

On Distribution Shift in Learning-based Bug Detectors

Jingxuan He¹ Luca Beurer-Kellner¹ Martin Vechev¹

Abstract

Deep learning has recently achieved initial success in program analysis tasks such as bug detection. Lacking real bugs, most existing works construct training and test data by injecting synthetic bugs into correct programs. Despite achieving high test accuracy (e.g., >90%), the resulting bug detectors are found to be surprisingly unusable in practice, i.e., <10% precision when used to scan real software repositories. In this work, we argue that this massive performance difference is caused by a distribution shift, i.e., a fundamental mismatch between the real bug distribution and the synthetic bug distribution used to train and evaluate the detectors. To address this key challenge, we propose to train a bug detector in two phases, first on a synthetic bug distribution to adapt the model to the bug detection domain, and then on a real bug distribution to drive the model towards the real distribution. During these two phases, we leverage a multi-task hierarchy, focal loss, and contrastive learning to further boost performance. We evaluate our approach extensively on three widely studied bug types, for which we construct new datasets carefully designed to capture the real bug distribution. The results demonstrate that our approach is practically effective and successfully mitigates the distribution shift: our learned detectors are highly performant on both our test set and the latest version of open source repositories. Our code, datasets, and models are publicly available at <https://github.com/eth-sri/learning-real-bug-detector>.

1. Introduction

The increasing amount of open source programs and advances in neural code models have stimulated the initial suc-

¹Department of Computer Science, ETH Zurich, Switzerland. Correspondence to: Jingxuan He <jingxuan.he@inf.ethz.ch>.

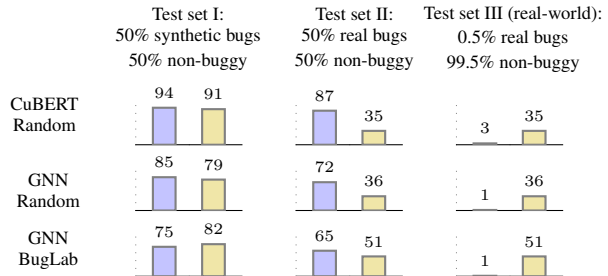


Figure 1: Performance of variable misuse classifiers are drastically reduced from synthetic test sets to a real-world test set. ■ and ■ are precision and recall, respectively.

cess of deep learning-based bug detectors (Vasic et al., 2019; Hellendoorn et al., 2020; Allamanis et al., 2021; Kanade et al., 2020; Chen et al., 2021b). These detectors can discover hard-to-spot bugs such as variable misuses (Allamanis et al., 2018) and wrong binary operators (Pradel & Sen, 2018), issues that greatly impair software reliability (Rice et al., 2017; Karampatsis & Sutton, 2020) and cannot be handled by traditional formal reasoning techniques.

Lacking real bugs, existing works build training sets by injecting few synthetic bugs, e.g., one (Kanade et al., 2020), three (Hellendoorn et al., 2020), or five (Allamanis et al., 2021), into each correct program. The learned bug detectors then achieve high accuracy on test sets created in the same way as the training set. However, when scanning real-world software repositories, these detectors were found to be highly imprecise and practically ineffective, achieving only 2% precision (Allamanis et al., 2021) or <10% precision (He et al., 2021). The key question then is: what is the root cause for this massive drop in performance?

Unveiling Distribution Shifts We argue that the root cause is *distribution shift* (Koh et al., 2021), a fundamental mismatch between the real bug distribution found in public code repositories and the synthetic bug distribution used to train and evaluate existing detectors. Concretely, real bugs are known to be different from synthetic ones (Yasunaga & Liang, 2021), and further, correct programs outnumber buggy ones in practice, e.g., around 1:2000 as reported in (Karampatsis & Sutton, 2020). This means that the real bug distribution inherently exhibits extreme data imbalance.

Figure 1 reproduces the performance drop, showing that existing detectors indeed fail to capture these two key factors. As in (Kanade et al., 2020), we fine-tune a classifier based on CuBERT for variable misuse bugs, using a balanced dataset with randomly injected synthetic bugs. Non-surprisingly, the fine-tuned model is close-to-perfect on test set I created in the same way as the fine-tuning set (top-left of Figure 1). Then, we replace the synthetic bugs in test set I with real bugs extracted from GitHub to create test set II (top-mid of Figure 1). The precision and recall drop by 7% and 56%, respectively, meaning that the model is significantly worse at finding real bugs. Next, we evaluate the classifier on test set III created by adding a large amount of non-buggy code to test set II so to mimic the real-world data imbalance. The model achieves a precision of only 3% (top-right of Figure 1). A similar performance loss occurs with graph neural networks (GNNs) (Allamanis et al., 2018), trained on either the dataset for fine-tuning the previous CuBERT model (mid row of Figure 1) or another balanced dataset where synthetic bugs are injected by BugLab (Allamanis et al., 2021), a learned bug selector (bottom row of Figure 1).

This Work: Alleviating Distribution Shifts In this work, we aim to alleviate such a distribution shift and learn bug detectors capturing the real bug distribution. To achieve this goal, we propose to train bug detectors in two phases: (1) on a balanced dataset with synthetic bugs, similarly to existing works, and then (2) on a dataset that captures data imbalance and contains a small number of real bugs, possible to be extracted from GitHub commits (Allamanis et al., 2021) or industry bug archives (Rice et al., 2017). In the first phase, the model deals with a relatively easier and larger training dataset. It quickly learns relevant features and captures the synthetic bug distribution. The second training dataset is significantly more difficult to learn from due to only a small number of positive samples and the extreme data imbalance. However, with the warm-up from the first training phase, the model can catch new learning signals and adapt to the real bug distribution. Such a two-phase learning process resembles the pre-training and fine-tuning scheme of large language models (Devlin et al., 2019) and self-supervised learning (Jing & Tian, 2021). The two phases are indeed both necessary: as we show in Section 5, without either of the two phases or mixing them into one, the learned detectors achieve sub-optimal bug detection performance.

To boost performance, our learning framework also leverages additional components such as task hierarchy (Sogaard & Goldberg, 2016), focal loss (Lin et al., 2017), and a contrastive loss term to differentiate buggy/non-buggy pairs.

Datasets, Evaluation, and Effectiveness In our work, we construct datasets capturing the real bug distribution. To the best of our knowledge, these datasets are among the

ones containing the largest number of real bugs (e.g., 1.7x of PyPIBugs (Allamanis et al., 2021)) and are the first to capture data imbalance, for the bug types we handle. We use half of these datasets for our second phase of training and the other two quarters for validation and testing, respectively. Our extensive evaluation shows that our method is practically effective: it yields highly performant bug detectors that achieve matched precision (or the precision gap is greatly reduced) on our constructed test set and the latest version of open source repositories. This demonstrates that our approach successfully mitigates the challenge of distribution shift and our dataset is suitable for evaluating practical usefulness of bug detectors.

