

Compilation Accuracy measures the percentage of generated programs that can be successfully compiled. We employ standard compilers: `javac` for Java and `py_compile`[1] for Python.

Syntax Match (SM)

Syntax Match denotes the ratio of sub-trees extracted from the candidate program's Abstract Syntax Tree (AST) that align or are similar structured with the sub-trees of the referenced program's AST.

Dataflow Match (DM)

Dataflow Match [37] measures the proportion of candidate dataflows that align with the total number of reference dataflows.

4.3.2 Execution-based Metrics:

Given that most code issues aren't purely syntactic, metrics based on n-grams, like CodeBLEU and BLEU, might not be the best fit. In program repair scenarios, there's often a significant lexical

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

overlap between the input and output. However, these match-based metrics might not accurately gauge the functional accuracy of the fixes generated by models. Additionally, there can be multiple valid solutions to fix a program that might differ from the reference. Hence, solely relying on match-based metrics might not capture the true essence of program repair. To address these concerns, we also assess **FIXEVAL** using execution-based metrics.

We provide the complete pipeline for evaluating a program on hidden test cases. We also evaluate how the benchmark models perform on those test cases when we generate programs from the models. We believe our work introduces a critical evaluation metric that goes beyond syntax-level checks, takes into consideration whether a candidate repair satisfies all constraints, and accounts for the actual correct code. For example, Exact match is a lower bound for a program output. Because a bug can be fixed in many ways and any of those can be significantly different from our predefined output. Our metric account for that and get the exact evaluation on a codes accuracy as it evaluates performance by running the code on a set of comprehensive unit tests.

Assessing every generated program based on execution across all test cases is both resource-intensive and time-consuming. To streamline this process, we've chosen two random data points for each problem from the test subset. This selection maintains a similar distribution of error verdicts, ensuring that the evaluation data mirrors the overall distribution of the complete test data for various verdicts like AC, WA, TLE, and so on. Our primary objective isn't a comprehensive evaluation of all models but rather to highlight the potential of the proposed dataset. Therefore, we focus our evaluation on CodeT5, which represents the current pinnacle of performance in this domain. We produce the top-10 outputs using beam search decoding and then assess these outputs by running our test suite, mirroring the evaluation methods of online judges. The execution-based evaluation metrics, pass@k and TCA@k, were originally introduced by [22] and Hendrycks et al. [17]. For clarity and completeness, we delve into their descriptions in the subsequent sections.

Pass@k

Pass@k [22] measures functional accuracy by generating k code samples for each problem. A problem is deemed solved if any of these samples pass all unit tests, and the overall percentage of resolved problems is then reported. Given the potential high variability in this computation of pass@k, we adopt the approach from [11]. Specifically, for each task, we generate up to n samples (in this study, $n = 10$ and $k \leq 10$), identify the number of correct samples $c \leq n$ that clear all unit tests, and compute the unbiased estimator of pass@k as:

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

where $\mathscr{D}test$ represents the **FIXEVAL** test set. It's worth noting that this metric is stringent; a code repair is labeled unsuccessful even if it fails just one test case.

Test Case Average (TCA@k)

To calculate the Test Case Average (TCA@k), we adopt the approach outlined by Hendrycks et al. [17] to determine the average number of successfully passed test cases. Specifically, let P represent the set of problems within the test set, with $|P|$ denoting the total number of problems in P. For a given problem $p \in P$, we denote the code fixes generated to address it as $\langle code^i p \rangle$, where i signifies the index of the generated fix, and k represents the total number of generated fixes. Furthermore, let the set of test cases for problem p be $(xp, c, y_{p,c})c = 1^{|C_p|}$, where xp, c and $y_{p,c}$ denote the input and output pair, and C_p is the count of available test case pairs for that problem. Then, the test case average for k generated fixes (TCA@k) is calculated as follows:

$$\frac{1}{|P|} \sum_{p \in P} \frac{1}{k} \sum_{i=1}^{k} \frac{1}{|C_p|} \sum_{c=1}^{|C_p|} 1 \left\{ \text{eval} \left(\left\langle \text{code}_p^i \right\rangle, x_{p,c} \right) = y_{p,c} \right\}, \tag{4.1}$$

where eval is the name of the function that evaluates a generated program on a test case by evaluating the output of the program matches with the intended result or not. Frequently, solutions will pass or generate correct result on a subset or a portion of the test cases while possibly failing to cover certain corner cases. In such scenarios, a more lenient model evaluation approach is appropriate, as stringent accuracy metrics might obscure actual model enhancements. Therefore, we view the test case average as a form of soft accuracy and present results for different numbers of generations, denoted as k.

