Defects4Ruby: Benchmarking and Analyzing Bug Detection and Repair for Ruby Using Language Models

Meghdad Dehghan*
University of British Columbia
Kelowna, BC, Canada
meghdad.dehghan@ubc.ca

Mohammadreza Saeidi*
University of British Columbia
Kelowna, BC, Canada
mohammadreza.saeidi@ubc.ca

Rohit Dandamudi*
University of British Columbia
Kelowna, BC, Canada
rohit.dandamudi@ubc.ca

Jie JW Wu
University of British Columbia
Kelowna, BC, Canada
jie.jw.wu@ubc.ca

Fatemeh H. Fard
University of British Columbia
Kelowna, BC, Canada
fatemeh.fard@ubc.ca

Gema Rodríguez-Pérez
University of British Columbia
Kelowna, BC, Canada
gema.rodriguezperez@ubc.ca

***Abstract*—Effective methods for detecting and repairing software bugs are essential to ensuring the stability and security of software systems. While Pre-trained Language Models (PLMs) and Large Language Models (LLMs) are used for bug detection and Automated Program Repair (APR), the literature mainly studied a few programming languages, leaving behind low-resource languages such as Ruby. Low-resource programming languages are languages with limited training data available, however, they can be widely used for several applications and have a large community of developers. In this study, we provide the first insights for bug detection and automatic program repair for Ruby language using PLMs and LLMs. In addition to reporting results, we contribute to the research field by open-sourcing our collected dataset, named Defects4Ruby, to study these tasks in Ruby. Our results on Defects4Ruby show that PLMs and LLMs underperform in bug detection compared to traditional machine learning models; while PLMs outperform LLMs for APR. Both bug detection and APR results indicate a need to develop new techniques to detect and fix bugs in Ruby.**

***Index Terms*—Bug Detection, Automated Program Repair, Language Models, Mining Software Repositories, Ruby.**

I. Introduction

In the rapidly evolving field of software development, bugs profoundly impact software systems, leading to crashes, data corruption, and security vulnerabilities [1]. Bugs disrupt software functionality and pose significant financial and reputational risks for companies [2]. Addressing bugs incurs substantial costs, with companies allocating considerable resources to bug detection, localization, and repair [3]. This highlights the urgent need for techniques to detect and fix bugs automatically.

Recent advancements in artificial intelligence, particularly in Pre-trained Language Models (PLMs), have opened new avenues for automating software development tasks. PLMs, with their remarkable ability to simulate human linguistic capabilities [4], have shown promise in various programming-related tasks, including code generation [5, 6], code summarization [7], bug detection [8], and Automated Program Repair (APR) [9]. Despite impressive results in tasks such as code generation, the success rate of PLMs in complex programming tasks like bug detection and APR remains relatively low [4]. Little is known about their performance for low-resource programming languages, such as Ruby, where limited training samples are available [10, 11].

Ruby is a widely used programming language with applications ranging from web development to game design and is utilized by companies like Airbnb, Shopify, and GitHub [12, 13]. The Rails framework, one of Ruby's most popular tools, is extensively adopted for web server development. With over two million projects hosted on GitHub and a large community of developers, Ruby is among the top 10 programming languages used on GitHub [14]. Despite its popularity, automated techniques such as bug detection and APR remain underexplored for Ruby, particularly with the advent of PLMs and Large Language Models (LLMs). While research exists for high-resource languages like Java [9] and C [15], and even for Ruby in traditional machine learning contexts [16], the potential of PLMs and LLMs in automating software engineering tasks for Ruby remains largely unexplored.

To address this gap, we curate a dataset of real-world Ruby bugs by mining commits from GitHub repositories, following methodologies employed in prior studies [17, 18]. This approach enables us to collect diverse bugs at the method level while mitigating data leakage issues. We publish this open-source dataset to support open science initiatives (see Section IX for details).

Using the curated dataset, we evaluate the capabilities of PLMs, such as CodeBERT and CodeT5, and LLMs, including GPT-4o and CodeLlama, for bug detection and APR tasks in Ruby. Additionally, we compare their performance with

*These authors contributed equally to this work.

traditional machine learning models. Our experiments are conducted in both intra-project and cross-project settings, providing insights into the generalizability of these models across different domains.

We aim to answer the following research questions in this study:

- **RQ1: How do PLMs and LLMs perform in bug detection and APR tasks for Ruby?** While prior studies have explored PLMs' performance for various code-related tasks [19, 20], little is known about their effectiveness in bug detection and APR for Ruby. This question evaluates the models' efficiency in fine-tuning and zero-shot learning for these tasks. The results show that machine learning algorithms slightly perform better in bug detection as a classification task while the PLM and LLMs fail to effectively classify the buggy and non-buggy codes. In the APR task, the results of the PLM outperforms the LLMs indicating the potential superior of fine-tuning over zero-shot prompting of larger models.
- **RQ2: How do PLMs perform in bug detection and APR in cross-project settings?** Real-world software projects often require models to generalize across domains. This question investigates the impact of cross-project settings on the performance of PLMs and traditional machine learning models for bug detection and APR tasks in Ruby. The machine learning methods consistently outperform the PLM in all cross-project experiments for the bug detection task. The results of the APR task for this RQ indicate that the PLM performs better than the Naive Copy method.

Our contributions are summarized as follows:

1) **Dataset Curation:** We curate a diverse dataset of real-world Ruby bugs by mining GitHub repositories. This dataset is made publicly available to encourage further research in this domain.
2) **Model Evaluation:** We systematically evaluate the performance of PLMs, LLMs, and traditional machine learning models for bug detection and APR tasks in Ruby, in both intra- and cross-project settings.
3) **Insights:** We provide actionable insights for researchers, with results indicating close-to-random performance in some scenarios, which we report as negative results.

The rest of this paper is structured as follows: Section II and Section III cover methodology, detailing the data collection process and chosen models with evaluation metrics, respectively. Section IV reports the findings of our study. Section V discusses the results and their implications. Section VI reviews related work. Section VII examines the threats to validity. Finally, Section VIII concludes the study and outlines directions for future work.

II. Dataset

A. Data Collection

In this paper, inspired by Chowdhury et al. [17] and Tufano et al.'s [18] approaches, we mine GitHub data to collect pairs of buggy and fixed code. The core idea is to treat the code before a bug-fixing commit as the buggy version and the code after the commit as the potentially fixed version. This methodology relies on bug-fixing commits being submitted to address identified bugs in the codebase. Consequently, we can reasonably conclude that the source code before such a commit contains a bug, and the changes introduced by the commit are intended to fix it. However, as suggested by Chowdhury et al. [17], certain additional criteria should be met to ensure that the code after a bug-fixing commit is free of bugs. For instance, the code should exhibit a two-year stability period and lack subsequent bug-fixing commits on the same method. The details of the data collection process are as follows:

1) Repository Selection: The first step of collecting the required dataset is finding repositories that will be mined to collect bug-fixing commits. To do so, GST [21], we use a dataset containing 25 characteristics of $735,669$ GitHub repositories. With various inclusion and exclusion criteria, the GST dataset can be queried using a web application[1].

