Detect-Localize-Repair: A Unified Framework for Learning to Debug with CodeT5

Nghi D. Q. Bui, Yue Wang, Steven C.H. Hoi
Salesforce Research Asia
{nghi.bui,wang.y,shoi}@salesforce.com

Abstract

Automated software debugging is a crucial task for improving the productivity of software developers. Many neural-based techniques have been proven effective for debugging-related tasks such as bug localization and program repair (or bug fixing). However, these techniques often focus only on either one of them or approach them in a stage-wise manner, ignoring the mutual benefits between them. In this work, we propose a novel unified *Detect-Localize-Repair* framework based on a pretrained programming language model CodeT5 to seamlessly address these tasks, named CodeT5-DLR. Specifically, we propose three objectives to adapt the generic CodeT5 for debugging: a bug detection objective to determine whether a given code snippet is buggy or not, a bug localization objective to identify the buggy lines, and a program repair objective to translate the buggy code to its fixed version. We evaluate it on each of these tasks and their combined setting on two newly collected line-level debugging datasets in Java and Python. Extensive results show that our model significantly outperforms existing baselines from both NLP and software engineering domains.

1 Introduction

Program debugging is crucial, yet most cost-dominating in software development. The goal of program debugging is to localize erroneous lines of a program (*bug localization*) and fixes this buggy patch (*program repair*). The majority of debugging tools falls into two categories: program analysis-based and neural-based. To debug a program, program analysis-based techniques employ compiler-based and software engineering theory to build code analysis tools. These methods have a significant disadvantage in that they are not scalable to large and complicated programs. On the other hand, a recent trend is to use neural-based techniques (Lutellier et al., 2020; Jiang et al., 2021; Zhu et al., 2021; Mashhadi and Hemmati, 2021; Ding et al., 2020; Wang et al., 2021) based on the *naturalness hypothesis* of software code (Hindle et al., 2016). They adopt a generic data-driven approach to train neural networks to automatically acquire bug-fix patterns through learning from a massive corpora of previous bug-fixes.

However, these techniques suffer from a few major drawbacks. First, they often utilize code-specific or language-specific features such as control flow, data flow, and abstract syntax trees (ASTs), which requires a significant amount of engineering effort for a careful design of code representations and thus hinders their applicability to more diverse domains or programming languages. Second, recent studies have focused on detecting bugs at coarse-grained code granularity such as function level or file level, which has been shown to be impractical in real-world use (Zou et al., 2019). It is also not ideal to localize bugs at too fine-grained level like the token level, which might lead to a large number of false positives (Allamanis et al., 2021). Line-level or statement-level bug localization, on the other hand, has been extensively studied in the domain of program analysis, such as spectrum-based bug localization (Abreu et al., 2009; Le et al., 2013; Xie et al., 2013; Abreu et al., 2007), in which the buggy statements are localized based on the signals of failed test cases, or mutation-based bug localization (Jia and Harman, 2010; Papadakis and Le Traon, 2015; Zhang et al., 2018), in which the buggy statements are localized by randomly mutating the statements and measuring against a test suite. However, these traditional techniques necessitate the execution of test cases in order to complete the bug localization process, which has scalability issues. As a result, we propose that line-level bug detection is more reasonable if we can use large-scale bug fixes datasets; and it corresponds to how human developers read and debug programs. Finally, these techniques are

only intended for either bug localization or program repair, or treat them separately as two stages, which fails to exploit their potential mutual benefits. Intuitively, a preciser bug detector is able to inform the repairer with more accurate buggy information to aid the bug fixing, while a good repairer usually has strong code understanding that is also required in bug detection.

To address these issues, we propose a unified framework for adapting a general pretrained programming language model for line-level debugging and repair. Our framework is based on a key observation about how programmers debug their code. First, he or she must determine whether or not a function is buggy. If it is buggy, the developer must localize the problematic line and provide a patch (repair). Inspired from this procedure, we propose three fine-tuning objectives on top of a pretrained language model for debugging. Using pretrained language models for code has two advantages. First, by treating code as natural language, it reduces the effort of the code representation engineering process. Second, it can leverage the pretrained knowledge gained from large number of source code. We employ CodeT5 (Wang et al., 2021) as the foundation model which has achieved state-of-the-art results on a wide range of code intelligence tasks. CodeT5 is pretrained on large-scale code corpus collected from Github using code-aware objectives, which endows the model with strong code understanding capability.

The first objective is the function-level bug detection task, which entails determining whether or not a particular piece of code includes a bug (D). Second, in order to give developers with more valuable information at a finer-grained code granularity, we propose a bug localization aim to identify the exact lines of code that include bugs (L). The third ojective is the program repair, which is used to convert the buggy code to the correct code (if applicable) (R). We expect that these tasks will complement one another and culminate in a robust, all-encompassing software debugging tool capable of doing many debugging-related tasks. CodeT5-DLR is the model created by applying all of the fine-tuning objectives.

To evaluate on the whole debugging procedure (D-L-R), we newly collected two large-scale bug-fix datasets in Java and Python programming languages from Github commits, which is released to facilitate future research as part of our contribution.

We consider two types of bugs: single-line and multi-line. We evaluate our CodeT5-DLR on three separate debugging-related tasks: function-level bug detection, line-level bug localization, and program repair. Our evaluation results show that our model significantly outperforms existing baselines on all of the tasks. We further conduct ablation studies to demonstrate that jointly training with the three objectives yields better performance than separately training on each single task. Finally, we design a unified evaluation task to combine all of these tasks that is to mimic how developers localize and fix bugs in real-world scenario, where our model is able to correctly localize 33.93% buggy lines and repair 46.93% for a single line bug fix Java dataset.