Chapter 5

Results

5.1 Results

Our preliminary experiments and analysis seek to answer the following queries: **(1)** How effectively do pretrained Transformer models tackle **FIXEVAL**? **(2)** How do match-based metrics compare in gauging performance against execution-based evaluations? **(3)** Are there discernible patterns in model performance based on problem complexity?

Our findings underscore the necessity for enhanced program repair evaluation methods, highlighting that **FIXEVAL** can address a pivotal gap in the research domain.

5.1.1 Pretrained Model Performance

To address our initial research inquiry, we computed the match-based metrics for our foundational approaches: Naive Copy, PLBART, and CodeT5. The outcomes for these models, based on the match-based metrics outlined in subsection 4.3.1, are displayed in Table 5.1. The 'Verdict' column signifies the incorporation of verdict data as conditional input during the generation of a potential program fix. Notably, Naive Copy outperforms in nearly all match-based evaluations, with the exception of Exact Match (EM). This can be attributed to the likelihood of code pairs undergoing modifications post-receipt of any verdict other than "Accepted". Among the language models compared, CodeT5 and PLBART, CodeT5 consistently surpasses PLBART in all metrics and cod-

Table 5.1: Match-based performance outcomes of language models trained on program repair dataset and evaluated using Java and Python codes from our newly introduced FIXEVAL dataset.

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).

ing languages. This superior performance of CodeT5 might stem from its unique identifier-aware pre-training objective, enabling it to discern patterns in programming languages more effectively.

Moreover, we noticed a slight uptick in performance for our baseline models when verdict data was used as conditional input, specifically for Java programs. This trend wasn't evident for Python. We theorize that Java's inherent verbosity might enhance model training when paired with verdict information. In contrast, Python's more concise nature might not yield the same benefits. A deeper dive into the influence of verdicts is presented in Section 5.2.4.

5.1.2 Match-based vs Execution-based Evaluation

We juxtapose match-based metric and execution-based evaluation metric to determine if they both align with model performance. Addressing our second research query, we assess CodeT5 using execution-based metrics, with the findings presented in Table 5.2 for the selected test set. While Naive Copy emerged as the top performer in match-based metrics (refer to Table 5.1), it notably lags in both execution-based evaluation metrics (as seen in Table 5.2). This indicates that TCA and pass@k might be more reflective of a program's functional accuracy, offering a more com-

Method	Language	Verdict	pass@k				top-k TCA			
			$k=1$	$k=3$	$k=5$	$k=10$	$k=1$	$k=3$	$k=5$	$k=10$
Naive Copy	Java	-	0.0	-	-	-	37.95	-	-	-
	Python	-	0.0	-	-	-	41.55	-	-	-
PLBART	Java	✗	7.51	14.21	17.32	24.14	39.89	33.02	31.88	29.56
	Java	✓	8.43	17.65	21.51	27.17	43.87	37.78	34.71	32.78
	Python	✗	6.15	12.59	15.74	19.98	49.81	40.79	37.63	34.43
	Python	✓	7.01	13.12	16.97	21.91	48.05	40.81	37.89	34.01
CodeT5	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

Table 5.2: Performance assessment using execution-based evaluation metrics on the **FIXEVAL** dataset.

prehensive evaluation than match-based metrics. Additionally, Figure 5.1 reveals that as difficulty escalates, TCA diminishes, whereas no such consistent trend is evident for match-based metrics. This implies that as problem complexity grows, solutions become more challenging to rectify, resulting in reduced TCA scores. We surmise that TCA and match-based metrics (like BLEU, DM, SM, and CB) diverge in behavior because a high similarity in match-based metrics doesn't necessarily equate to program accuracy.