Inclusion Criteria. In this paper, we select repositories with the main programming language being Ruby. In addition, selected repositories should:

1) Have at least 1,000 commits to ensure that the final dataset is not biased towards specific projects in terms of having bug-fix pairs. Without this criterion, several repositories may have significantly fewer bug-fix pairs than other projects (e.g., more enormous repositories).
2) Be created before June 3, 2022, to ensure that each selected repository contains at least one file that has remained unchanged for at least two years before the data collection date (i.e., June 3, 2024). This criterion is inspired by Chowdhury et al.'s [17] methodology.

Exclusion Criteria. Repositories that are forked are excluded. A fork is a new repository that shares code with the original repository [22]. If forked repositories are included in the data collection process, it can cause duplicate samples in the final dataset.

By applying these criteria, $2,114$ Ruby repositories were collected to extract bug-fixing commits in the next step.

2) Bug-Fixing Commits Identification: Bug-fixing commits are identified by traversing all commits in each selected repository. PyDriller [23], a tool that clones a project and iterates through its commits on a specific branch, is utilized for this purpose. Two Regular Expressions (RegEx) are applied to the commit messages: The first RegEx, as shown in Listing 1, excludes commits that are cherry-picking or revert commits, or related to documentation, test files, formatting, or linting. Cherry-picking involves copying a commit from one branch to another [24], potentially leading to duplicate samples in the dataset. Revert commits indicate that the changes are no longer valid. Commits related to documentation, formatting, and linting aim to improve code structure rather than address logical bugs. Commits related to tests are also excluded

[1]https://seart-ghs.si.usi.ch/

because changes in test files often reflect changes in the business logic.

Listing 1: Exclusion RegEx.

```
\\b(cherry|reverts?|tests?|rubocop|
documentations?|beautif(?:y|ying|ies)?|
t(re)?format(?:ing|ting)?|lint(?:er|ing))
\\b
```

The second RegEx, inclusion RegEx, identifies bug-fixing commits based on keywords found in their messages, following Chowdhury et al. [17] and Tufano et al.'s [18] studies. As shown in Listing 2, it looks for messages containing words like `fix` OR `solve` OR `repair` OR `address` AND `bug` OR `problem` OR `error` OR `exception` with variations in these words considered.

Listing 2: Inclusion RegEx.

```
\\b(?:fix(?:s|es|ed|ing)?|
(re)?solv(?:e|es|ed|ing)?|repair(?:ed|s|
ing)?|address(?:es|ed|ing)?)\\b.*?\\b(?:
bug(?:s|gy|y)?|problems?|errors?|
exceptions?)\\b
```

3) Bug-Fix Pairs Extraction: For each bug-fixing commit, changed files and methods are analyzed to isolate single-method bug-fix pairs, focusing exclusively on commits where exactly one method has been modified. Also, new methods are excluded as they lack a previous counterpart to be considered buggy code. Each commit's current and previous code snapshots are extracted to represent the buggy and fixed code versions, respectively. Several additional criteria, inspired by the methods in previous works [17, 18], are applied to ensure data quality, as described below.

Reviewing Future Commits: To ensure each extracted bug-fix pair accurately represents buggy and fixed code, the method's subsequent commits are examined to confirm that no later bug-fixing commits modify the current version of the technique. If a future bug-fixing commit alters the current version of the method, the version in the initial commit cannot reliably be labeled as "fixed" since it required further adjustment. In such cases, the version of the method after the later commit is designated as the "fixed" code. In contrast, the version before the initial commit is treated as the "buggy" code, preserving the validity of the bug-fix pairing.

Two-Year Stability Period: Inspired by Chowdhury et al. [17], a method labeled as "fixed" is required to remain unmodified for at least two years after the bug fix to ensure the fix's stability. The commit is excluded from the dataset if the period between subsequent changes is less than two years.

Excluding Methods in Test Files: Since modifications in test files typically mirror changes in the business logic, all methods in files identified as test files are excluded. Test files are recognized using a regular expression that detects variations of the term "test" in file names.

The methodology described yielded a collection of $10,702$ real-world bug-fix pairs for Ruby sourced from $1,289$ repositories. Notably, while the initial repository list included $2,114$ repositories, only $1,289$ contained at least one valid bug-fix pair, with the remaining 829 repositories excluded due to the absence of qualifying bug-fix pairs. This study will use this dataset for both bug detection and APR tasks.

4) Manual Evaluation: A manual analysis was performed on a randomly selected sample of 100 commits to ensure the quality of the collected bug-fixing commits. The first three authors individually reviewed these commits and determined whether they were submitted to fix reported bugs. After resolving seven conflicts, it was found that eight of the commits were not bug-fixing commits. This results in a precision of 92% for the dataset.

As this precision was above 91% and is accepted as a valid accuracy in similar studies [25], we used our collected dataset for the experiments in this work. Additionally, we discuss the related threats in Section VII.

5) Dataset Preparation: Following the described process, we collected $10,702$ bug-fix pairs, each consisting of a buggy method and its corresponding fixed method. These pairs are directly used in the APR task, where the buggy code serves as the input to the models, and the fixed code is treated as the ground truth for the models' output.

We process each pair separately for the bug detection task by treating the buggy and fixed code as individual records. A binary label is assigned to each record to distinguish between them, indicating whether the code is buggy or not. This process results in a dataset containing $21,404$ records, evenly split between buggy and non-buggy code.

As the total number of records in the dataset is approximately $10K$ for within-project settings and fewer than $5K$ for cross-project settings, we opted to divide the dataset into training and test splits, with the ratio of 8:2. This decision is motivated by the observation that the models show no overfitting issues, as evidenced by the low changes in training loss during the training phase. Even with an increased number of epochs, the models remain underfitted. The absence of overfitting issues eliminates the need for a validation split, which is primarily used for regularization and avoiding the models to be overfitted to the training data [26]. Instead, by limiting the dataset to training and test splits, we increase the number of training samples, potentially allowing the model to learn more effectively from the data. However, as shown in Section IV, our experiments reveal negligible improvements under this setting.

Given that bug detection and APR are time-sensitive tasks [17], we ensure that the models are trained on older data than what is included in the test set.

III. Language Models and Evaluation Metrics

This section covers different techniques incorporated in the study to tune or prompt the models in the chosen downstream tasks of bug detection and APR for the Ruby programming language.