2. Background

We now define the bug types our work handles and describe the state-of-the-art pointer models for detecting them.

Detecting Token-based Bugs We focus on token-based bugs caused by misuses of one or a few program tokens. One example is `var-misuse` where a variable use is wrong and should be replaced by another variable defined in the same scope. Formally, we model a program p as a sequence of n tokens $T = \langle t_1, t_2, \dots, t_n \rangle$. Fully handling a specific type of token-based bug in p involves three tasks:

- *Classification*: classify if p is buggy or not.
- *Localization*: if p is buggy, locate the bug.
- *Repair*: if p is buggy, repair the located bug.

Note that these three tasks form a dependency where the later task depends on the prior tasks. To complete the three tasks, we first extract $Loc \subseteq T$, a set of candidate tokens from T where a bug can be located. If $Loc = \emptyset$, p is non-buggy. Otherwise, we say that p is *eligible* for bug detection and proceed with the classification task. We assign a value in $\{\pm 1\}$ to p , where -1 (resp., 1) means that p is non-buggy (resp., buggy). If 1 is assigned to p , we continue with the localization and repair tasks. For localization, we identify a bug location token $t_{loc} \in Loc$. For repair, we apply simple rules (see Appendix B) to extract Rep , a set of candidate tokens that can be used to repair the bug located at t_{loc} and find a token $t_{rep} \in Rep$ as the final repair token. Loc and Rep are defined based on the specific type of bug to detect. For example with `var-misuse`, Loc is the set of variable uses in p and Rep is the set of variables defined in the scope of the wrong variable use. The above definition is general and applies to the three popular types of token-based bugs handled in this work: `var-misuse`, wrong binary operator (`wrong-binop`), and argument swapping (`arg-swap`). These bugs were initially studied in (Allamanis et al., 2018; Pradel & Sen, 2018). We provide examples of those bugs in Figure 2 and describe them in detail in Appendix B.

```

def compute_area(width, height):
    return width * width
def compute_area(width, height):
    return width + height
def buy_with(account):
    return lib.withdraw(120.0, account)
    
```

Figure 2: Example bugs handled by our work (left: var-misuse, middle: wrong-binop, right: arg-swap). The bug location and the repair token are marked as and , respectively.

Existing Pointer Models for Token-based Bugs State-of-the-art networks for handling token-based bugs follow the design of *pointer models* (Vasic et al., 2019), which identify bug locations and repair tokens based on predicted pointer vectors. Given the program tokens T , pointer models first apply an embedding method ϕ to convert each token t_i into an m -dimensional feature vector $h_i \in \mathbb{R}^m$:

$$[h_1, \dots, h_n] = \phi(\langle t_1, \dots, t_n \rangle).$$

Existing works instantiate ϕ as GNNs and GREATs (Hellendoorn et al., 2020), LSTMs (Vasic et al., 2019), or BERT (Kanade et al., 2020). Then, a feedforward network π^{loc} is applied on the feature vectors to obtain a probability vector $P^{loc} = [p_1^{loc}, \dots, p_n^{loc}]$ pointing to the bug location. The repair probabilities $P^{rep} = [p_1^{rep}, \dots, p_l^{rep}]$ are computed in a similar way with another feedforward network π^{rep} . We omit the steps for computing P^{loc} and P^{rep} for brevity and elaborate on them in Appendix A.

Importantly, in existing pointer models, classification is done *jointly* with localization. That is, each program has a special NO_BUG location (typically the first token). When the localization result points to NO_BUG, the classification result is -1 . Otherwise, the localization result points to a bug location and the classification result is 1.

Training Existing Pointer Models For training, two masks, C^{loc} and C^{rep} , are required as ground truth labels. C^{loc} sets the index of the correct bug location as 1 and other indices to 0. Similarly, C^{rep} sets the indices of the correct repair tokens as 1 and other indices to 0. The localization and repair losses are:

$$L^{loc} = - \sum_i p_i^{loc} \times C^{loc}[i],$$

$$L^{rep} = - \sum_i p_i^{rep} \times C^{rep}[i].$$

The additive loss $L = L^{loc} + L^{rep}$ is optimized. In Section 3, we introduce additional loss terms to L .

3. Learning Distribution-Aware Bug Detectors

Building on the pointer models discussed in Section 2, we now present our framework for learning bug detectors capable of capturing the real bug distribution.

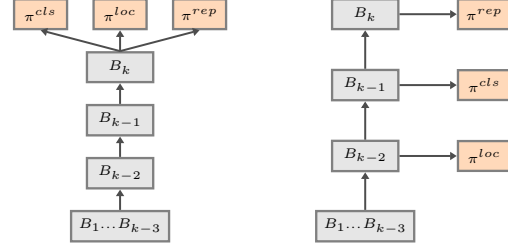


Figure 3: Standard feature sharing (left) v.s. an example of our task hierarchy (right).

3.1. Network Architecture with Multi-Task Hierarchy

We first describe architectural changes to the pointer model.

Adding Classification Head In our early experiments, we found that a drawback of pointer models is mixing the classification and localization results in one pointer vector. As a result, the model can be confused by the two tasks. We propose to perform the two tasks individually by adding a *binary classification head*: We treat the first token t_1 as the classification token $t_{[cls]}$ and apply a feedforward network $\pi^{cls} : \mathbb{R}^m \rightarrow \mathbb{R}^2$ over its feature vector $h_{[cls]}$ to compute the classification probabilities p_{-1}^{cls} and p_1^{cls} :

$$[p_{-1}^{cls}, p_1^{cls}] = \text{softmax}(\pi^{cls}(h_{[cls]})).$$

Task Hierarchy To exploit the inter-dependence of the *cls*, *loc*, and *rep* tasks, we formulate a *task hierarchy* for the pointer model. This allows the corresponding components to reinforce each other and improve overall performance. Task hierarchies are a popular multi-task learning technique (Zhang & Yang, 2021) and are effective in natural language (Søgaard & Goldberg, 2016) and computer vision (Guo et al., 2018). To the best of our knowledge, this work is the first to apply a task hierarchy on code tasks.

Using a task hierarchy, we process each task one by one following a specific order instead of addressing all tasks in the same layer. Formally, to encode a task hierarchy, we consider the feature embedding function ϕ to consist of k feature transformation layers B_1, \dots, B_k , which is a standard design of existing pointer models (Hellendoorn et al., 2020; Allamanis et al., 2021; Kanade et al., 2020):

$$\phi(T) = (B_k \circ B_{k-1} \circ \dots \circ B_1)(T).$$

We order our tasks *cls*, *loc* and *rep*, and apply their feedforward networks separately on the last three feature trans-

formation layers. Figure 3 shows a task hierarchy with the order $[loc, cls, rep]$ and compares it with feature sharing.