5.2 Ablation Analysis

We also conduct ablation studies on **FIXEVAL** to delve into the impact of various components, such as verdict information, decoding algorithms, and so on. The subsequent analyses are grounded in the results from CodeT5, incorporating verdicts, and are focused on sampled Java instances.

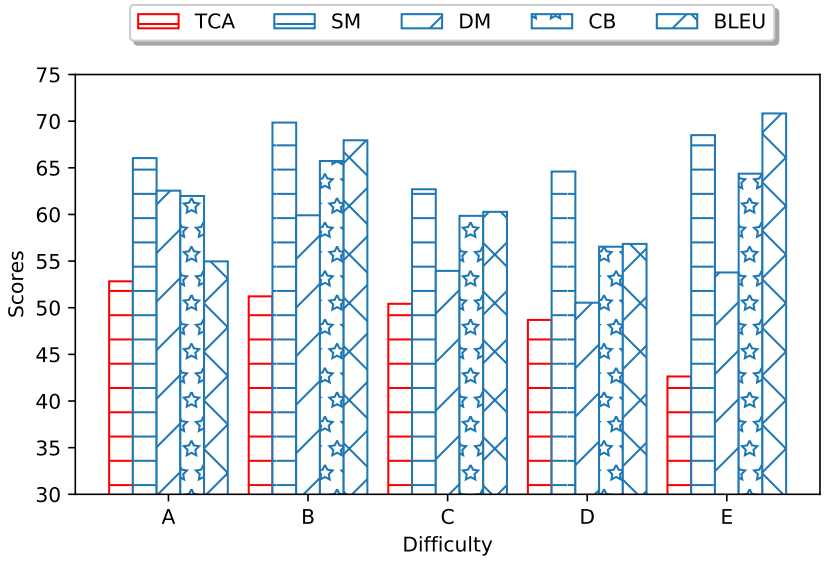

Figure 5.1: Comparison of BLEU and Test Case Average (TCA) across different levels of problem difficulty.

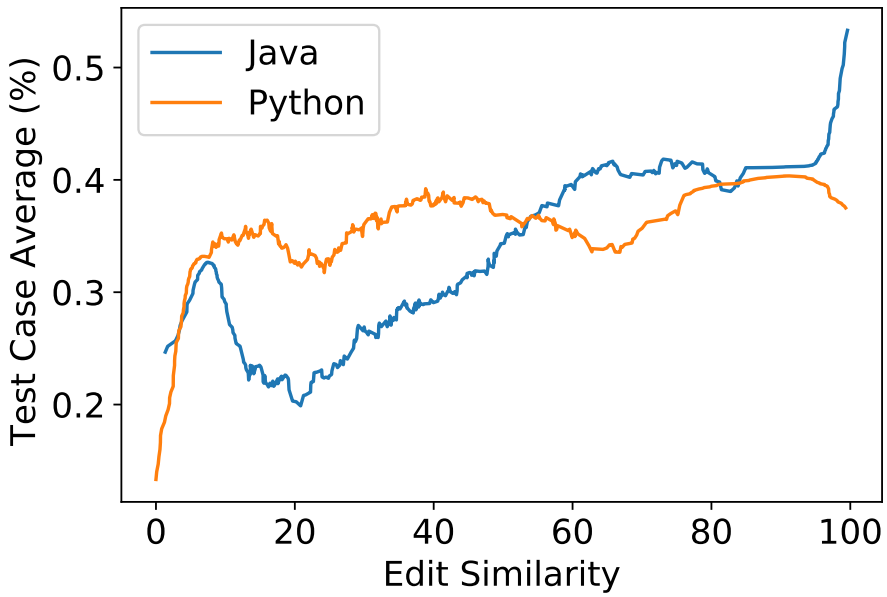

Figure 5.2: TCA rises with greater edit similarity between buggy and reference code.