Our major contributions are three-fold:

- We propose a unified Detect-Localize-Repair framework (CodeT5-DLR) based on CodeT5 to seamlessly solve three program debugging tasks: function-level bug detection, line-level bug localization and program repair.

- We introduce two newly collected large-scale line-level debugging datasets in Java and Python programming languages with useful information for future research, including the buggy line indicator; and the before-fixed version and the after-fixed version of code snippets.

- We conduct extensive evaluations on each of debugging tasks and their combined task, where our model outperforms existing baselines with a significant margin.

2 Method

As shown in Figure 1, we present a unified framework to jointly address three crucial tasks in program debugging: *bug detection* (whether a given code snippet contains bugs), *bug localization* (which lines are buggy), and *program repair* (how to repair bugs). We first define the input/output formulation of these tasks in § 2.1 and then revisit the foundation model CodeT5 in § 2.2, followed by introducing each of tasks in § 2.3 in detail.

2.1 Problem Definition

Let $\mathcal{D}$ be a program debugging dataset consisting of $|\mathcal{D}|$ triplets of (X, Y, F). X is the source program patch at function level and $Y = \{y_1, ..., y_L\}$ is its buggy labels for each line, where $y_i \in [0, 1]$
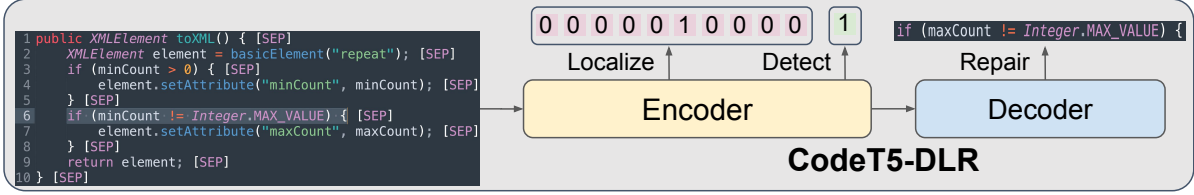
Figure 1: An overview of our CodeT5-DLR framework to jointly detect, localize, and repair bugs.

represents whether the i-th line is buggy or not and L denotes the number of lines in X. F is the target fixed program if source patch X contains any buggy lines, otherwise it is an empty string. Let y denote such function-level binary label and $y = 1$ if there exists $y_i = 1$ and else $y = 0$. For the input format of X, we insert a special token [SEP] for each line to inform the end of line information.

2.2 Revisiting CodeT5

CodeT5 (Wang et al., 2021) is a unified pretrained encoder-decoder language models for code. It was pretrained on a large-scale source code corpus collected from Github which consists of 8 different programming languages (including Java and Python). Moreover, CodeT5 proposed an identifier-aware pretraining objective to endow the model with code-specific knowledge. Besides that, it employs a code-specific Byte-Pair Encoding (BPE) (Sennrich et al., 2016) tokenizer that is able to avoid Out-of-Vocabulary (OoV) problems. CodeT5 has achieved state-of-the-art performance on a wide range of code intelligence tasks in CodeXGLUE (Lu et al., 2021) such as defect detection and code refinement. In this work, we adapt CodeT5 as our foundation model and propose a new unified framework to jointly solve bug localization and repair.

2.3 Detect-Localize-Repair Framework

In this subsection, we introduce how to adapt CodeT5 for bug detection, localization, and repair.

Function-Level Bug Detection The goal of this task is to detect whether a function contains any bugs. Given an input code patch X, we aim to learn the binary probability of $P_\theta(y|X)$ with CodeT5 parameterized by θ. Specifically, we pass the source patch X to the encoder of CodeT5 and adopt the last encoder state as the sequence representation of X, followed by a linear layer on top of it for a binary classification. This task is optimized with a standard cross entropy loss (denoted as $\mathcal{L}_{detect}$) in training and a patch is considered as buggy if the

predicted probability is higher than a threshold of 0.5 in inference.

Line-Level Bug Localization A further step of bug detection is to localize which exact lines are buggy. This is an important intermediate task for the final successful repair. This task is formulated to compute $P_\phi(Y|X)$, which ϕ denotes the parameter of the encoder of CodeT5. Specifically, we gather the last layer states of all [SEP] tokens from the encoder and map them to a vector of probabilities $\hat{Y} = \{\hat{y}_1, ..., \hat{y}_L\}$. We approach the bug localization as a sequence labeling task and apply the binary cross entropy loss (denoted as $\mathcal{L}_{localize}$) between $\hat{Y}$ and Y during training. For inference, we obtain the top-k predictions and measure how they match the ground truth with retrieval metrics. Notably, we consider two settings of bug localization where the source patch can have only one single buggy line or multiple buggy lines.

Program Repair This task aims to translate a buggy source patch X into its fixed version F. Formally, we aim to learn the probability $P_\theta(F|X) = \prod_{j=1}^{n} P_\theta(Y_j|X, F_1 : F_j - 1)$, where $F_1 : F_j - 1$ is the previous sequence before the j-th token and n denotes the number of tokens in the target sequence F. We approach this task a sequence-to-sequence problem and train with a standard sequence generation loss (denoted as $\mathcal{L}_{repair}$). During inference, we adopt beam search to generate a ranked list of fixed candidate patches.

Joint Training During training, we adopt multi-task learning to simultaneously optimize these three tasks by combining their losses in an end-to-end manner:

$$\mathcal{L}_{all} = \mathcal{L}_{detect} + \mathcal{L}_{localize} + \mathcal{L}_{repair} \quad (1)$$

The intuition behind this design is that these tasks are highly related and can complement to each other. For instance, a preciser bug locator can better inform the repairer with bug location to aid the bug fixing. Therefore, we expect these tasks can benefit from such a joint training paradigm.