3.2. Imbalanced Classification with Focal Loss

To handle the extreme data imbalance in practical bug classification, we leverage the focal loss from imbalanced learning in computer vision and natural language processing (Lin et al., 2017; Li et al., 2020). Focal loss is defined as:

$$L^{cls} = \text{FL}([p_{-1}^{cls}, p_1^{cls}], y) = -(1 - p_y^{cls})^\gamma \log(p_y^{cls}),$$

where y is the ground truth label. Compared with the standard cross entropy loss, focal loss adds an adjusting factor $(1 - p_y^{cls})^\gamma$ serving as an importance weight for the current sample. When the model has high confidence with large p_y^{cls} , the adjusting factor becomes exponentially small. This helps the model put less attention on the large volume of easy, negative samples and focus on hard samples which the model is unsure about. γ is a tunable parameter that we set to 2 according to (Lin et al., 2017).

3.3. Two-phase Training

Bug detectors are ideally trained on a dataset containing a large number of real bugs as encountered in practice. However, since bugs are scarce, the ideal dataset does not exist and is hard to obtain with either manual or automatic approaches (He et al., 2021). Only a small number of real bugs can be extracted from GitHub commits (Allamanis et al., 2021; Karampatsis & Sutton, 2020) or industry bug archives (Rice et al., 2017), which are not sufficient for data intensive deep models. Our two-phase training overcomes this dilemma by utilizing both synthetic and real bugs.

Phase 1: Training with large amounts of synthetic bugs

In the first phase, we train the model using a dataset containing a large number of synthetic bugs and their correct version. Even though learning from such a dataset does not yield a final model that captures the real bug distribution, it drives the model to learn relevant features for bug detection and paves the way for the second phase. We create this dataset following existing work (Kanade et al., 2020): we obtain a large set of open source programs P , inject synthetic bugs into each program $p \in P$, and create a buggy program p' , which results in an 1 : 1 balanced dataset.

Since p and p' only differ in one or a few tokens, the model sometimes struggles to distinguish between the two, which impairs classification performance. We alleviate this issue by forcing the model to produce distant classification feature embeddings $h_{[cls]}$ for p and $h'_{[cls]}$ for p' . This is achieved by a contrastive loss term measuring the cosine similarity of $h_{[cls]}$ and $h'_{[cls]}$:

$$L^{contrastive} = \cos(h_{[cls]}, h'_{[cls]}).$$

The final loss in the first training phase is the addition of four loss terms: L^{cls} , L^{loc} , L^{rep} , and $\beta L^{contrastive}$, where β is the tunable weight of the contrastive loss.

Phase 2: Training with real bugs and data imbalance

The second training phase provides additional supervision to drive the model trained in phase 1 to the real bug distribution. To achieve this, we leverage a training dataset that mimics the real bug distribution. The dataset contains a small number of real bugs (typically hundreds) extracted from GitHub commits, which helps the model to adapt from synthetic bugs to real ones. Moreover, we add a large amount of non-buggy samples (e.g., hundreds of thousands) to mimic the real data imbalance. For more details on how we construct this dataset, please see Section 4. The loss for the second training phase is the sum of the three task losses L^{cls} , L^{loc} , and L^{rep} .

4. Implementation and Dataset Construction

In this section, we discuss the implementation of our learning framework and our dataset construction procedure.

Leveraging CuBERT We implement our framework on top of CuBERT (Kanade et al., 2020), a BERT-like model (Devlin et al., 2019) pretrained on source code. Still, our framework is general and can be applied to any existing pointer models (as we show in Section 5.1, our two-phase training brings improvement to the GNN model in (Allamanis et al., 2021)). We chose CuBERT mainly because, according to (Chen et al., 2021b), it achieves top-level performance on many programming tasks, including detecting synthetic `var-misuse` and `wrong-binop` bugs. Moreover, the reproducibility package provided by the CuBERT authors is of high quality and easy to extend. We present implementation details in Appendix B.

Dataset Construction We focus on Python code. Figure 4 shows our dataset construction process. Careful deduplication was applied throughout the entire process (Allamanis, 2019). After construction, we obtain a balanced dataset with synthetic bugs, called `syn-train`, used for the first training phase. Moreover, we obtain an imbalanced dataset with real bugs, which is randomly split into `real-train` (used for the second training phase), `real-val` (used as the validation set), and `real-test` (used as the blind test set). The split ratio is 0.5:0.25:0.25. Instead of splitting by files, we split the dataset by repositories. This prevents distributing files from the same repositories into different splits and requires generalization across codebases (Koh et al., 2021). Note that we do not evaluate on synthetic bugs as it does not reflect the practical usage. The statistics of the constructed datasets are given in Table 1.

Table 1: The statistics of our constructed dataset.

Bug Type	syn-train			real-train			real-val			real-test		
	repo	buggy	non-buggy	repo	buggy	non-buggy	repo	buggy	non-buggy	repo	buggy	non-buggy
var-misuse	2, 654	147, 409	147, 409	339	626	118, 888	169	347	63, 703	170	336	61, 539
wrong-binop	4, 944	150, 825	150, 825	368	872	73, 015	184	356	20, 341	185	491	41, 303
arg-swap	2, 009	157, 530	157, 530	372	469	82, 442	186	218	40, 305	185	246	48, 473

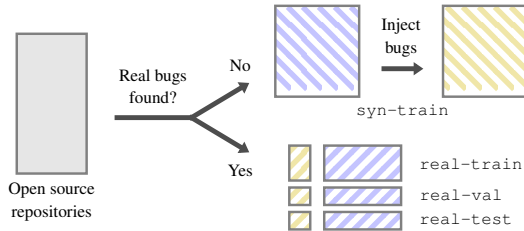


Figure 4: Our data construction process. For the precise sizes the datasets, refer to Table 1.

In Figure 4, we start the construction with a set of open source repositories (ETH Py150 Open (eth, 2022; Raychev et al., 2016) for `var-misuse` and `wrong-binop`, and on top of that 894 additional repositories for `arg-swap` to collect enough real bugs). We go over the commit history of the repositories and extract real bugs that align with the bug-inducing rewrite rules of (Allamanis et al., 2021) applied to both versions of a changed file. The repositories are then split into two sets depending on whether any real bug is found. To construct `syn-train`, we extract $\sim 150k$ eligible functions as non-buggy samples for each bug type from the repositories in which no real bugs were found (blue hatched). Then, we inject bugs into \square to create synthetic buggy samples (yellow hatched). The bugs are selected uniformly at random among all bug candidates. We leave the use of more advanced learning methods for bug selection (Patra & Pradel, 2021; Yasunaga & Liang, 2021) as future work. To construct `real-train`, we combine the found real bugs (red hatched) and other eligible functions from the same repositories, which serve as non-buggy samples (blue hatched). Since the number of eligible functions is significantly larger than the number of real bugs, real data imbalance is preserved. Finally, we perform random splitting to obtain `real-datasets` as discussed before.

5. Experimental Evaluation

In this section, we present an extensive evaluation of our framework. We first describe our experimental setup.