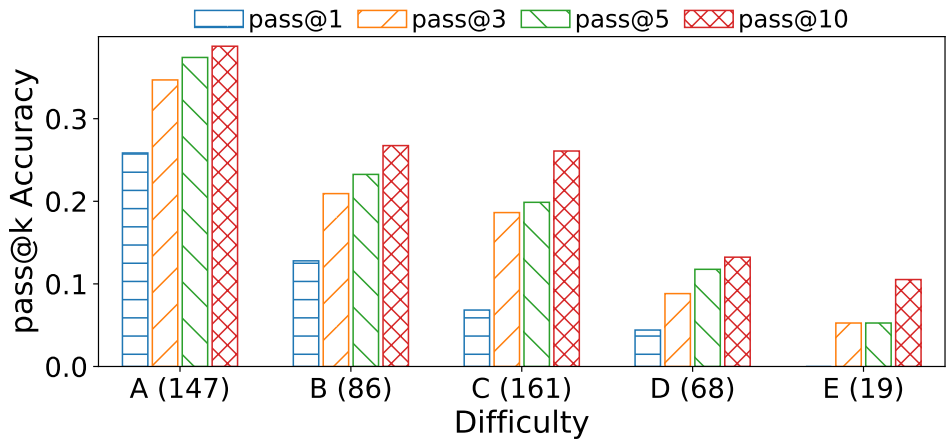

Figure 5.3: Accuracy at various difficulty levels. Labels A to E correspond to increasing difficulty.

5.2.1 Correlation to Edit Similarity

We assess how the model performance based on edit similarity, where we presume that a lower edit similarity between the erroneous and corrected code indicates a more challenging error to fix. We arrange all evaluation data points by the edit similarity between the faulty and reference (corrected) code. Subsequently, we plot the test case average for our best-performing model in both Java and Python. Figure 5.2 illustrates the results, showing a generally increasing trend for Java, indicating a positive connection between edit similarity and model effectiveness. However, this correlation is less evident in the case of Python.

5.2.2 Correlation to Problem Difficulty

We examine the impact of problem complexity as outlined in Section 3. From Figure 5.3, it's evident that as the difficulty escalates (progressing from A to E, indicating problems are becoming more challenging), the model's performance diminishes. Concurrently, accuracy sees an uptick as more programs are generated for the identical input code (transitioning from pass@1 to pass@10).

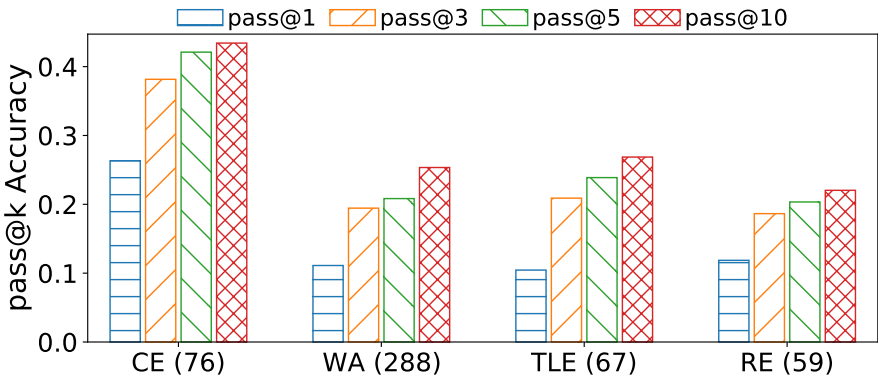

Figure 5.4: Accuracy at different verdict categories, with CE indicating compilation error, WA for wrong answer, TLE for time limit exceeded, and RE for runtime error.

5.2.3 Correlation to Evaluation Verdict

We investigate how different verdict types impact model performance. As depicted in Figure 5.4, we observe that fixing compilation errors (CE) tends to be relatively easier, as these errors typically involve syntactical corrections to the code. In contrast, addressing runtime errors (RE) or time limit exceeded errors (TLE) proves to be considerably more challenging, as they often signify semantically incorrect code that necessitates extensive modifications, possibly spanning the entire algorithm.