A. The Choice of Language Models

CodeT5 is a pre-trained Transformer model designed for code understanding and generation tasks. It employs a unified

framework that utilizes the coding semantics conveyed through the identifiers assigned by developers, allowing for a more seamless and accurate understanding of code. Moreover, it will enable multi-task learning, making it a versatile and powerful tool for developers to use in their projects [27]. It also performs well in vulnerability detection compared to the novel deep learning method at a function level scope [28] that fits the tasks and problems we aim to investigate in this study.

Code-Llama-7b-Instruct This model is one of the latest state-of-the-art open-source Large Language Models and part of the CodeLlama model series released from Meta [29]. CodeLlama is a highly specialized version of Llama2, designed to cater specifically to the needs of developers and programmers. It results from further fine-tuning of the Llama-2 model on programming language corpus [29]. This has enabled CodeLlama to develop a deep understanding of the intricacies of coding languages and their nuances, making it a powerful tool for developers to streamline their coding process. Given our limited resources, we use the 7B version of this model to enable experimentation on a single GPU. Additionally, since we apply zero-shot prompting without additional fine-tuning, we rely on the instruction-tuned version to generate task-specific responses. Without instruction tuning, pre-trained versions of large language models (LLMs) tend to provide open-ended answers that do not meet our requirements. In Section IV, we elaborate on the prompt design used to obtain the desired output from the model.

GPT-4o This model is one of the most popular and latest commercial language models developed by OpenAI, capable of handling various tasks and equipped with advanced reasoning capabilities [30]. Introduced in May 2024, this model can respond to text, image, and audio queries. It performs better than its competitors on the HumanEval [20] benchmark, one of the most widely recognized benchmarks for code generation and intelligence. This model was selected because it is currently the most widely used for solving various tasks, including programming.

Table I presents an overview of the PLMs and LLMs used in this study, including their sizes and corresponding data used during their training phase.

TABLE I: PLMs and LLMs used in our experiments.

Model	Training Data	Parameter Sizes
CodeT5	Ruby, JavaScript, Go, Python, Java, PHP, C, C#	Small (60M) and Base (220M)
CodeLlama	C++, Java, PHP, TypeScript, C#, Bash	7B
GPT-4o	NA	NA

B. Learning methods

1) Fine Tuning: Fine-tuning PLMs represents an initial approach in transfer learning aimed at adapting language models to specific downstream tasks [31]. This technique updates all model parameters during the adaptation phase, necessitating significant time and memory resources, particularly as model sizes expand [6]. Building upon prior research [9, 32], we undertake fine-tuning of a PLM, i.e., CodeT5 (base) [27], aimed to analyze the Ruby dataset for the bug detection and APR tasks. Following the prior research [33], we set the initial value of hyper-parameters for our experiments as shown in Table II.

TABLE II: The initial value of hyper-parameters used to fine-tune the PLM for both bug detection and APR tasks.

Hyper-parameter	Value
Learning rate	1e-5
Optimizer	AdamW
Warm up steps	10%
Adam-epsilon	1e-8
Training batch size	16
Validation batch size	16

We set the models' input token size for both tasks to 256 tokens. The model's output size for the APR task is also set to 256 tokens. We train the models on all dataset variants, both for RQ1 and RQ2, for three epochs and report the results on the validation set after the final epoch. All experiments are conducted on a single NVIDIA Tesla V100 32GB GPU.

2) Zero-shot learning: Zero-shot learning or prompting is one of the primary techniques in in-context learning, where the model is prompted with the task to be performed without being given any examples of the task [34]. In most use cases, developers apply zero-shot prompting to LLMs to solve their problems in the software development process. In this study, we use zero-shot prompting of LLMs, i.e., Code-Llama-7b-Instruct and GPT-4o, as a widely used and primary approach to evaluate bug detection and APR tasks using such models.

For the bug detection task, we use prompts to direct the models to classify the provided code as either 'buggy' or 'non-buggy'. Initially, these models are adjusted to generate more information about the program snippets. To ensure we can identify the code label in the model output, we prompt them to provide only the label of the given code without additional information. The following is the prompt for bug detection by LLMs: *"Classify the following Ruby code as 'buggy' or 'non-buggy.' Only answer with 'buggy' or 'non-buggy':* {`CODE`}", where {`CODE`} represents the input code, which may be either buggy or non-buggy.

We ask the models to generate a fixed code version for the APR task without analyzing the existing bugs. This allows us to retrieve only the fixed code snippet and compare it with the ground truth in the dataset. The prompt for this task is as follows: *"Fix the bug in the given code and output the fixed version of the code; only respond with the fixed code, without any extra explanations:* {`CODE`}", where {`CODE`} represents the given buggy code provided to the model.

C. Machine Learning Methods

Multiple studies have demonstrated that machine learning methods are highly effective in identifying defects [35, 36]. As such, we have carefully selected two models that have shown exemplary performance in classification tasks.

TABLE III: Accuracy, precision, recall, and F1-score of different models on bug detection. The results for Code-Llama-7b-Instruct and GPT-4o models are reported in over 100 samples.

Model	Accuracy	Precision	Recall	F-Score
Random Forest	56.97%	56.99%	56.97%	56.96%
XGBoost	**58.78%**	**58.79%**	**58.78%**	**58.74%**
CodeT5	50.95%	51.05%	50.95%	50.54%
Code-Llama-7b-Instruct	44.00%	44.19%	44.00%	37.87%
GPT-4o	49.00%	49.08%	49.00%	49.04%

TABLE IV: BLEU score of language models on APR task. The results of Code-Llama-7b-Instruct and GPT-4o models are reported on 100 dataset samples.

Model	BLEU
CodeT5	**90.12%**
Code-Llama-7b-Instruct	61.57%
GPT-4o	78.68%

1) XGBoost: XGBoost [37] is an open-source machine-learning method that utilizes an advanced algorithm called gradient boosting to power scalable and distributed decision trees. It is designed to work seamlessly with large datasets and can be used for various applications, including regression, classification, and ranking problems. The parallel tree boosting functionality ensures faster and more accurate model training, making XGBoost the leading choice for data scientists and machine learning enthusiasts worldwide.

2) Random Forest: Ensemble classifiers are a group of machine learning algorithms that combine the predictions of multiple models to improve overall performance. One commonly used ensemble classifier is Random Forest [38]. This algorithm is known for its ease of use and flexibility, as it can handle both classification and regression problems. When constructing decision trees in a Random Forest, a random selection of attributes is chosen, and individual trees are created using a simple algorithm. Unlike other decision tree algorithms, pruning is not performed at each node, and attributes are sampled randomly. The unlabelled example is classified based on the majority of voting. One significant advantage of Random Forest is its speed and ability to handle many input attributes [36]. This algorithm is also less prone to overfitting, a common problem in decision tree algorithms.