Training and Model We perform training per bug type because the three bug types have different characteristics. We tried a number of training configurations and found that our full method with all the techniques described in Section 3 performed the best on the validation set. For

`var-misuse`, `wrong-binop`, and `arg-swap`, the best orders for the task hierarchy are $[cls, loc, rep]$, $[rep, loc, cls]$, and $[loc, cls, rep]$, respectively. The best β for the contrastive loss are 0.5, 4, and 0.5, respectively. We provide training and model details in Appendix B.

Testing and Metrics We perform testing on the `real-test` dataset. For space reasons, we only discuss the testing results for `var-misuse` and `wrong-binop` in this section. We show the results for `arg-swap` in Appendix C.

Instead of accuracy (Allamanis et al., 2021; Hellendoorn et al., 2020; Vasic et al., 2019), we use precision and recall that are known to be better suited for data imbalance settings (wik, 2022; Saito & Rehmsmeier, 2015). They are computed per evaluation target `tgt` as follows:

$$P^{tgt} = \frac{tp^{tgt}}{tp^{tgt} + fp^{tgt}}, \quad R^{tgt} = \frac{tp^{tgt}}{\#buggy},$$

where $\#buggy$ is the number of samples labeled as buggy. When $tp^{tgt} + fp^{tgt} = 0$, we assign $P^{tgt} = 0$. We consider three evaluation targets `cls`, `cls-loc`, and `cls-loc-rep`:

- `cls`: binary classification. tp^{cls} means that the classification prediction and the ground truth are both buggy. A sample is an fp^{cls} when the classification result is buggy but the ground truth is non-buggy.
- `cls-loc`: joint classification and localization. A sample is a $tp^{cls-loc}$ when it is a tp^{cls} and the localization token is correctly predicted. A sample is an $fp^{cls-loc}$ when it is a fp^{cls} or a tp^{cls} with wrong localization result.
- `cls-loc-rep`: joint classification, localization, and repair. A sample is a $tp^{cls-loc-rep}$ when it is a $tp^{cls-loc}$ and the repair token is correctly predicted. A sample is an $fp^{cls-loc-rep}$ when it is a $fp^{cls-loc}$ or a $tp^{cls-loc}$ with wrong repair result.

The dependency between classification, localization, and repair determines how a bug detector is used in practice. That is, the users will look at the localization result only when the classification result is buggy. And the repair result is worth checking only when the classification returns buggy and the localization result is correct. Our metrics conform to this dependency by ensuring that the performance of the later target is bounded by the previous target.

During model comparison, we first compare precision and

Table 2: Changing training phases (var-misuse).

Method	cls		cls-loc		cls-loc-rep	
	P	R	P	R	P	R
Only Synthetic	3.43	35.42	2.45	25.30	2.10	21.73
Only Real	0	0	0	0	0	0
Mix	5.66	24.11	4.61	19.64	4.19	17.86
Two Synthetic	35.59	6.25	32.20	5.65	30.51	5.36
Our Full Method	64.79	13.69	61.97	13.10	56.34	11.90

Table 3: Changing training phases (wrong-binop).

Method	cls		cls-loc		cls-loc-rep	
	P	R	P	R	P	R
Only Synthetic	9.69	49.08	8.93	45.21	8.09	40.94
Only Real	47.74	25.87	45.49	24.64	42.11	22.81
Mix	12.97	41.55	12.02	38.49	10.93	35.03
Two Synthetic	26.85	5.91	25.00	5.50	21.30	4.68
Our Full Method	52.30	43.99	51.09	42.97	49.64	41.75

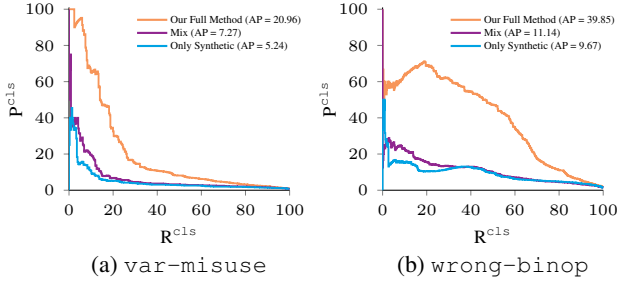


Figure 5: The effectiveness of our two-phase training demonstrated by precision-recall curves and AP.

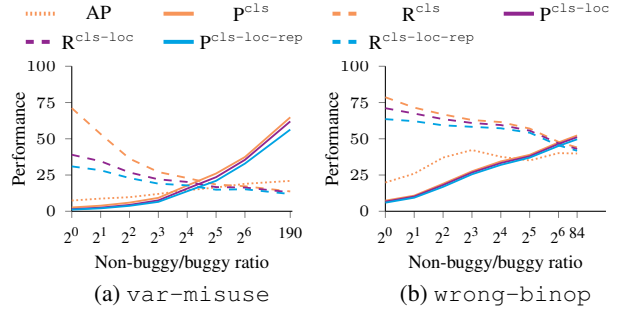


Figure 6: Model performance with various non-buggy/buggy ratios in the second training phase.

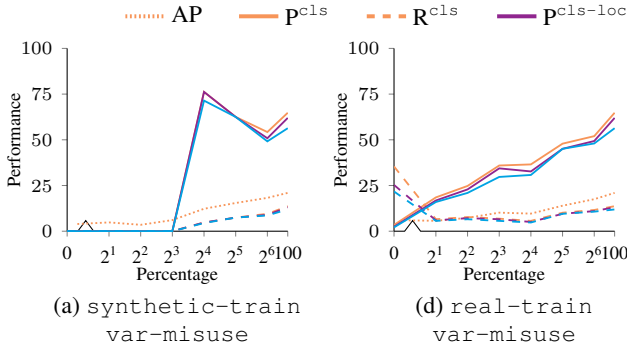


Figure 7: Model performance with subsampled syn-train or real-train.

recall without manual tuning of the thresholds to purely assess learnability. We prefer higher precision when the recall is comparable. This is because high precision reduces the burden of manual inspection to rule out false positives. When it is necessary to compare the full bug detection ability, we plot precision-recall curves by varying classification thresholds and compare average precision (AP).

5.1. Evaluation Results on Two-phase Training

We present the evaluation results on our two-phase training.

Changing Training Phases We create four baselines by only changing the training phases: (i) Only Synthetic: training only on syn-train; (ii) Only Real: training only on real-train; (iii) Mix: combining syn-train and real-train into a single training phase; (iv) Two Syn-

thetic: two-phase training, first on syn-train and then on a new training set constructed by replacing the real bugs in real-train with synthetic ones while maintaining the imbalance. They are compared with Our Full Method in Tables 2 and 3. We make the following observations:

- Unable to capture data imbalance, Only Synthetic is extremely imprecise (i.e., <10% precision), matching the results from previous works (Allamanis et al., 2021; He et al., 2021). Such imprecise detectors will flood users with false positives and are practically useless.
- For var-misuse, Only Real classifies all test samples as non-buggy. For wrong-binop, Only Real has significantly lower recall than Our Full Method. These results show that our first training phase can make the learning stable or improve model performance.