3   Datasets

We collect two new datasets, one is the single line bug-fixes pair and the other is the multi-line bug-fixes pair. The single line bug-fixes (Karampatsis and Sutton, 2020) have been considered recently as one of the major issues that affects the quality of source code. These bugs can be fixed easily with simple code changes such as changing operators, renaming identifiers, swaping variables and so on. However, these bugs occurred frequently, and current static-based techniques are incapable of detecting them accurately (less than 10% in accuracy).

Existing datasets (Karampatsis and Sutton, 2020; Richter and Wehrheim, 2022), however, are not suitable for our purpose due to three reasons: **(1)** they contain only the code changes at the file level while our goal is to detect buggy lines at both function level and line level; **(2)** they does not contain the before and after function-level information of the code changes but only the patches at line-level; and **(3)** these datasets are mostly for single-line bug fixes, while our goal is to extend for a more realistic setting of multi-line bugs. With such reasons, we decide to collect datasets by ourselves for a comprehensive evaluation.

We follow similar steps from (Karampatsis and Sutton, 2020) to collect two datasets in Java and Python. Concretely, we extract bug-fixes code changes from Github commits, we use Pydriller (Spadini et al., 2018), a tool that mines software repositories from Git. To decide if a commit fixes a bug, we follow Karampatsis and Sutton (2020) to check if its commit message contains at least one of the keywords: *error*, *bug*, *fix*, *issue*, *mistake*, *incorrect*, *fault*, *defect*, *flaw*, and *type*. This heuristic has been shown to achieve 96% accuracy on a set of 300 manually verified commits (Ray et al., 2016) and 97.6% on a set of 384 manually verified commits (Tufano et al., 2018).

The code changes are made up of three parts of a source file: before changes, after changes, and the difference between the two (patch). However, because we want to localise bugs at the *function and line level* rather than the file level, we need to perform additional preprocessing to extract the code changes at the *function level*. We use Lizard [1] to extract the functions and compare the different

between the functions from the before and after version of a source file (obtained from Pydriller).

We end up with two datasets of different types and languages, one is for single-line bug-fixes in Java (**SL-Java**) and the other is the multi-lines bug-fixes in Python (**ML-Python**). For **SL-Java**, beside the code changes for bug fixes, we also follow Karampatsis and Sutton (2020) to use tree-sitter [2] to identify 13 bug patterns for the single buggy lines. We do not aim to detect the exact patterns because they can be easily detected using matching rules on ASTs if we can localize if a line is buggy. However, the patterns is useful in analysing how well our techniques work on each pattern, allowing us to gain a deeper understanding of the debugging process.

Table 1 shows the details of 13 bug patterns in our SL-Java dataset. We also provide a bug-fix sample for each of the pattern. Overall, the CHANGE_IDENTIFIER appears the most and SWAP_BOOLEAN_LITERAL appears the least.

Table 2 shows the statistics of our datasets, which have been splited into training, validation, testing sets. For function-level bug detection task, we use the whole code snippet. The buggy or non-buggy label is decided by simply treating the before version as the buggy (label 0) and the after version as non-buggy (label 1). For line-level bug localization task, we use the buggy line's number information to train the model to localize which line is buggy. For program repair, the before version is used as the source input and the after version is used as the target sequence.

Figure 2 shows a sample in our SL-Java dataset. A sample comprises primarily of the buggy code snippet (Before) and the fixed code snippet (After) (After). It also includes the line number of the defective line. Multiple buggy lines are present in the ML-Python samples as opposed to a single buggy line. Each instance also include other meta data, such as commit message, commit id (SHA1) and project name so that it is easy to trace back the original bug information.

4   Experiments

We employ the CodeT5-base (220M) [3] as the foundation model for our unified CodeT5-DLR framework. For the purpose of ablation study, we also consider three variants of our model:

---

[1] https://github.com/terryyin/lizard, a code analysis tool for extracting functions from source code files that supports multiple programming languages.

[2] https://github.com/tree-sitter/tree-sitter
[3] https://github.com/salesforce/CodeT5

Table 1: Detail of 13 Bug Patterns in SL-Java dataset.

| Index	Bug Pattern	Num Instances	Example	
			Before	Affter
P0	CHANGE OPERATOR	3784	logLevel>=Log.ASSERT	logLevel<=Log.ASSERT
P1	CHANGE_OPERAND	1089	x<=1	z<=1
P2	CHANGE_IDENTIFIER	15006	this.userDn	this.userName
P3	CHANGE_NUMERAL	8969	player.stepHeight=0.5F	player.stepHeight=0.6F
P4	CHANGE_CALLER_IN_FUNCTION	3180	mBlockStream.remaining()	inStream.remaining()
P5	CHANGE_UNARY_OPERATOR	2212	!segment.isOk()	segment.isOk()
P6	OVERLOAD_METHOD_MORE_ARGS	9732	Messaging.sendTr(sender,key)	Messaging.sendTr(sender,key,npc.getName())
P7	OVERLOAD_METHOD_DELETED_ARGS	2577	registerCommandsNow(commands)	registerCommandsNow()
P8	DIFFERENT_METHOD_SAME_ARGS	18914	server.getStartedLabel()	server.getStartedName()
P9	MORE_SPECIFIC_IF	3981	getIndex()>=arrayLength	arrayLength>0 && getIndex()>=arrayLength
P10	LESS_SPECIFIC_IF	3919	pluginId==null	pluginId==null || pluginID.length()==0
P11	SWAP_ARGUMENTS	1244	new Duration(DateTime.now(),time)	new Duration(time, DateTime.now()
P12	SWAP_BOOLEAN_LITERAL	897	doTest(false)	doTest(true)
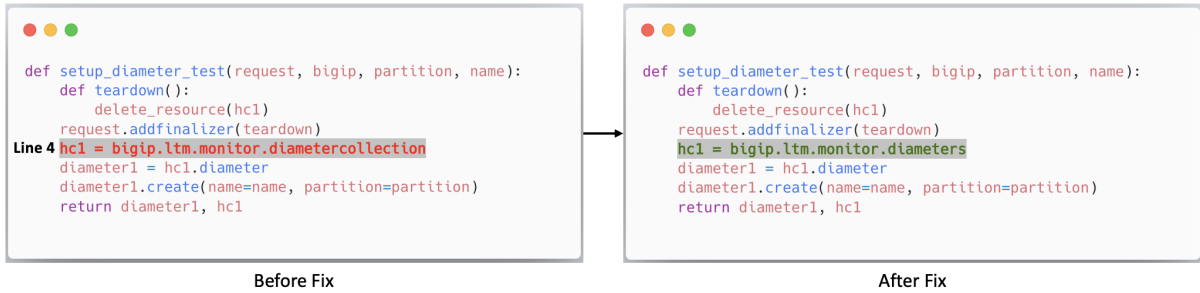
Figure 2: A sample of an instance in our SL-Java dataset