To identify optimal hyperparameter configurations for training machine learning algorithms, we conducted experiments using different values of maximum depth of the trees in $\{16, 64, 256, 512\}$ and number of estimators in $\{2, 10, 100\}$. Based on these experiments and their results, we set the maximum tree depth and the number of tree estimators to 256 and 10, respectively. We opt for these values across all experiments involving machine learning algorithms.

D. Evaluation Metrics

This study reports accuracy, precision, recall, and F1-score as commonly used metrics for evaluating classification tasks, such as bug detection [17]. Accuracy measures the proportion of correct predictions made by the model out of the total number of predictions. Precision evaluates the quality of positive predictions by calculating the ratio of correctly predicted positive samples (e.g., buggy code) to the total number of samples predicted as positive. Recall is the ratio of correctly predicted positive samples to the total number of positives. Finally, the F1-score provides a harmonic mean of precision and recall, balancing their trade-offs. These metrics collectively offer a comprehensive assessment of the model's performance in the bug detection task.

For the APR task, the generated code from the models is expected to be assessed about the ground truth code in the dataset. To measure the similarity of these code snippets, we use the widely used *BLEU score* [39] similar to prior studies [27, 33, 40]. This metric measures the similarity of two sequences of tokens by incorporating the overlapping n-grams in the sequences. The exact definition of this metric is formulated as follows:

$$BLEU = BP \times exp(\sum_{n=1}^{N} w_n log(p_n)) \quad (1)$$

Where BP is the brevity penalty, calculated based on the generated output's length and the dataset's ground truth code. Additionally, p_n represents the modified precision for n-grams, and w_n denotes the weight of each n-gram. We report the BLEU score with $n = 4$ in all the experiments similar to prior studies [33].

IV. Results

This section presents the results of applied methods to evaluate PLMs, LLMs, and ML algorithms for both bug detection and APR tasks.

A. PLMs and LLMs Performance for Bug Detection and APR

To answer this question, we examine the bug detection capability of classical machine learning methods on the Ruby dataset and compare it with the results of the language model's fine-tuning and zero-shot prompting. The results are represented in Table III and Table IV for bug detection and APR tasks, respectively. The Ruby dataset used for this research question contains more than 10K samples, whereas for the larger models, i.e., Code-Llama-7b-Instruct and GPT4-o, we report the results on 100 randomly selected records. This choice is made for LLMs, as we use the prompting strategy rather than training the models. As observed for bug detection tasks, when comparing the classical machine learning methods

TABLE V: Accuracy, precision, recall, and f1-score of CodeT5 model vs. machine learning algorithms, i.e., Random Forest and XGBoost, on bug detection in the cross-project setting.

Dataset Splits	Model	Accuracy	Precision	Recall	F1-Score
Train(3156) / Test(890)	CodeT5	47.07%	46.24%	47.07%	43.28%
	Random Forest	**59.48%**	**59.51%**	**59.48%**	**59.47%**
	XGBoost	55.50%	55.57%	55.50%	55.44%
Train(3392) / Test (654)	CodeT5	46.43%	46.60%	46.43%	46.39%
	Random Forest	**53.88%**	**54.32%**	**53.88%**	**53.66%**
	XGBoost	53.57%	53.58%	53.57%	53.57%
Train(3451) / Test (595)	CodeT5	54.83%	55.00%	54.83%	54.87%
	Random Forest	**63.19%**	**63.76%**	**63.19%**	**63.16%**
	XGBoost	56.66%	57.09%	56.66%	56.65%
Train(3187) / Test (859)	CodeT5	51.43%	51.57%	51.43%	51.37%
	Random Forest	50.96%	50.80%	50.96%	50.62%
	XGBoost	**53.71%**	**53.69%**	**53.71%**	**53.69%**
Train(3302) / Test (744)	CodeT5	49.84%	50.13%	49.84%	49.80%
	Random Forest	56.37%	57.28%	56.37%	55.95%
	XGBoost	**59.32%**	**59.55%**	**59.32%**	**55.96%**
Train(2919) / Test (1127)	CodeT5	50.53%	50.67%	50.53%	50.55%
	Random Forest	47.86%	48.17%	47.86%	47.69%
	XGBoost	**51.79%**	**51.80%**	**51.79%**	**51.79%**
Train(3556) / Test (490)	CodeT5	49.09%	49.10%	49.09%	49.09%
	Random Forest	52.55%	52.54%	52.55%	52.33%
	XGBoost	**53.04%**	**53.03%**	**53.04%**	**52.94%**
Train(3560) / Test (486)	CodeT5	51.30%	51.67%	51.30%	50.61%
	Random Forest	**53.57%**	**53.51%**	**53.57%**	**53.39%**
	XGBoost	51.90%	52.02%	51.90%	51.84%
Train(2924) / Test (1122)	CodeT5	52.15%	53.06%	52.15%	50.16%
	Random Forest	**56.11%**	**56.08%**	**56.11%**	**56.06%**
	XGBoost	53.18%	53.27%	53.18%	53.15%
Train(3019) / Test (1027)	CodeT5	50.63%	50.29%	50.63%	50.14%
	Random Forest	**52.64%**	**52.66%**	**52.64%**	**52.65%**
	XGBoost	51.37%	51.61%	51.37%	51.36%
Average	CodeT5	50.33%	50.43%	50.33%	49.63%
	Random Forest	**54.66%**	**54.86%**	**54.66%**	**54.50%**
	XGBoost	54.00%	54.12%	54.00%	53.64%

TABLE VI: The BLEU Score of CodeT5 model on 10 cross-project data splits for automated program repair task. The Naive Copy column shows the score when no changes is made in the buggy code.

Dataset Split	CodeT5	Naive Copy
Train(3156) / Test (890)	**88.33%**	84.45%
Train(3392) / Test (654)	**90.78%**	80.43%
Train(3451) / Test (595)	85.46%	**86.56%**
Train(3187) / Test (859)	**90.09%**	80.82%
Train(3302) / Test (744)	86.16%	**86.21%**
Train(2919) / Test (1127)	**90.97%**	87.22%
Train(3556) / Test (490)	**91.46%**	85.20%
Train(3560) / Test (486)	**90.83%**	84.60%
Train(2924) / Test (1122)	**89.38%**	84.65%
Train(3019) / Test (1027)	**90.71%**	84.50%
Average	**89.42%**	84.46%

with language models, the Random Forest algorithm achieves better results with an accuracy of 58.78%

The results of all language models are close to 50%, similar to randomly guessing the labels of bugs in the dataset. This suggests that the learning capability of the models on the low-resource language Ruby for this task is limited. In contrast, classical machine learning approaches still yield better performance for this task.

More interestingly, the performance of zero-shot prompting with Code-Llama-7b-Instruct and GPT-4o models is even below 50%, indicating that even larger models fail to detect bugs effectively. In scenarios with only the buggy code, learning-based methods, such as traditional machine learning algorithms and fine-tuning pre-trained language models, continue to show superior performance.