Table 4: Applying our two-phase training on GNN and BugLab (Allamanis et al., 2021) (var-misuse).

Model	Training Phases	cls		cls-loc		cls-loc-rep	
		P	R	P	R	P	R
GNN	Only Synthetic	1.31	36.31	0.57	15.77	0.41	11.31
GNN	Synthetic + Real	57.14	1.19	57.14	1.19	42.86	0.89
GNN	Only BugLab	1.16	50.60	0.35	15.48	0.22	9.82
GNN	BugLab + Real	66.67	0.60	66.67	0.60	66.67	0.60
Our Model	Synthetic + Real	64.79	13.69	61.97	13.10	56.34	11.90

Table 5: Applying our two-phase training on GNN and BugLab (Allamanis et al., 2021) (wrong-binop).

Model	Training Phases	cls		cls-loc		cls-loc-rep	
		P	R	P	R	P	R
GNN	Only Synthetic	5.59	42.57	4.79	36.46	3.64	27.70
GNN	Synthetic + Real	44.62	11.81	43.85	11.61	43.85	11.61
GNN	Only BugLab	3.10	55.60	2.08	37.27	1.47	26.48
GNN	BugLab + Real	51.80	32.18	51.15	31.77	50.82	31.57
Our Model	Synthetic + Real	52.30	43.99	51.09	42.97	49.64	41.75

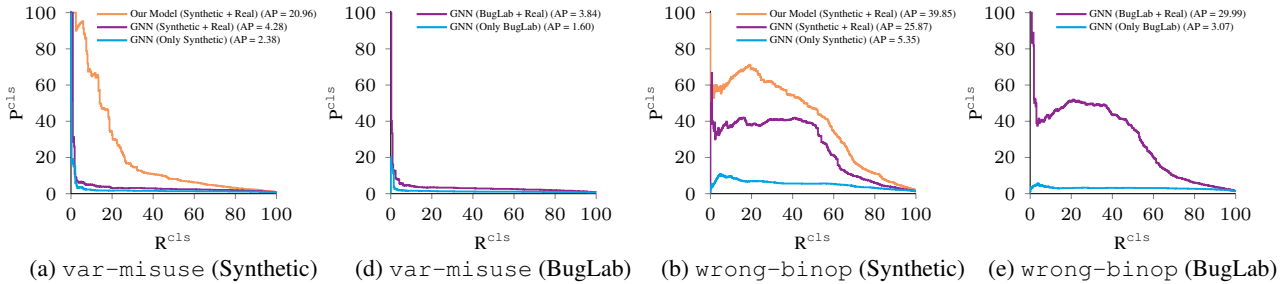


Figure 8: Precision-recall curves and AP for GNN and Our Model with different training phases.

- Mix does not perform well even if it is trained on real bugs. This is because, in the mixed dataset, synthetic bugs outnumber real bugs. As a result, the model does not receive enough learning signals from real bugs.
- By capturing data imbalance, Two Synthetic is more precise than Only Synthetic but sacrifices recall. Moreover, Our Full Method reaches significantly higher precision and recall than Two Synthetic, showing the importance of real bugs in training.

A precision-recall trade-off exists between Only Synthetic, Mix, and Our Full Method. To fully compare their bug classification capability, we plot their precision-recall curves and AP in Figure 5. The results show that Our Full Method significantly outperforms Only Synthetic and Mix with 3-4x higher AP. This means that our two-phase training helps the model generalize better to the real bug distribution.

Varying Data Imbalance Ratio To show how the data imbalance in the second training phase helps the model training, we vary the number of non-buggy training samples and keep the buggy training samples the same, resulting in different non-buggy/buggy ratios (2^{0-6} and the original ratio). The results are plotted in Figure 6, showing that the non-buggy/buggy ratio affects the precision-recall trade-off. Moreover, AP increases with data imbalance ratio. From 1:1 to the original ratio, AP increases from 7.40 to 20.96 for var-misuse and from 19.84 to 39.85 for wrong-binop.

Varying Amount of Training Data We also vary the size of our training sets syn-train and real-train. In each experiment, we subsample one training set (with percentage

0, 2, 4, 8, 16, 32, and 64) and fully use the other training set. The subsampling is done by repositories. The results are plotted in Figure 7. A general observation is that more training data, in either syn-train or real-train, improves AP. More data in syn-train increases R^{cls} . For var-misuse, the model starts to classify samples as buggy only when given a sufficient amount of data from syn-train. For wrong-binop, the amount of data in syn-train does not affect the precision. For real-train, we can make consistent observations across var-misuse and wrong-binop: first, more data in real-train improves P^{cls} ; second, as the amount of data in real-train increases from 0, the R^{cls} first decreases but then starts increasing from around 16%-32%.

Applying Two-phase Training to Existing Methods

Next, we demonstrate that our two-phase training method can benefit other methods. We consider BugLab, a learned bug selector for injecting synthetic bugs, and its GNN implementation (Allamanis et al., 2021). We train four GNN models with different training phases:

- Only Synthetic: train only on syn-train.
- Synthetic + Real: two-phase training same as ours, first on syn-train and then on real-train.
- Only BugLab: train only on a balanced dataset where bugs are created by BugLab.
- BugLab + Real: two-phase training, first on a balanced dataset created by BugLab and then on real-train.

In Tables 4 and 5, we show the results of the trained variants

Table 6: Evaluating other techniques (*var-misuse*).

Method	cls		cls-loc		cls-loc-rep	
	P	R	P	R	P	R
No cls Head	49.45	13.39	48.35	13.10	46.15	12.50
No Hierarchy	51.82	16.96	50.91	16.67	48.18	15.77
No Focal Loss	64.18	12.80	61.19	12.20	58.21	11.61
No Contrastive	60.00	14.29	57.50	13.69	53.75	12.80
Our Full Method	64.79	13.69	61.97	13.10	56.34	11.90

together with Our Model (Synthetic + Real) which corresponds to Our Full Method. Comparing models trained with Synthetic + Real, we can see that Our Model clearly outperforms GNN with significantly higher precision and recall. This is likely because Our Model starts from a CuBERT model pretrained on a large corpus of code. Moreover, compared with Only Synthetic, GNN trained with Synthetic + Real achieves significantly higher precision. The same phenomenon also applies when the first training phase is done with BugLab. We provide precision-recall curves and AP of the trained variants in Figure 8, showing that our second training phase can help GNN, trained with either Only Synthetic or Only BugLab, achieve higher AP, especially for *wrong-binop* bugs.

5.2. Evaluation Results on Other Techniques

We show the effectiveness of our other techniques with four baselines listed as follows. Each baseline excludes one technique as described below and keeps the other techniques the same as Our Full Method:

- No cls Head: no classification head. Classification and localization are done jointly like existing pointer models.
- No Hierarchy: no task hierarchy. All tasks are performed after the last feature transformation layer.
- No Focal Loss: use cross entropy loss for classification.
- No Contrastive: no contrastive learning with $\beta = 0$.