Table 2: Statistics of our datasets

Split	SL-Java		ML-Python	
	#Projects	#Instances	#Projects	#Instances
Train	2,348	52,789	4,428	132,243
Val	335	7,465	480	22,395
Test	700	15,250	867	35,457

CodeT5-D trained with $\mathcal{L}_{detect}$, CodeT5-L trained with $\mathcal{L}_{localize}$, and CodeT5-R trained with $\mathcal{L}_{repair}$. We set the maximum source and target sequence lengths to 512. All experiments are performed on NVIDIA A100 GPUs with 40 GB memory.

4.1 Function-Level Bug Detection

Metrics For this task, we use two metrics: the F1 score and the False Positive Rate (FPR). F1 is the standard metric for this type of task because it is a binary classification problem (buggy or not). The FPR, on the other hand, is critical for determining a bug localization system's usability in a real-world scenario. A good bug detection system should produce as few false positives as possible (Allamanis et al., 2021; Vasic et al., 2019). FPR is calculated as the ratio between the number of non-buggy functions wrongly categorized as buggy (false positives) and the total number of actual non-buggy functions.

Baselines The function-level bug detection can seen as the code classification task, i.e, assign a label to a given code snippet. We choose Tree-based CNN (Mou et al., 2016), a well-known method for code classification as a baseline. We also including some others SOTA pretrained language models of code, which are CodeBERT (Feng et al., 2020), GraphCodeBERT (Guo et al., 2020), and PLBART (Ahmad et al., 2021). We used their public checkpoints and fine-tune them for this task. We also include SpotBugs [4], a widely used static analysis-based baseline (Karampatsis and Sutton, 2020; Habib and Pradel, 2018) for bug detection task. For CodeT5, we use 3 baselines: CodeT5-L, CodeT5-D and CodeT5-DLR. CodeT5-R is not trained for bug detection but its output can also be used to detect bug [5].

Results Table 3 shows the results of function-level bug detection task. Our model fine-tuned with all 3 objectives together achieve the best performance in terms of both F1 score and FPR, while our CodeT5-D with the only function-level bug detection objective still yields better results than the baselines. The SpotBugs baseline achieves only

---

[4] https://github.com/spotbugs/spotbugs
[5] When perform training, we feed the model with both negative and positive samples so the PR module is able to decide to generate fixed code or not

Table 3: Performance of function-level bug detection. ↑: the higher the better, ↓: the lower the better.

Model	SL-Java		ML-Python	
	F1 ↑	FPR ↓	F1 ↑	FPR ↓
SpotBugs	4.6	89.29	-	-
TBCNN	45.49	38.03	51.24	48.20
CodeBERT	55.67	40.02	50.24	50.32
GraphCodeBERT	56.44	38.99	49.30	53.20
PLBART	59.01	35.21	52.33	51.24
CodeT5-R	50.94	40.29	52.93	46.45
CodeT5-D	59.28	34.32	52.93	46.45
CodeT5-DLR	**63.46**	**31.24**	**54.83**	**43.21**

4.6 in F1 score, which is consistent with the performance reported in Habib and Pradel (2018) for static analysis-based bug detector.

4.2 Line-Level Bug Localization

Metrics   We use 3 metrics for this task, which are Mean Reciprocal Rank (MRR), Mean Average Precision (MAP), and False Positive Rate (FPR). In reality, we do not know how many lines of code is buggy, so we retrieve top-k lines with highest scores and measure if the ground-truth buggy lines(s) belong to these top-k lines. As such, we can formulate this problem as an information retrieval problem, with the goal of returning a ranked list of relevant lines to a query (the query is to retrieve all of the buggy lines among the lines). For this reason, we use the well-known metrics of MRR and MAP to evaluate for this buggy line retrieval task. In our evaluation settings, each of these metrics will be appropriate for a different datasets. For the SL-Java, because there is only one buggy line in the ground truth, the MRR is appropriate for evaluating the performance of this dataset. On the other hand, each of sample in ML-Python contains numerous buggy lines in the ground truth, the MAP is better suited for ML-Python. With this, MRR and MAP are computed with respect to k, resulting in MRR@k and MAP@k, where k is number of lines retrieved for evaluation. We choose $k = 1$ and $k = 5$ for our evaluation.