Effect of Decoding Algorithms: We produce potential corrected programs using a variety of decoding methods: (i) the greedy approach, (ii) beam search with a beam size of 10, and (iii) top-k sampling, where the top-k probability and temperature are pragmatically set at 0.95 and 0.7 respectively. As depicted in Figure 5.5, beam search decoding typically outperforms both the greedy and sampling methods. Additionally, when experimenting with sampling temperatures ranging from 0.2 to 1.2, we notice only slight variations in performance. We

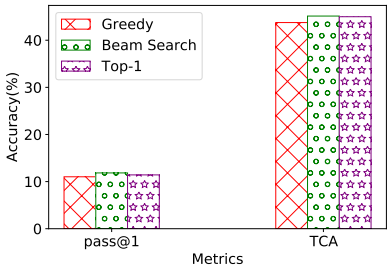

Figure 5.5: Sensitivity of accuracy for strict (pass@1) and soft (top-1 TCA) accuracy using greedy, beam search, and top-k sampling decoding algorithms.

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 5.6: Examples of successful repairs of faulty Java programs with additional model input of verdict information. Buggy and corrected statements are highlighted in red and green, respectively.

attribute this to the problem's characteristics, as the corrected program is often closely aligned with its erroneous counterpart. This similarity likely boosts the model's prediction confidence, meaning that adjustments in temperature don't lead to significant alterations in the model's output.

5.2.4 Effect of Modeling Verdict

We analyze the test cases that were effectively fixed only when the model had access to the verdict information. It becomes evident that this verdict data plays a pivotal role in rectifying certain in-

stances of flawed code. When providing the same code to program repair models, those without access to the verdict information often tend to introduce unnecessary code segments that are syntactically correct but fail to address the underlying issue. Conversely, the model that has access to verdict information can accurately identify the error's location, leading to a more cohesive code correction. Pertinent examples are showcased in Figure 5.6.

5.2.5 Summary of Findings

As suspected, the results show that naive copy baseline performs well across all match-based metrics, demonstrating the inefficiency of such evaluation metrics for program repair. Whereas Naive Copy performs worse on Execution based evaluation metric confirming the effectiveness of a comprehensive test suite and execution based evaluation metrics strengthening our claim of it's necessity. Finally, our experiments show that BLUE does not correlate well with model performance across increasing problem difficulty levels. Our findings confirm the necessity for improved program repair evaluation methodologies, illustrating that **FIXEVAL** can address a significant gap in the research community.

We examined performance patterns in relation to edit similarity, the complexity of the problem, and evaluation verdicts, highlighting that while some tasks of bug-fixing are straightforward, many pose significant challenges. As such, we advocate for subsequent research to take into account all the aspects we've discussed during evaluations. The selection of a decoding strategy yields only minor variations, suggesting it's not a primary determinant in enhancing bug-fixing models. We recommend that future studies explore feedback-driven methods (for instance, feedback from a guiding source or oracle) to refine bug-fixing models.

Chapter 6

Limitations

6.1 Limitations

Our dataset, as presented in this study, has certain limitations. Through our research, where we assessed the errors made by models, we observed frequent model errors such as the unnecessary addition of code segments that don't address the specific bug. This likely stems from a lack of detailed information about the bug and its context. Another constraint is while the dataset of buggy and fixed pairs can be adapted for various coding languages, expanding the programs and its associated test cases for more problems is contingent on what AtCoder offers. Moreover, the computation of the execution based evaluation is time-intensive for individual program evaluations. To mitigate computational demands, we randomly select two data points per problem from the test dataset. However, leveraging parallel processing and high-performance computing can expedite execution-based evaluations, making them more efficient for large datasets. From a research perspective, our dataset focuses on Java and Python bugs, which might not be representative of other programming languages. Further studies are essential to determine FixEval's efficacy in assessing deep learning models for other languages. It's also worth noting that competitive programming submissions might not mirror real-world software bugs encountered by professional developers. There's a pressing need for benchmarks that emulate genuine software applications to assess deep learning models' capabilities in automated program repair.

In Future we also plan to entend this thesis by using agents to run until a solution is reached

satisfying all test cases for a problem solution. We are also planning to Generate knowledge graphs on larger code bases as it's difficult to establish cross file dependency otherwise. We will also focus on incorporating function level relations in the entity, relation mapping as knowledge along with variable initialization. Then explainable GNN models like GNNLrp, Deeplift can also be used to understand the model outputs.