The above baselines have similar-level recall but noticeably lower precision than Our Full Method. This means all the evaluated techniques contribute to the high performance of Our Full Method. The classification head and task hierarchy play a major role for *var-misuse*. We provide results with different task orders and β values in Appendix C.

5.3. Scanning Latest Open Source Repositories

In an even more practical setting, we evaluate our method on the task of scanning the latest version of open source repositories. To achieve this, we obtain 1118 (resp., 2339) GitHub repositories for *var-misuse* and *wrong-binop* (resp., *arg-swap*). Those repositories do not overlap with the ones used to construct *syn-train* and *real-train*. We apply

Table 7: Evaluating other techniques (*wrong-binop*).

Method	cls		cls-loc		cls-loc-rep	
	P	R	P	R	P	R
No cls Head	48.20	43.58	47.30	42.77	45.95	41.55
No Hierarchy	48.70	45.62	47.83	44.81	46.52	43.58
No Focal Loss	49.32	44.60	48.42	43.79	47.97	43.38
No Contrastive	47.96	43.18	47.06	42.36	46.15	41.55
Our Full Method	52.30	43.99	51.09	42.97	49.64	41.75

Table 8: Manual inspection result on the reported warnings.

Bug Type	Bugs	Quality Issues	False Positives
<i>var-misuse</i>	50	10	40
<i>wrong-binop</i>	6	80	14
<i>wrong-binop-filter</i>	37	11	52
<i>arg-swap</i>	17	3	80

our full method on all eligible functions in the repositories, without any sample filtering or threshold tuning, and deduplicate the reported warnings together with the extracted real bugs. This results in 427 warnings for *var-misuse*, 2102 for *wrong-binop*, and 203 for *arg-swap*.

For each bug type, we manually investigate 100 randomly sampled warnings and, following (Pradel & Sen, 2018), categorize them into (i) *Bugs*: warnings that cause wrong program behaviors, errors, or crashes; (ii) *Code Quality Issues*: warnings that are not bugs but impair code quality (e.g., unused variables), or do not conform to Python coding conventions, and therefore should be raised and fixed; (iii) *False Positives*: the rest. To reduce human bias, two authors independently assessed the warnings and discussed differing opinions to reach an agreement. We show the inspection statistics in Table 8 and present case studies in Appendix D. Moreover, we report a number of bugs to the developers and the links to the bug reports are provided in Appendix E.

For *var-misuse*, most code quality issues are unused variables. For *wrong-binop*, most warnings are related to `==`, `!=`, `is`, or `is not`. Our detector flags the use of `==` and `!=` for comparing with `None`, and the use of `is` (resp., `is not`) for equality (resp., inequality) check with primitive types. Those behaviors, categorized by us as code quality issues, do not conform to Python coding conventions and even cause bugs in rare cases (sof, 2022). Our model learns to detect them because such samples exist as real bugs in *real-train*. Depending on the demand on code quality, users might want to turn off such warnings. We simulate this case by filtering out those behaviors and inspect another 100 random warnings from the 255 warnings after filtering. The results are shown in row *wrong-binop-filter* of Table 8. The bug ratio becomes significantly higher than the original version. For *arg-swap*, our model mostly detects bugs with Python standard library functions, such

as `isinstance` and `super`, or APIs of popular libraries such as TensorFlow. Most false positives are reported on repository-specific functions not seen during training.

The inspection results demonstrate that our detectors are performant and useful in practice. Counting both bugs and code quality issues as true positives, the precision either matches the evaluation results with `real-test` (`var-misuse` and `wrong-binop`) or the performance gap discussed in Section 1 is greatly reduced (`arg-swap`). This demonstrates that our method is able to handle the real bug distribution and our dataset can be reliably used for measuring the practical effectiveness of bug detectors.

6. Related Work

We now discuss works most closely related to ours.

Machine Learning for Bug Detection GNNs (Allamanis et al., 2018), LSTMs (Vasic et al., 2019), and GREAT (Hellendoorn et al., 2020) are used to detect `var-misuse` bugs. Deepbugs (Pradel & Sen, 2018) learns classifiers based on code embeddings to detect `wrong-binop`, `arg-swap`, and incorrect operands bugs. Hoppity (Dinella et al., 2020) learns to perform graph transformations representing small code edits. A number of models are proposed to handle multiple coding tasks including bug detection. This includes PLUR (Chen et al., 2021b), a unified graph-based framework for code understanding, and pre-trained code models such as CuBERT (Kanade et al., 2020) and CodeBert (Feng et al., 2020).

The above works mainly use datasets with synthetic bugs to train and evaluate the learned detectors. Some spend efforts on evaluation with real bugs but none of them completely capture the real bug distribution: the authors of (Vasic et al., 2019) and (Hellendoorn et al., 2020) evaluate their models on a small set of paired code changes from GitHub (i.e., buggy/non-buggy ratio is 1:1). The PyPIBugs (Allamanis et al., 2021) and ManySStuBs4J (Karampatsis & Sutton, 2020) datasets use real bugs from GitHub commits but do not contain non-buggy samples. Hoppity (Dinella et al., 2020) is trained and evaluated on small code edits in GitHub commits, which are not necessarily bugs and can be refactoring, version changes, and other code changes (Berabi et al., 2021). Compared with the above datasets, our datasets with real bugs are the closest to the real bug distribution so far.

Other works focus on complex bugs such as security vulnerabilities (Li et al., 2018; Zhou et al., 2019; Chen et al., 2022). We believe that the characteristics of bugs discussed in our work are general and extensible to complex bugs.

Distribution Shift in Bug Detection and Repair A few works try to create realistic bugs for training bug detectors

or fixers. BugLab (Allamanis et al., 2021) jointly learns a bug selector with the detector to create bugs for training. Since no real bugs are involved in the training process, it is unclear if the learned selector actually constructs realistic bugs. Based on code embeddings, SemSeed (Patra & Pradel, 2021) learns manually defined bug patterns from real bugs, to create new, realistic bugs, which can be used to train bug detectors. Unlike SemSeed, our bug detectors learn directly from real bugs which avoids one level of information loss. BIFI (Yasunaga & Liang, 2021) jointly learns a breaker for injecting errors to code and a fixer for fixing errors. Focusing on fixing parsing and compilation errors, BIFI assumes a perfect external error classifier (e.g., AST parsers and compilers), while our work learns a classifier for software bugs. Namer (He et al., 2021) proposes a similar two-step learning recipe for finding naming issues. Different from our work, Namer relies on manually defined patterns and does not benefit from training with synthetic bugs.

Neural Models of Code Apart from bug detection, neural models are adopted for a number of other code tasks including method name suggestion (Alon et al., 2019; Allamanis et al., 2016; Zügner et al., 2021), type inference (Wei et al., 2020; Allamanis et al., 2020), code editing (Brody et al., 2020; Yin et al., 2019), and program synthesis (Alon et al., 2020; Brockschmidt et al., 2019; Mukherjee et al., 2021). More recently, large language models are used to generate real-world code (Austin et al., 2021; Chen et al., 2021a).