In addition to MRR and MAP, we use False Positive Rate (FPR) to evaluate. Given a code snippet and retrieved buggy lines, the FPR in this case is calculated as the ratio between the number of non-buggy lines wrongly categorized as buggy (false positives) and the total number of actual non-buggy lines. We also compute FPR with respected to top-k lines retrieved, similar to MRR and MAP.

Baselines   For this task, we also chose baselines that are similar to the Function-level bug localization task, which are CodeBERT (Feng et al., 2020), GraphCodeBERT (Guo et al., 2020), and PLBART (Ahmad et al., 2021). In addition, we include 2 additional baselines that have been used to detect vulnerability in software engineering, which are DeepLineDP (Pornprasit and Tantithamtha-vorn, 2022) and LineVul (Fu and Tantithamtha-vorn, 2022). They work by simply performing prediction at the function level, then using attention scores from the backbone neural architecture to retrieve the line scores to predict vulnerability at line level. DeepLineDP is based on the Hierarchical Attention Network (Yang et al., 2016), which divides the source code into three layers: function, line, and token, with each level processed by a BiGRU neural network. LineVul is based on a vanilla Transformer (Vaswani et al., 2017), and its scores are calculated by averaging the token scores from the multi head attention layer. DeepLineDP and LineVul has not been used for bug localization before, but we try our best to adapt their software artifacts [6] [7] into our use case.

Results   The results of line-level bug localization task are shown in Table 4. When fine-tuning on all objectives, our model outperforms all baselines in terms of all metrics. The FPR is low when we only aim to detect one line of buggy code. When we increase k to 5, the FPR increases. However, the MRR and MAP are also better for either single-line bug detection (SL-Java) or multi-line bug detection (ML-Python) with k = 5. It means that as we broaden the scope of buggy line retrieval, the number of correctly detected bugs increases, but the model produces more false alarms. When performing bug localization at the line level, this is a trade off that must be made.

4.3 Program Repair

Metrics   We use 2 metrics for program repair: Exact Match (EM) and BLEU. For EM, if the generated program exactly matches the ground truth correct program, then EM=1, otherwise EM=0. BLEU score is a standard metric that is usually used to measure translation-based tasks.

Baselines   We also use CodeBERT (Feng et al., 2020), GraphCodeBERT (Guo et al., 2020), and

---

[6] https://github.com/awsm-research/DeepLineDP
[7] https://github.com/awsm-research/LineVul

Table 4: Performance of line-level bug localization. ↑: the higher the better and ↓: the lower the better.

Model	SL-Java				ML-Python			
	MRR@1 ↑	MRR@5 ↑	FPR@1 ↓	FPR@5 ↓	MAP@1 ↑	MAP@5 ↑	FPR@1 ↓	FPR@5 ↓
DeepLineDP	14.05	15.33	25.30	41.20	9.35	11.98	25.60	73.21
LineVul	15.34	16.79	23.59	39.45	10.46	14.24	27.46	67.22
CodeBERT	21.66	29.69	17.20	21.57	18.67	26.30	19.92	58.90
GraphCodeBERT	20.35	28.45	18.67	20.45	22.59	28.92	19.20	53.24
PLBART	23.02	30.98	12.93	14.67	23.22	30.56	13.98	46.22
CodeT5-L	24.40	32.01	7.33	10.39	24.59	32.40	13.67	42.44
CodeT5-DLR	**26.78**	**34.67**	**3.04**	**8.05**	**26.98**	**33.75**	**9.84**	**38.46**
CodeT5-DLR-new	**27.67**	**38.38**	**3.01**	**7.23**	-	-	-	-

Table 5: Performance of Program Repair task.

Model	SL-Java		ML-Python	
	EM	BLEU	EM	BLEU
CodeBERT	3.66	34.21	3.40	29.40
GraphCodeBERT	3.35	35.29	3.31	30.59
PLBART	6.02	40.12	5.39	33.92
CodeT5-R	7.30	40.20	6.01	35.54
CodeT5-DLR	**10.30**	**43.42**	**6.30**	**38.44**

PLBART (Ahmad et al., 2021) for the program repair task. We fine-tune these pretrained models with the $\mathcal{L}_{repair}$ objective to generate the fixed code from buggy code.

Results Table 5 shows the results of program repair task. Our model when fine-tuning on all 3 objectives achieves the best performance among the baselines with significant margins, both in terms of EM and BLEU. In overall, EM for ML-Python is lower than EM for SL-Java, it is because that it is more challenging to generate fixed code given that there are multi buggy lines in the buggy code.

4.4 Qualitative Analysis

4.4.1 End-to-End Bug Detection and Repair

We have shown that our model performs the best among all of the baselines for 3 tasks: function-level bug detection, line-level bug localization and program repair. However, since these 3 tasks are evaluate individually, they still do not reflect the full capability of our model in a unified manner, i.e., both detect bugs and suggest fixes. It also reflects how the developer debugs program in their daily work. We perform additional experiments to illustrate these steps in order. First, we use the function-level bug detection module to predict a set of buggy functions from the test set, regardless of whether they are buggy or not. Then, using the detected samples, we use the line-level bug localization module to identify buggy lines within

these detected samples (not all samples in the test set). We then use the program repair module to suggest fixes for these samples as well. Figure 3 shows a bug example that our model can detect, localize and repair. Note that this is a real example from an open source project [8] with 17K stars on Github. This is a real bug-fix commit with the commit message "*Minor fix in polyglot native API*". First, our function-level bug detection model can detect that this is a buggy line. Second, the line-level bug localization model ranks the line *contextBuilder.allowNativeAccess(allow_create_thread);* as the top-1 line that is buggy. Finally, the program repair module translates the whole buggy function to the fixed function, and the buggy part "allowCreateThread" is translated to the correct version "allowNativeAccess". This fix could be referred to the *CHANGE_CALLER_IN_FUNCTION* pattern, in which the invoked function of an object is changed.