Despite the poor performance of the pre-trained language model on the bug detection task, its results on the automated program repair task is relatively high. Table IV reports the results for APR. Machine learning methods are only applied to the bug detection task, as these methods are compatible with classification tasks and not the generative task of APR. The BLEU score of the CodeT5 model is 90.12%, which is higher than the naive copy score in our dataset. The naive copy score for Ruby on APR is 84.36%. This score is obtained when the initial buggy code is considered as the fixed code and is thus

compared with the ground truth fixed code. In other words, this suggests that the model does not learn to introduce any changes to the initial code. We consider the naive copy of the input code as the baseline similar to previous works [40].

The results of zero-shot prompting for LLMs are lower than the baseline score, with scores of 61.57% and 78.68% for the Code-Llama-7b-Instruct and GPT-4o models, respectively. This indicates that these models tend to produce more corrupting changes to the initial code rather than fixed patches. In Section V, we further investigate this issue by providing an example of for each of the models.

Finding 1: Results of the first RQ show that Random Forest outperforms PLMs and LLMs in bug detection tasks for the Ruby language. All language models achieve near 50% accuracy, equivalent to random guessing, indicating that such models fail to address the bug detection task for Ruby effectively. In contrast, fine-tuning PLMs on the Ruby dataset shows improved results compared to the baseline score on the APR task. Such improvement is not observed when using LLMs with zero-shot prompting. This suggests that fine-tuning PLMs has an advantage in APR tasks for low-resource languages over zero-shot prompting of larger models, indicating the need for other techniques to use LLMs for APR in Ruby.

B. Models' Performance on Bug Detection and APR in Cross-Project Setting

Table V shows the results of the fine-tuned PLM and ML algorithms on different dataset variants for the bug detection task, along with the train and test split sizes. Similar to the previous RQ, we observe that the performance of ML algorithms significantly outperforms that of the fine-tuned PLM. The performance decreases compared to RQ1, indicating the models' performance drops when tested on new data, i.e., projects. Among these algorithms, Random Forest scores higher than XGBoost in 6 of the ten experiments.

More specifically, the results of the fine-tuned PLM are near 50% for all dataset variants, with several cases falling below 50% for all metrics. Such poor performance demonstrates the inefficiency of fine-tuning PLMs when the domain of bugs differs between the train and test splits. However, ML algorithms maintain their performance even when the domain of the training data differs from the test data. These algorithms are more robust in low-resource scenarios where the available data for training is limited.

Table VI shows the results for the same data distributions on the APR task with the fine-tuned CodeT5 model. As observed, the results are higher than the naive copy score in 8 out of 10 experiments, with 91.54% as the highest and 85.46% as the lowest BLEU scores. Such scores indicate the effectiveness of fine-tuning models for APR, even when the train and test splits distributions differ.

Finding 2: In analyzing 10 Ruby datasets with different domains for their train/test splits, fine-tuned CodeT5 does not outperform traditional machine learning methods. The Random Forest algorithm performs better in most cases for detecting bugs than other methods in this cross-project evaluation.
The enhanced performance of the PLM in the APR task is still observable across different domains compared to naive copy. However, the average performance of the model on all cross-project datasets drops when compared with the intra-project setting (RQ1) indicating that the having samples from same projects and domains in both train and test data splits helps the performance improvement of the model.

V. Discussion

A. Result Analysis

In Section IV, the performance of the selected PLM for the bug detection task was discussed. However, the results were unsatisfactory, with the maximum accuracy reaching only 50.95%, comparable to guessing randomly. Additionally, the models were fine-tuned and evaluated in a cross-project setting, where the distributions of the training and testing splits differed, each containing distinct projects. Similarly, significant changes in the results were not observed, and the models' performance remained poor. In both cross-project and intra-project settings, the machine learning algorithms outperformed the PLM and the LLMs for the first RQ, highlighting the effectiveness of such methods for low-resource programming languages like Ruby.

Other studies [15, 40] have demonstrated higher performance of fine-tuning PLMs for the bug detection task in the C programming language (62.08%). Considering the lower readability and, consequently, reduced understandability of C code snippets by language models compared to Ruby, the higher performance of PLMs on C is noteworthy. These results highlight the significant impact of data quality and size when training these models for the bug detection task. Notably, the dataset used in the other study contains nearly 20K samples, which is double the size of our dataset. This could be a potential reason for the higher performance of the models on C, in addition to the data quality and the types of bugs present in the dataset.

B. Example Cases

Despite of the high score of the PLM and good scores of LLMs for the APR task, the real capability of such models in fixing buggy codes is limited. Figure 1 illustrate an example of the APR task in which the buggy code, ground truth fixed code, and generated codes of the models are represented. As it is observed, the generated codes by all models do not correctly address the initial bug in the input code. Particularly, among those three models, Code-Llama-7b-Instruct fails to capture the buggy line of the code and makes unnecessary changes in wrong parts. CodeT5 and GPT-4o models correctly capture the

```ruby
def index
  authorize(Guidance)
  paginable_renderise(
    partial: "index",
    scope: org.admin,
    query_params: { sort_field: "guidances
        .text", sort_direction: :asc }
  )
end
```

(a) Input buggy code.

```ruby
def index
  authorize(Guidance)
  paginable_renderise(
    partial: "index",
    scope: Guidance.by_org(current_user.org)
                   .includes(:guidance_group,
                             :themes),
    query_params: { sort_field: "guidances
        .text", sort_direction: :asc }
  )
end
```

(b) Fixed code as ground truth.

```ruby
def index
  authorize(Guidance)
  paginable_renderise(
    partial: "index",
    scope: Guidance.
    query_params: { sort_field: "guidances
        .text", sort_direction: :asc }
  )
end
```

(c) Generated code by CodeT5 model.

```ruby
def index
  authorize(Guidance)
  paginable_renderise(
    partial: "index",
    scope: org.admin,
    query_params: { sort_field: "guidances
        .text", sort_direction: :asc },
    sort_field: "guidances.text",
    sort_direction: :asc
  )
end
```

(d) Generated code by Code-Llama-7b-Instruct model.

```ruby
def index
  authorize(Guidance)
  paginable_renderise(
    partial: "index",
    scope: Guidance.all,
    query_params: { sort_field: "text",
        sort_direction: :asc }
  )
end
```

(e) Generated code by GPT-4o model.

Fig. 1: An example of buggy code and its fixed version and generated codes by different models. Highlighted lines represent the buggy part of the input code (a) and the changes in others (b, c, d, and e).

buggy line of the code yet both of them fail to fix it correctly. Additionally, the generated codes by larger models, i.e., Code-Llama-7b-Instruct and GPT-4o, are syntactically correct while the generated code by CodeT5 is corrupted.