Code Rewriting for Data Augmentation Semantics-preserving code rewriting is used for producing programs, e.g., for adversarial training of type inference models (Bielik & Vechev, 2020) and contrastive learning of code clone detectors (Jain et al., 2021). The rewritten and the original programs are considered to be similar. In the setting of bug detection, however, the programs created by bug-injection rules and the original ones should be considered by the model to be distinct (Patra & Pradel, 2021; Allamanis et al., 2021), which is captured by our contrastive loss.

7. Conclusion

In this work, we revealed a fundamental mismatch between the real bug distribution and the synthetic bug distribution used to train and evaluate existing learning-based bug detectors. To mitigate this distribution shift, we proposed a two-phase learning method combined with a task hierarchy, focal loss, and contrastive learning. Our evaluation demonstrates that the method yields bug detectors able to capture the real bug distribution. We believe that our work is an important step towards understanding the complex nature of bug detection and learning practically useful bug detectors.

References

- ETH Py150 Open Corpus, 2022. URL https://github.com/google-research-datasets/eth_py150_open.
- What is the difference between "is None" and "== None", 2022. URL <https://stackoverflow.com/questions/3257919/what-is-the-difference-between-is-none-and-none>.
- Wikipedia - Precision and Recall for Imbalanced Data, 2022. URL https://en.wikipedia.org/wiki/Precision_and_recall#Imbalanced_data.
- Allamanis, M. The adverse effects of code duplication in machine learning models of code. In Masuhara, H. and Petricek, T. (eds.), *Onward!*, 2019. URL <https://doi.org/10.1145/3359591.3359735>.
- Allamanis, M., Peng, H., and Sutton, C. A convolutional attention network for extreme summarization of source code. In *ICML*, 2016. URL <http://proceedings.mlr.press/v48/allamanis16.html>.
- Allamanis, M., Brockschmidt, M., and Khademi, M. Learning to represent programs with graphs. In *ICLR*, 2018. URL <https://openreview.net/forum?id=BJOFETxR->.
- Allamanis, M., Barr, E. T., Ducouso, S., and Gao, Z. Typilus: neural type hints. In *PLDI*, 2020. URL <https://doi.org/10.1145/3385412.3385997>.
- Allamanis, M., Jackson-Flux, H., and Brockschmidt, M. Self-supervised bug detection and repair. In *NeurIPS*, 2021. URL <https://arxiv.org/abs/2105.12787>.
- Alon, U., Zilberstein, M., Levy, O., and Yahav, E. code2vec: learning distributed representations of code. *Proc. ACM Program. Lang.*, 3(POPL):40:1–40:29, 2019. URL <https://doi.org/10.1145/3290353>.
- Alon, U., Sadaka, R., Levy, O., and Yahav, E. Structural language models of code. In *ICML*, Proceedings of Machine Learning Research, 2020. URL <http://proceedings.mlr.press/v119/alon20a.html>.
- Austin, J., Odena, A., Nye, M., Bosma, M., Michalewski, H., Dohan, D., Jiang, E., Cai, C. J., Terry, M., Le, Q. V., and Sutton, C. Program synthesis with large language models. *CoRR*, abs/2108.07732, 2021. URL <https://arxiv.org/abs/2108.07732>.
- Berabi, B., He, J., Raychev, V., and Vechev, M. Tfix: Learning to fix coding errors with a text-to-text transformer. In *ICML*, 2021. URL <http://proceedings.mlr.press/v139/berabi21a.html>.
- Bielik, P. and Vechev, M. Adversarial robustness for code. In *ICML*, 2020. URL <http://proceedings.mlr.press/v119/bielik20a.html>.
- Brockschmidt, M., Allamanis, M., Gaunt, A. L., and Polozov, O. Generative code modeling with graphs. In *ICLR*, 2019. URL <https://openreview.net/forum?id=Bke4KsA5FX>.
- Brody, S., Alon, U., and Yahav, E. A structural model for contextual code changes. *Proc. ACM Program. Lang.*, 4(OOPSLA):215:1–215:28, 2020. URL <https://doi.org/10.1145/3428283>.
- Chen, M., Tworek, J., Jun, H., Yuan, Q., de Oliveira Pinto, H. P., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., Ray, A., Puri, R., Krueger, G., Petrov, M., Khlaaf, H., Sastry, G., Mishkin, P., Chan, B., Gray, S., Ryder, N., Pavlov, M., Power, A., Kaiser, L., Bavarian, M., Winter, C., Tillet, P., Such, F. P., Cummings, D., Plappert, M., Chantzis, F., Barnes, E., Herbert-Voss, A., Guss, W. H., Nichol, A., Paino, A., Tezak, N., Tang, J., Babuschkin, I., Balaji, S., Jain, S., Saunders, W., Hesse, C., Carr, A. N., Leike, J., Achiam, J., Misra, V., Morikawa, E., Radford, A., Knight, M., Brundage, M., Murati, M., Mayer, K., Welinder, P., McGrew, B., Amodei, D., McCandlish, S., Sutskever, I., and Zaremba, W. Evaluating large language models trained on code. *CoRR*, abs/2107.03374, 2021a. URL <https://arxiv.org/abs/2107.03374>.
- Chen, Z., Hellendoorn, V. J., Lamblin, P., Maniatis, P., Manzagol, P.-A., Tarlow, D., and Moitra, S. Plur: A unifying, graph-based view of program learning, understanding, and repair. In *NeurIPS*, 2021b. URL <https://research.google/pubs/pub50846/>.
- Chen, Z., Komrusch, S. J., and Monperrus, M. Neural transfer learning for repairing security vulnerabilities in c code. *IEEE Transactions on Software Engineering*, 2022. URL <https://ieeexplore.ieee.org/document/9699412>.
- Devlin, J., Chang, M., Lee, K., and Toutanova, K. BERT: pre-training of deep bidirectional transformers for language understanding. In *NAACL*, 2019. URL <https://doi.org/10.18653/v1/n19-1423>.
- Dinella, E., Dai, H., Li, Z., Naik, M., Song, L., and Wang, K. Hoppity: Learning graph transformations to detect and fix bugs in programs. In *ICLR*, 2020. URL <https://openreview.net/forum?id=SJeqs6EFvB>.
- Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, B., Liu, T., Jiang, D., and Zhou, M. Codebert: A pre-trained model for programming and natural languages. In *Findings of EMNLP*,