Figure 4 shows another example. This is yet another bug found in a real-world project [9]. Our DLR process can also identify the correct buggy line but fails to recommend correct fixes. However, this is due to the fact that the fix is for the pattern *CHANGE_NUMERAL*, which is very challenging to know the exact numeral to replace (3476 to 3344). An enhancement to our technique is the ability to suggest fixes for the missing whole of a broken line, similar to Guo et al. (2021) for code completion. We leave this as a part of future investigation.

4.4.2 Analysis of Detected Bug Patterns

The SL-Java dataset contains bug pattern information, analyzing and understanding how different

---

[8] https://github.com/oracle/graal/commit/9059770c00748fac01a75bac6f30d074783d2e69

[9] https://github.com/apache/druid/commit/7ebe053ac1abce6e3b218beaee801ebbc6da2ecb

Figure 3: A *CHANGE_CALLER_IN_FUNCTION* bug that our Code-DLR can successfully detect and repair.

Figure 4: A *CHANGE_NUMERAL* bug that our Code-DLR can successfully detect but suggest wrong fixes.

bug patterns can be detected and fixed is critical for future debugging system improvement. We break down the performance by bug-pattern and computer the percentage of successfully detected bugs for each of the patterns in terms of F1 score based on the results of function-level bug detection in Table 3 We use 3 baselines for this analysis, which are: CodeBert, CodeT5-L. and CodeT5-DLR. Figure 5 illustrate such results. CodeT5-L performs better than CodeT5-DLR on an average of 4.1%, and is better than CodeBert on an average of 10.2% in terms of F1. For some patterns, such as P1(change operand), P2 (change identifier) and P5 (change caller in fuction), CodeT5-DLR perform much better than CoderBert (>14% on average) [10].

4.5 Analysis of End-to-End Bug Detection and Repair

In addition to the samples shown in Section 4.4.1, we also want to see how the other models compare to ours in this end-to-end process. However, because we discovered no baseline in the litera-

---

[10]Due to page number constraints, we were unable to display the pattern name on the bar chart; instead, we abstracted the pattern into some index, such as P0, P1, and so on. Readers are encouraged to view the explanations for each of the patterns in our supplementary materials.

ture that performs this unify task in one model, we were unable to compare with the other baselines. As such, we compare this to the models that are trained individually for each task and perform the same debugging steps as described. Table 6 shows that CodeT5-DLR continues to outperform its variants. Note that the performance of CodeT5-DLR this Table for line-level bug localization (BL) and program repair (PR) are different than the same model for the same tasks in Table 4 and Table 5.

Table 6: Performance of Unify Debugging Procedure. CodeT5-L performs worse than CodeT5-DLR for the line-level bug localization (BL) task in term of MRR@5. CodeT5-R also performs worse than CodeT5-DLR for the program repair (PR) task in term of BLEU.

Model	SL-Java		ML-Python	
	BL	PR	BL	PR
CodeT5-L	28.58	-	24.59	-
CodeT5-R	-	43.20	-	38.50
CodeT5-DLR	33.93	46.93	28.49	41.21

5 Discussion

In this section, we discuss the design choice of the model architecture, as well as some advantages and disadvantages in comparison to other techniques. A recently developed line of work also employs a joint objective function for bug localization and repair (Allamanis et al., 2021; Vinyals et al., 2015). However, the task they are attempting is not the same as ours. They aim to detect bugs at token level, where the token is considered as a missing slot in the code representation and the goal is to fill in such missing slot. A pointer network (Vinyals et al., 2015) is leveraged to predict whether a location is buggy or not. In fact, we can also design a pointer network to detect whether a line is buggy or not in the line-level bug localization objective (instead of the current ranking method). However, because we want to detect both buggy and non-buggy code, this design is ineffective in our case. Their method is based on the assumption that the buggy code is given, the goal is to locate the bug. We may end up feeding the pointer net a lot of non-buggy lines, which usually have a much larger number than buggy lines. Because of the imbalance between buggy and non-buggy lines, the pointer net design does not work well in our case (confirmed with experiments).

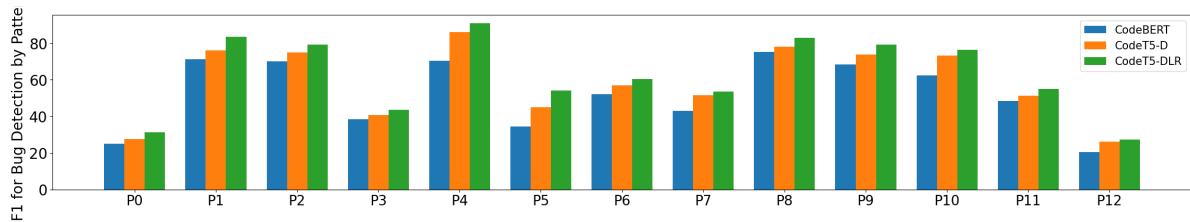In addition, their models are impractical in real-

Figure 5: Analysis of bug detection in F1 score, broken down by 14 bug patterns.

world use case due to high FPR(98% in Allamanis et al. (2021)). Also, the evaluation process of these techniques is mostly based on synthetic datasets, i.e., the bugs are generated using some heuristics, making the evaluation results unrealistic (He et al., 2022). By contrast, our work does not rely on synthetic bug data, but rather on real-world datasets from Github projects, making the results more useful in practice. In addition, the granularity of bugs we are targeting is different so that our technique cannot be directly compared to theirs.