A manual analysis of the results of the initial experiments (fine-tuned CodeT5) was conducted to investigate the reasons for the poor performance further. A random dataset of 200 predicted bug-fix pairs was curated, including 50 samples each of True Positive, True Negative, False Positive, and False Negative. These samples were independently analyzed by two authors to identify any meaningful patterns between the correctly predicted samples and the failures. However, this analysis did not reveal any specific pattern that the model followed to produce better results for certain types of bugs.

In our manual analysis, it was also observed that detecting whether a code snippet contains a bug was challenging for the authors, primarily because they were not familiar with the projects' specific business requirements. Some buggy methods are heavily dependent on the context of the project. In other words, a specific method might be considered buggy in one project while functioning correctly in another due to differences in their business requirements. To this end, since we are not providing further information about the existing bugs in the code, it is possible to achieve poor performance on these tasks.

Another issue we found in our manual analysis is related to the scope of the collected bugs. In our data collection phase, we consider the changed methods to be the scope of the bug and only feed these methods to the model. However, based on the manual analysis, we observed many methods that produced bugs in other methods or files. In other words, although the technique caused a bug in the system, it could not be discovered by considering the changed method. In fact, by considering only the changed method, the method would

be regarded as a non-buggy code. All these factors likely contributed to our study's low-accuracy of the models.

C. Actionable Insights

Based on our analysis, detecting bugs and fixing them for Ruby language is not easily doable by the current models, nor can they classify the codes as buggy or non-buggy. Even though machine learning models achieved higher scores, they still have scores below 60% for bug detection. Similarly, as discussed above, though models achieved high BLEU scores, this is not a reliable metric to ensure that the generated code is executable. These issues mainly arise from the lack of benchmarks and the need for extended datasets. We recommend the following directions for the researcher.

Insight 1: Benchmark datasets must ensure the generated fixed codes are executable. These benchmarks should have test cases, following a similar approach as HumanEval or other code generation benchmarks. As Ruby is not a high resource, this task might require code translation from existing works to Ruby.

Insight 2: The APR task requires a clear understanding of the project context and cannot be effectively learned by simply providing buggy code alongside its corresponding fix. Future studies could explore alternative methods for collecting datasets. For instance, researchers could link code commits to their related bug reports (such as GitHub issues) and utilize labels on these issues to identify bug-fixing commits. Additionally, leveraging metadata from datasets, as seen in CodeNet [41] could help recognize the project contexts associated with buggy and corrected code.

Insight 3: In addition to datasets, new methods are required for bug detection and APR tasks for the Ruby language. These techniques should be tailored specifically to the Ruby language to improve the performance and could vary. For example, techniques can be developed to detect noise in the dataset or integrate the business logic/project concept in the models by incorporating retrieval-augmented generated (RAG) [42] approaches or prompt engineering or prompt tuning with LLMs that direct the model to include a Ruby-specific context.

Insight 4: Bug detection using machine learning algorithms demonstrated better performance, and their score did not decrease much in the cross-project setting. This might suggest the robustness of these algorithms for this task. Researchers can rely on computationally cheaper models, i.e., machine learning models when developing techniques for bug detection in Ruby. Similar results were seen in text classification where XGBoost outperforms LLMs, such as GPT-4, in text classification task [43].

VI. Related Work

This section provides a summary of the research conducted so far on the downstream tasks being considered. It covers a wide range of relevant studies and explores the intricacies of the research, providing valuable insights into the current state of the field.

A. Bug detection

The field of software engineering has been grappling with identifying bugs in code using various methods such as static analysis and machine learning. Different representations of the bugs, such as abstract syntax trees or code snippets, have been used before feeding them into the models [36], and empirical evidence has shown no significance in a particular form of input given to the model [44]. A study introduced a new model called DexBERT [45], based on the traditional BERT [46] transformer representation that does bug detection and outperforms the conventional deep learning method called smali2vec. Despite promising results, there has been a noticeable lack of research on solving this downstream task using PLMs, unlike many other downstream software engineering tasks [4]. Our research aims to address this gap in the literature and provide insights into this unexplored yet longstanding area of interest.

B. APR

A lot of traditional research has been done on APR over time. The current papers on APR on PLMs [9] use datasets from state-of-the-art deep learning/machine learning papers, such as CoCoNut [47] CodeRep [48] and MegaDiff [49].

There has been work evaluating and fine-tuning Pre-Trained Language Models for APR [32]. Across four APR benchmarks, the study finds that the best-performing PLM, when unaltered, outperforms state-of-the-art deep-learning (DL)–based APR techniques by 72% in bug fixing. Moreover, the study addresses the efficiency of different PLMs in terms of size, time, and memory usage, offering promising directions for improving PLMs and advocating for transparent reporting practices to mitigate data leakage concerns in future evaluations. Different sizes were considered to feed into the model; the most noteworthy was using Simple Stupid Bugs to train CodeBert to solve APR task [9]. Previous research has demonstrated that APR can be effectively addressed by PLMs with satisfactory performance. By employing fully fine-tuning and Parameter-Efficient Fine-Tuning (PEFT) [33] approaches, we can discover unexplored opportunities in low-resource programming languages. This paves the way for us to experiment with innovative techniques and push the boundaries of what can be achieved in this field.

VII. Threats to Validity

We discuss the potential threats to provide more guidance on the interpretation, limitation, and other alternatives of the empirical experiment.

A. Construct Validity

This threat relates to the bug-fixing commit identification. The reliance on RegEx to classify bug-fixing commits could lead to misclassification. While manual validation estimated a 92% precision, false positives might remain, affecting the quality of the dataset. We however, followed the practices from previous studies to ensure a reliable data collection.

Another threat to validity is the assumption that code becomes bug-free after a bug-fixing commit. In this study, we

relied on this assumption to construct a dataset of bug-fix pairs, treating the code post-bug-fixing commit as a "fixed" version. However, this may not always be accurate, as undetected bugs might persist in the fixed code. To address this limitation, we applied stringent filtering criteria. For instance, we excluded methods associated with future bug-fixing commits and ensured the code remained unchanged for at least two years after the bug-fixing commit. These measures were intended to increase confidence that the code was truly fixed. Despite these efforts, the possibility of residual bugs in the fixed code cannot be entirely eliminated. Such undetected bugs could introduce noise into the dataset and affect the reliability of model evaluation. Future work could focus on improving the methodology for verifying the correctness of fixed code, refining filtering criteria, and aligning the dataset more closely with the needs of bug detection tasks.

B. External Validity

This relates to the representativeness of datasets. In this work, the dataset focuses exclusively on Ruby projects from GitHub, which may not represent bugs and fixes from other closed-source projects, limiting the generalizability of the findings to other programming languages. Therefore, we cannot make a sound claim on the effectiveness of projects in another programming language, given we have not tested on other scenarios.