Chapter 7

Conclusion

7.1 Conclusion

We present **FIXEVAL**, an innovative context-aware dataset designed to enhance the creation and evaluation of bug-fixing models. The dataset used here comprises computer programs submitted to online judging platforms for competitive programming challenges, including their corresponding test cases and metadata. Any information related to the programmer who wrote these programs has been anonymized. Unlike previous benchmarks that rely on assessments with publicly available GitHub Codebases or programming assignments, we offer a novel evaluation corpus capable of assessing the correctness, accuracy, and efficiency of generated codes. we employ language models, and it's essential to acknowledge the associated risks and potential harms, which have been extensively discussed in previous research, such as in [6], [10], [7], among others.

Our study evaluates the performance of SOTA models on this dataset, revealing that conventional evaluation metrics are less effective than execution-based metrics derived from test suites that encompass real-world program repair demands commonly encountered in practice. The **FIXEVAL** dataset also opens up avenues for various potential future applications and research directions. It can serve as a valuable resource for evaluating the automation of diverse software engineering tasks, including code completion, code editing, code search, verdict-conditioned code repair, verdict prediction, and chain edit suggestion tasks. Furthermore, as the provided test cases are language-agnostic, our work can be effortlessly extended to encompass other programming languages like

Go, C++, JavaScript etc. 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 extensive metadata and diverse nature of the **FIXEVAL** dataset offer the potential for evaluating various other software engineering tasks. We anticipate that **FIXEVAL** will be an acceleration towards development of more advanced program repair models.

Chapter 8

Summary

In this thesis we provide a new dataset and a methodology for better evaluating programs generated by automated methods. In recent years there have been many research in code automation such as automatic program generation, search, auto completion, bug fixing etc. Though most of the deep learning based approaches tackle these problem form NLP point of view, programming language is much more different and structured than natural language. The quality assessment of programming language is also different. While most of the previous approaches only care for token based matching to assess the quality of a generated program by an automated method, we propose an execution based evaluation method to be used to better evaluate these programs. We claim to correctly evaluate functional correctness of these programs execution based evaluation is much more important and relevant than token based matching which mainly focuses on syntax.

Bibliography

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

[2] 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*, pages 2655–2668, Online, June 2021. Association for Computational Linguistics. doi: 10.18653/v1/2021.naacl-main.211. URL https://aclanthology.org/2021.naacl-main.211.

[3] 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*, 2021.

[4] Andrea Arcuri. On the automation of fixing software bugs. In *Companion of the 30th International Conference on Software Engineering*, ICSE Companion '08, page 10031006, New York, NY, USA, 2008. Association for Computing Machinery. ISBN 9781605580791. doi: 10.1145/1370175.1370223. URL https://doi.org/10.1145/1370175.1370223.

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

[6] 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*, pages 5185–5198, Online, July 2020. Associa-

tion for Computational Linguistics. doi: 10.18653/v1/2020.acl-main.463. URL https://aclanthology.org/2020.acl-main.463.

[7] 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*, FAccT '21, page 610623, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450383097. doi: 10.1145/3442188.3445922. URL https://doi.org/10.1145/3442188.3445922.

[8] 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*, pages 780–791. PMLR, 2021.

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

[10] 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.

[11] 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.

[12] 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*, pages 30–39, 2014.

[13] 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.

[14] 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)*, pages 275–286. IEEE, 2020.

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

[16] 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.

[17] 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.

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

[19] 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.

[20] 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*, pages 437–440, 2014.

[21] 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)*, pages 802–811. IEEE, 2013.

[22] 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.

[23] 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)*, pages 3–13, 2012. doi: 10.1109/ICSE.2012.6227211.

[24] 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.

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

[26] 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.

[27] 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*, pages 55–56, 2017.

[28] 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.

[29] 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.

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

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

[32] 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*, pages 925–936, 2019.

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

[34] 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*, pages 311–318, 2002.

[35] 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.

[36] 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.

[37] 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.

[38] 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*, pages 724–734, 2014.

[39] 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.

[40] 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)*, pages 25–36. IEEE, 2019.

[41] 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.

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