6 Related Work

Pretrained Language Models for Code Recently, language models in natural language processing has been applied to model source code (Feng et al., 2020; Wang et al., 2021; Guo et al., 2020; Ahmad et al., 2021; Bui et al., 2021; Elnaggar et al., 2021; Peng et al., 2021; Kanade et al., 2020). CodeBERT (Feng et al., 2020) pretrain a model of code on multiple programming languages by adapting a Roberta model (Liu et al., 2019). CuBERT (Kanade et al., 2020) pretrains a BERT model for code using a large dataset of curated Python files. In general, most of the techniques treat code similar to texts and adapt the same pretraining strategies as for natural language. Some techniques, such as CodeT5 (Wang et al., 2021), encode source code features such as identifier information, data flow, and function name, among others, to pretrain code models, which may result in better overall performance.

Neural-based Bug Localization and Program Repair Bug localization and program repairs have received a lot of attention in terms of combining language models with traditional static analysis-based methods to improve performance. A recent trend is to generate synthetic simple bugs by rewriting rules into programs and then use self-supervised learning to train jointly models for bug localization and repair (Allamanis et al., 2021; Yasunaga and Liang, 2021, 2020; Vasic et al., 2019). However, these techniques are almost impractical

for real-world use case since they only target *simple bugs* and the models are trained mostly on synthetic data. There are also many recent neural-based techniques that target only program repairs (Lutellier et al., 2020; Zhu et al., 2021; Jiang et al., 2021; Chen et al., 2019; Li et al., 2020; Tufano et al., 2018). In contrast, our CodeT5-DLR aims to combine the strengths of each of these techniques in order to fine-tune a foundation model for jointly localizing bugs and repairing programs at a reasonable code granularity (function and line level).

7 Conclusion

We proposed a novel detect-localize-repair framework for jointly detecting bugs, localizing bugs and suggesting program repairs. Our model is built on the CodeT5 foundation model and is fine-tuned jointly to achieve three debugging-related objectives: function-level bug detection, line-level bug localization, and program repair. These three objectives are based on how software developers locate bugs and repair programs in their daily work. Our evaluation results show that training these 3 objectives together yields better results than fine-tuning on each objective individually. Furthermore, we also contribute to provide two new datasets that can be used for evaluating both bug localization and program repair tasks. Our datasets differ from existing datasets in that we provide the exact line that is buggy, as well as the before and after versions of a code snippet. We will make our datasets publicly available to facilitate research on this topic.

8   Limitations

There are still a few drawbacks of our technique that need further investigations. Our intention in designing these three objectives is that each of the two will assist the third in making the specific task for that objective better than when training individually. However, this does not provide a consistent signal across the three modules. For example, while the function-level bug detection module indicates that a function is not buggy, the program repair module continues to generate fixes. This inconsistency has been reflected in Table 3. A future research direction would be to design the training pipeline step by step, with the program repair module only providing repairs based on the bug detection module's signal. Second, we only use within-function information for the three tasks. A function in a program, on the other hand, is usually linked to other parts of the program or other functions within the same file. This information is commonly referred to as contexts, and it should be used as additional information when localizing bugs. We leave these investigations for the future work.

References

Rui Abreu, Peter Zoeteweij, and Arjan JC Van Gemund. 2007. On the accuracy of spectrum-based fault localization. In *Testing: Academic and industrial conference practice and research techniques-MUTATION (TAICPART-MUTATION 2007)*, pages 89–98. IEEE.

Rui Abreu, Peter Zoeteweij, and Arjan JC Van Gemund. 2009. Spectrum-based multiple fault localization. In *2009 IEEE/ACM International Conference on Automated Software Engineering*, pages 88–99. IEEE.

Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. 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, NAACL-HLT 2021, Online, June 6-11, 2021*, pages 2655–2668. Association for Computational Linguistics.

Miltiadis Allamanis, Henry Jackson-Flux, and Marc Brockschmidt. 2021. Self-supervised bug detection and repair. *Advances in Neural Information Processing Systems*, 34:27865–27876.

Nghi DQ Bui, Yijun Yu, and Lingxiao Jiang. 2021. Self-supervised contrastive learning for code retrieval and summarization via semantic-preserving transformations. In *Proceedings of the 44th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 511–521.

Zimin Chen, Steve Kommrusch, Michele Tufano, Louis-Noël Pouchet, Denys Poshyvanyk, and Martin Monperrus. 2019. Sequencer: Sequence-to-sequence learning for end-to-end program repair. *IEEE Transactions on Software Engineering*, 47(9):1943–1959.

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

Ahmed Elnaggar, Wei Ding, Llion Jones, Tom Gibbs, Tamas Feher, Christoph Angerer, Silvia Severini, Florian Matthes, and Burkhard Rost. 2021. Codetrans: Towards cracking the language of silicon's code through self-supervised deep learning and high performance computing. *arXiv preprint arXiv:2104.02443*.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 1536–1547, Online. Association for Computational Linguistics.

Michael Fu and Chakkrit Tantithamthavorn. 2022. Linevul: A transformer-based line-level vulnerability prediction.

Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, et al. 2020. Graphcodebert: Pre-training code representations with data flow. *arXiv preprint arXiv:2009.08366*.

Daya Guo, Alexey Svyatkovskiy, Jian Yin, Nan Duan, Marc Brockschmidt, and Miltiadis Allamanis. 2021. Learning to complete code with sketches. In *International Conference on Learning Representations*.

Andrew Habib and Michael Pradel. 2018. How many of all bugs do we find? a study of static bug detectors. In *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 317–328. IEEE.

Jingxuan He, Luca Beurer-Kellner, and Martin Vechev. 2022. On distribution shift in learning-based bug detectors. *arXiv preprint arXiv:2204.10049*.