C. Internal Validity

Internal threats refer to factors arising from internal considerations. In this study, we set the hyperparameter values for fine-tuning the PLM based on prior research and experimented with various configurations to identify the optimal settings for the machine learning algorithms. While this approach is widely accepted, there remains a possibility that the selected values may not represent the optimal configurations for the models and algorithms. However, based on the low obtained scores, we believe changing the parameters would not have a significant boost in the performance.

D. Conclusion Validity

This relates to the potential risk that the manual validation uses 100 commits. While useful, 100 commits may not be representative of the entire dataset, leading to potential overestimation or underestimation of dataset quality. For all tests, we used widely used parameters for the bug detection and APR tasks. For APR, we were unable to apply Pass@K to ensure the fixed code is executable, due to the lack of availability of such dataset that also includes tests cases. We emphasize however, as discussed in the previous section, that high BLEU scores do not mean that models are able to fix code reliably, and other techniques are required to improve the performance of APR for Ruby.

VIII. Conclusion

This study addresses the underexplored challenges of bug detection and automated program repair in low-resource programming languages, focusing on Ruby. Leveraging pre-trained language models and large language models, we conducted a comprehensive analysis of these tasks using our newly introduced dataset, Defects4Ruby. Our contributions to this study are 1) the Defects4Ruby dataset, mined from GitHub using established APR methodologies, which enables the study of software bug resolution in Ruby, and 2) the insights in evaluating PLMs and LLMs for this dataset in Ruby. Our findings reveal that while PLMs and LLMs show potential for APR, their performance in bug detection remains suboptimal, with traditional machine learning models performing competitively on certain metrics. This highlights a critical need for new techniques tailored to low-resource languages like Ruby. Specifically, PLMs such as CodeT5 demonstrate promising capabilities for APR, achieving significantly higher performance than LLMs. However, bug detection scores falling below 60% indicate substantial room for improvement.

To summarize, our study underscores the limitations of existing approaches and opens avenues for future research, including the development of domain-specific models, improved data augmentation techniques, and a deeper exploration of fine-tuning and prompting strategies for bug-related tasks. The study contributes to the existing research gap by providing insights into two complex downstream tasks in software bug resolution for Ruby. Our findings and the open-sourced Defects4Ruby dataset will serve as valuable resources for researchers and practitioners working to address software engineering challenges in low-resource languages, specially Ruby.

IX. Data Availability

The data and scripts to reproduce our results are available in the replication package.[2]

References

[1] K. Huang, B. Chen, C. Xu, Y. Wang, B. Shi, X. Peng, Y. Wu, and Y. Liu, "Characterizing usages, updates and risks of third-party libraries in java projects," *Empirical Software Engineering*, vol. 27, no. 4, p. 90, 2022.

[2] Y. Kamei, E. Shihab, B. Adams, A. E. Hassan, A. Mockus, A. Sinha, and N. Ubayashi, "A large-scale empirical study of just-in-time quality assurance," *IEEE Transactions on Software Engineering*, vol. 39, no. 6, pp. 757–773, 2012.

[3] C. Jones and O. Bonsignour, *The Economics of Software Quality*, 1st ed. Addison-Wesley Professional, 2011.

[4] X. Hou, Y. Zhao, Y. Liu, Z. Yang, K. Wang, L. Li, X. Luo, D. Lo, J. Grundy, and H. Wang, "Large language models for software engineering: A systematic literature review," 2023.

[5] D. Yan, Z. Gao, and Z. Liu, "A closer look at different difficulty levels code generation abilities of chatgpt," in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2023, pp. 1887–1898.

[6] M. Weyssow, X. Zhou, K. Kim, D. Lo, and H. Sahraoui, "Exploring parameter-efficient fine-tuning techniques for code generation with large language models," 2024.

[7] T. Ahmed, K. S. Pai, P. Devanbu, and E. T. Barr, "Improving few-shot prompts with relevant static analysis products," *arXiv preprint arXiv:2304.06815*, 2023.

[8] Y. Wu, Z. Li, J. M. Zhang, M. Papadakis, M. Harman, and Y. Liu, "Large language models in fault localisation," *arXiv preprint arXiv:2308.15276*, 2023.

[2]https://osf.io/yd57t/?view_only=dbd817dd5d564bf0b1c4b9efb72c9095

[9] E. Mashhadi and H. Hemmati, "Applying codebert for automated program repair of java simple bugs," in *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, 2021, pp. 505–509.

[10] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou, "Codebert: A pre-trained model for programming and natural languages," 2020.

[11] H. Husain, H.-H. Wu, T. Gazit, M. Allamanis, and M. Brockschmidt, "Codesearchnet challenge: Evaluating the state of semantic code search," 2020.

[12] S. Chandran and K. Abraham, "A correlative scrutiny on two programming dialects: Ruby vs python," *International Journal of Engineering and Advanced Technology*, vol. 9, pp. 4395–4404, 02 2020.

[13] S. Kaleba, O. Larose, R. Jones, and S. Marr, "Who you gonna call: analyzing the run-time call-site behavior of ruby applications," in *Proceedings of the 18th ACM SIGPLAN International Symposium on Dynamic Languages*, 2022, pp. 15–28.

[14] GitHub Blog, "The state of open source and ai," https://github.blog/2023-11-08-the-state-of-open-source-and-ai/, November 2023.

[15] Y. Zhou, S. Liu, J. Siow, X. Du, and Y. Liu, "Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks," 2019.

[16] J. P. Near and D. Jackson, "Finding security bugs in web applications using a catalog of access control patterns," in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 947–958. [Online]. Available: https://doi.org/10.1145/2884781.2884836

[17] S. Chowdhury, G. Uddin, H. Hemmati, and R. Holmes, "Method-level bug prediction: Problems and promises," *ACM Transactions on Software Engineering and Methodology*, vol. 33, no. 4, pp. 1–31, 2024.

[18] M. Tufano, C. Watson, G. Bavota, M. D. Penta, M. White, and D. Poshyvanyk, "An empirical study on learning bug-fixing patches in the wild via neural machine translation," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 28, no. 4, pp. 1–29, 2019.

[19] R. Dandamudi and G. Rodriguez-Perez, "A preliminary study of multilingual code language models for code generation task using translated benchmarks," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering Workshops*, ser. ASEW '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 94–99. [Online]. Available: https://doi.org/10.1145/3691621.3694939

[20] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. de Oliveira Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, R. Puri, G. Krueger, M. Petrov, H. Khlaaf, G. Sastry, P. Mishkin, B. Chan, S. Gray, N. Ryder, M. Pavlov, A. Power, L. Kaiser, M. Bavarian, C. Winter, P. Tillet, F. P. Such, D. Cummings, M. Plappert, F. Chantzis, E. Barnes, A. Herbert-Voss, W. H. Guss, A. Nichol, A. Paino, N. Tezak, J. Tang, I. Babuschkin, S. Balaji, S. Jain, W. Saunders, C. Hesse, A. N. Carr, J. Leike, J. Achiam, V. Misra, E. Morikawa, A. Radford, M. Knight, M. Brundage, M. Murati, K. Mayer, P. Welinder, B. McGrew, D. Amodei, S. McCandlish, I. Sutskever, and W. Zaremba, "Evaluating large language models trained on code," 2021.

[21] O. Dabic, E. Aghajani, and G. Bavota, "Sampling projects in github for MSR studies," in *18th IEEE/ACM International Conference on Mining Software Repositories, MSR 2021*. IEEE, 2021, pp. 560–564.

[22] GitHub. (2024) Fork a repository. Github. [Online]. Available: https://docs.github.com/en/pull-requests/collaborating-with-pull-requests/working-with-forks/fork-a-repo

[23] D. Spadini, M. Aniche, and A. Bacchelli, *PyDriller: Python Framework for Mining Software Repositories*, 2018.

[24] GitHub. (2024) About git cherry-pick. Github. [Online]. Available: https://docs.github.com/en/desktop/managing-commits/cherry-picking-a-commit-in-github-desktop

[25] F. Madeiral, T. Durieux, V. Sobreira, and M. Maia, "Towards an automated approach for bug fix pattern detection," *arXiv preprint arXiv:1807.11286*, 2018.

[26] H. Li, G. K. Rajbahadur, D. Lin, C.-P. Bezemer, and Z. M. Jiang, "Keeping deep learning models in check: A history-based approach to mitigate overfitting," *IEEE Access*, 2024.

[27] Y. Wang, W. Wang, S. Joty, and S. C. Hoi, "CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation," in *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, M.-F. Moens, X. Huang, L. Specia, and S. W.-t. Yih, Eds. Online and Punta Cana, Dominican Republic: Association for Computational Linguistics, Nov. 2021, pp. 8696–8708. [Online]. Available: https://aclanthology.org/2021.emnlp-main.685

[28] W. Tang, M. Tang, M. Ban, Z. Zhao, and M. Feng, "Csgvd: A deep learning approach combining sequence and graph embedding for source code vulnerability detection," *Journal of Systems and Software*, vol. 199, p. 111623, 2023. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121223000183

[29] B. Rozière *et al.*, "Code llama: Open foundation models for code," 2023, arXiv:2308.12950 [cs.CL].

[30] OpenAI, "gpt-4o," https://openai.com/index/hello-gpt-4o/, 2024.

[31] A. Radford and K. Narasimhan, "Improving language understanding by generative pre-training," 2018. [Online]. Available: https://api.semanticscholar.org/CorpusID:49313245

[32] N. Jiang, K. Liu, T. Lutellier, and L. Tan, "Impact of code language models on automated program repair," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, 2023, pp. 1430–1442.

[33] C. Wang, Y. Yang, C. Gao, Y. Peng, H. Zhang, and M. R. Lyu, "No more fine-tuning? an experimental evaluation of prompt tuning in code intelligence," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022, pp. 382–394.

[34] T. B. Brown, "Language models are few-shot learners," *arXiv preprint arXiv:2005.14165*, 2020.

[35] E. Ceylan, F. O. Kutlubay, and A. B. Bener, "Software defect identification using machine learning techniques," in *32nd EUROMICRO Conference on Software Engineering and Advanced Applications (EUROMICRO'06)*, 2006, pp. 240–247.

[36] S. Aleem, L. F. Capretz, and F. Ahmed, "Benchmarking machine learning technologies for software defect detection," 2015.

[37] T. Chen and C. Guestrin, "Xgboost: A scalable tree boosting system," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '16. ACM, Aug. 2016. [Online]. Available: http://dx.doi.org/10.1145/2939672.2939785

[38] L. Breiman, "Random forests," *Machine Learning*, vol. 45, no. 1, pp. 5–32, 10 2001. [Online]. Available: https://doi.org/10.1023/A:1010933404324

[39] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, "Bleu: a method for automatic evaluation of machine translation," in *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, 2002, pp. 311–318.

[40] S. Lu, D. Guo, S. Ren, J. Huang, A. Svyatkovskiy, A. Blanco, C. Clement, D. Drain, D. Jiang, D. Tang, G. Li, L. Zhou, L. Shou, L. Zhou, M. Tufano, M. Gong, M. Zhou, N. Duan, N. Sundaresan, S. K. Deng, S. Fu, and S. Liu, "Codexglue: A machine learning benchmark dataset for code understanding and generation," 2021.

[41] R. Puri, D. S. Kung, G. Janssen, W. Zhang, G. Domeniconi, V. Zolotov, J. Dolby, J. Chen, M. Choudhury, L. Decker, V. Thost, L. Buratti, S. Pujar, S. Ramji, U. Finkler, S. Malaika, and F. Reiss, "Codenet: A large-scale ai for code dataset for learning a diversity of coding tasks," 2021.

[42] P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Küttler, M. Lewis, W. tau Yih, T. Rocktäschel, S. Riedel, and D. Kiela, "Retrieval-augmented generation for knowledge-intensive nlp tasks," 2021. [Online]. Available: https://arxiv.org/abs/2005.11401

[43] M. Bohacek and M. Bravansky, "When XGBoost outperforms GPT-4 on text classification: A case study," in *Proceedings of the 4th Workshop on Trustworthy Natural Language Processing (TrustNLP 2024)*, A. Ovalle, K.-W. Chang, Y. T. Cao, N. Mehrabi, J. Zhao, A. Galstyan, J. Dhamala, A. Kumar, and R. Gupta, Eds. Mexico City, Mexico: Association for Computational Linguistics, Jun. 2024, pp. 51–60. [Online]. Available: https://aclanthology.org/2024.trustnlp-1.5

[44] C. Niu, C. Li, V. Ng, D. Chen, J. Ge, and B. Luo, "An empirical comparison of pre-trained models of source code," 2023.

[45] T. Sun, K. Allix, K. Kim, X. Zhou, D. Kim, D. Lo, T. F. Bissyandé, and J. Klein, "Dexbert: Effective, task-agnostic and fine-grained representation learning of android bytecode," *IEEE Transactions on Software Engineering*, vol. 49, no. 10, pp. 4691–4706, 2023.

[46] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," 2019.

[47] T. Lutellier, H. V. Pham, L. Pang, Y. Li, M. Wei, and L. Tan, "Coconut: combining context-aware neural translation models using ensemble for program repair," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 101–114. [Online]. Available: https://doi.org/10.1145/3395363.3397369

[48] Z. Chen and M. Monperrus, "The codrep machine learning on source code competition," 2018.
[49] M. Monperrus, M. Martinez, H. Ye, F. Madeiral, T. Durieux, and Z. Yu, "Megadiff: A dataset of 600k java source code changes categorized by diff size," 2021.