Abram Hindle, Earl T Barr, Mark Gabel, Zhendong Su, and Premkumar Devanbu. 2016. On the naturalness of software. *Communications of the ACM*, 59(5):122–131.

Yue Jia and Mark Harman. 2010. An analysis and survey of the development of mutation testing. *IEEE transactions on software engineering*, 37(5):649–678.

Nan Jiang, Thibaud Lutellier, and Lin Tan. 2021. Cure: Code-aware neural machine translation for automatic program repair. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 1161–1173. IEEE.

Aditya Kanade, Petros Maniatis, Gogul Balakrishnan, and Kensen Shi. 2020. Learning and evaluating contextual embedding of source code. In *International Conference on Machine Learning*, pages 5110–5121. PMLR.

Rafael-Michael Karampatsis and Charles Sutton. 2020. How often do single-statement bugs occur? the manysstubs4j dataset. In *Proceedings of the 17th International Conference on Mining Software Repositories*, pages 573–577.

Tien-Duy B Le, Ferdian Thung, and David Lo. 2013. Theory and practice, do they match? a case with spectrum-based fault localization. In *2013 IEEE International Conference on Software Maintenance*, pages 380–383. IEEE.

Yi Li, Shaohua Wang, and Tien N Nguyen. 2020. Dlfix: Context-based code transformation learning for automated program repair. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pages 602–614.

Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692*.

Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, Ming Gong, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie LIU. 2021. CodeXGLUE: A machine learning benchmark dataset for code understanding and generation. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 1)*.

Thibaud Lutellier, Hung Viet Pham, Lawrence Pang, Yitong Li, Moshi Wei, and Lin Tan. 2020. 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*, pages 101–114.

Ehsan Mashhadi and Hadi Hemmati. 2021. Applying codebert for automated program repair of java simple bugs. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, pages 505–509. IEEE.

Lili Mou, Ge Li, Lu Zhang, Tao Wang, and Zhi Jin. 2016. Convolutional neural networks over tree structures for programming language processing. In *Thirtieth AAAI conference on artificial intelligence*.

Mike Papadakis and Yves Le Traon. 2015. Metallaxis-fl: mutation-based fault localization. *Software Testing, Verification and Reliability*, 25(5-7):605–628.

Dinglan Peng, Shuxin Zheng, Yatao Li, Guolin Ke, Di He, and Tie-Yan Liu. 2021. How could neural networks understand programs? In *International Conference on Machine Learning*, pages 8476–8486. PMLR.

Chanathip Pornprasit and Chakkrit Tantithamthavorn. 2022. Deeplinedp: Towards a deep learning approach for line-level defect prediction. *IEEE Transactions on Software Engineering*.

Baishakhi Ray, Vincent Hellendoorn, Saheel Godhane, Zhaopeng Tu, Alberto Bacchelli, and Premkumar Devanbu. 2016. On the" naturalness" of buggy code. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pages 428–439. IEEE.

Cedric Richter and Heike Wehrheim. 2022. Tssb-3m: Mining single statement bugs at massive scale. *arXiv preprint arXiv:2201.12046*.

Rico Sennrich, Barry Haddow, and Alexandra Birch. 2016. Neural machine translation of rare words with subword units. In *ACL (1)*. The Association for Computer Linguistics.

Davide Spadini, Maurício Aniche, and Alberto Bacchelli. 2018. PyDriller: Python framework for mining software repositories. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering - ESEC/FSE 2018*, pages 908–911, New York, New York, USA. ACM Press.

Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. 2018. An empirical investigation into learning bug-fixing patches in the wild via neural machine translation. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 832–837.

Marko Vasic, Aditya Kanade, Petros Maniatis, David Bieber, and Rishabh Singh. 2019. Neural program repair by jointly learning to localize and repair. *arXiv preprint arXiv:1904.01720*.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems*, 30.

Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. 2015. Pointer networks. *Advances in neural information processing systems*, 28.

Yue Wang, Weishi Wang, Shafiq R. Joty, and Steven C. H. Hoi. 2021. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. pages 8696–8708.

Xiaoyuan Xie, Tsong Yueh Chen, Fei-Ching Kuo, and Baowen Xu. 2013. A theoretical analysis of the risk evaluation formulas for spectrum-based fault localization. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 22(4):1–40.

Zichao Yang, Diyi Yang, Chris Dyer, Xiaodong He, Alex Smola, and Eduard Hovy. 2016. Hierarchical attention networks for document classification. In *Proceedings of the 2016 conference of the North American chapter of the association for computational linguistics: human language technologies*, pages 1480–1489.

Michihiro Yasunaga and Percy Liang. 2020. Graph-based, self-supervised program repair from diagnostic feedback. In *International Conference on Machine Learning*, pages 10799–10808. PMLR.

Michihiro Yasunaga and Percy Liang. 2021. Break-it-fix-it: Unsupervised learning for program repair. In *International Conference on Machine Learning*, pages 11941–11952. PMLR.

Jie Zhang, Lingming Zhang, Mark Harman, Dan Hao, Yue Jia, and Lu Zhang. 2018. Predictive mutation testing. *IEEE Transactions on Software Engineering*, 45(9):898–918.

Qihao Zhu, Zeyu Sun, Yuan-an Xiao, Wenjie Zhang, Kang Yuan, Yingfei Xiong, and Lu Zhang. 2021. A syntax-guided edit decoder for neural program repair. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 341–353.

Daming Zou, Jingjing Liang, Yingfei Xiong, Michael D Ernst, and Lu Zhang. 2019. An empirical study of fault localization families and their combinations. *IEEE Transactions on Software Engineering*, 47(2):332–347.