2020. URL <https://doi.org/10.18653/v1/2020.findings-emnlp.139>.
- Guo, M., Haque, A., Huang, D., Yeung, S., and Fei-Fei, L. Dynamic task prioritization for multitask learning. In *ECCV*, 2018. URL https://doi.org/10.1007/978-3-030-01270-0_17.
- He, J., Lee, C., Raychev, V., and Vechev, M. Learning to find naming issues with big code and small supervision. In *PLDI*, 2021. URL <https://doi.org/10.1145/3453483.3454045>.
- Hellendoorn, V. J., Sutton, C., Singh, R., Maniatis, P., and Bieber, D. Global relational models of source code. In *ICLR*, 2020. URL <https://openreview.net/forum?id=B1lnbRNtwr>.
- Jain, P., Jain, A., Zhang, T., Abbeel, P., Gonzalez, J., and Stoica, I. Contrastive code representation learning. In *EMNLP*, 2021. URL <https://doi.org/10.18653/v1/2021.emnlp-main.482>.
- Jing, L. and Tian, Y. Self-supervised visual feature learning with deep neural networks: A survey. *TPAMI*, 43(11):4037–4058, 2021. URL <https://doi.org/10.1109/TPAMI.2020.2992393>.
- Kanade, A., Maniatis, P., Balakrishnan, G., and Shi, K. Learning and evaluating contextual embedding of source code. In *ICML*, 2020. URL <http://proceedings.mlr.press/v119/kanade20a.html>.
- Karampatsis, R. and Sutton, C. How often do single-statement bugs occur?: The manysstubs4j dataset. In *MSR*, 2020. URL <https://doi.org/10.1145/3379597.3387491>.
- Koh, P. W., Sagawa, S., Marklund, H., Xie, S. M., Zhang, M., Balsubramani, A., Hu, W., Yasunaga, M., Phillips, R. L., Gao, I., Lee, T., David, E., Stavness, I., Guo, W., Earnshaw, B., Haque, I., Beery, S. M., Leskovec, J., Kundaje, A., Pierson, E., Levine, S., Finn, C., and Liang, P. WILDS: A benchmark of in-the-wild distribution shifts. In *ICML*, 2021. URL <http://proceedings.mlr.press/v139/koh21a.html>.
- Li, X., Sun, X., Meng, Y., Liang, J., Wu, F., and Li, J. Dice loss for data-imbalanced NLP tasks. In *ACL*, 2020. URL <https://doi.org/10.18653/v1/2020.acl-main.45>.
- Li, Z., Zou, D., Xu, S., Ou, X., Jin, H., Wang, S., Deng, Z., and Zhong, Y. Vuldeepecker: A deep learning-based system for vulnerability detection. In *NDSS*, 2018. URL http://wp.internet-society.org/ndss/wp-content/uploads/sites/25/2018/02/ndss2018_03A-2_Li_paper.pdf.
- Lin, T., Goyal, P., Girshick, R. B., He, K., and Dollár, P. Focal loss for dense object detection. In *ICCV*, 2017. URL <https://doi.org/10.1109/ICCV.2017.324>.
- Mukherjee, R., Wen, Y., Chaudhari, D., Reps, T. W., Chaudhuri, S., and Jermaine, C. Neural program generation modulo static analysis. 2021. URL <https://arxiv.org/abs/2111.01633>.
- Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Köpf, A., Yang, E. Z., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., and Chintala, S. Pytorch: An imperative style, high-performance deep learning library. In *NeurIPS 2019*, 2019. URL <https://arxiv.org/abs/1912.01703>.
- Patra, J. and Pradel, M. Semantic bug seeding: a learning-based approach for creating realistic bugs. In *ES-EC/FSE*, 2021. URL <https://doi.org/10.1145/3468264.3468623>.
- Pradel, M. and Sen, K. Deepbugs: a learning approach to name-based bug detection. *Proc. ACM Program. Lang.*, 2(OOPSLA):147:1–147:25, 2018. URL <https://doi.org/10.1145/3276517>.
- Raychev, V., Bielik, P., and Vechev, M. Probabilistic model for code with decision trees. In *OOPSLA*, 2016. URL <https://doi.org/10.1145/2983990.2984041>.
- Rice, A., Aftandilian, E., Jaspan, C., Johnston, E., Pradel, M., and Arroyo-Paredes, Y. Detecting argument selection defects. *Proc. ACM Program. Lang.*, 1(OOPSLA):104:1–104:22, 2017. URL <https://doi.org/10.1145/3133928>.
- Saito, T. and Rehmsmeier, M. The precision-recall plot is more informative than the roc plot when evaluating binary classifiers on imbalanced datasets. *PLoS one*, 10(3):e0118432, 2015. URL <https://www.ncbi.nlm.nih.gov/pmc/articles/PMC4349800/>.
- Søgaard, A. and Goldberg, Y. Deep multi-task learning with low level tasks supervised at lower layers. In *ACL*, 2016. URL <https://doi.org/10.18653/v1/p16-2038>.
- Vasic, M., Kanade, A., Maniatis, P., Bieber, D., and Singh, R. Neural program repair by jointly learning to localize and repair. In *ICLR*, 2019. URL <https://openreview.net/forum?id=ByloJ20qtm>.
- Wei, J., Goyal, M., Durrett, G., and Dillig, I. Lambdanet: Probabilistic type inference using graph neural networks. In *ICLR*, 2020. URL <https://openreview.net/forum?id=Hkx6hANTwh>.

- Wolf, T., Debut, L., Sanh, V., Chaumond, J., Delangue, C., Moi, A., Cistac, P., Rault, T., Louf, R., Funtowicz, M., and Brew, J. Huggingface’s transformers: State-of-the-art natural language processing. *CoRR*, abs/1910.03771, 2019. URL <http://arxiv.org/abs/1910.03771>.
- Yasunaga, M. and Liang, P. Break-it-fix-it: Unsupervised learning for program repair. In *ICML, 2021*. URL <http://proceedings.mlr.press/v139/yasunaga21a.html>.
- Yin, P., Neubig, G., Allamanis, M., Brockschmidt, M., and Gaunt, A. L. Learning to represent edits. In *ICLR, 2019*. URL <https://openreview.net/forum?id=BJ16AjC5F7>.
- Zhang, Y. and Yang, Q. A survey on multi-task learning. *IEEE Transactions on Knowledge and Data Engineering*, 2021. URL <https://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=9392366>.
- Zhou, Y., Liu, S., Siow, J. K., Du, X., and Liu, Y. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. In *NeurIPS, 2019*. URL <https://proceedings.neurips.cc/paper/2019/hash/49265d2447bc3bbfe9e76306ce40a31f-Abstract.html>.
- Zügner, D., Kirschstein, T., Catasta, M., Leskovec, J., and Günnemann, S. Language-agnostic representation learning of source code from structure and context. In *ICLR, 2021*. URL <https://openreview.net/forum?id=Xh5eMZVONGF>.

A. Computing Localization and Repair Probabilities

Now we discuss how pointer models compute the localization probabilities $P^{loc} = [p_1^{loc}, \dots, p_n^{loc}]$ and the repair probabilities $P^{rep} = [p_1^{rep}, \dots, p_l^{rep}]$. Given the feature embeddings $[h_1, \dots, h_n]$ for program tokens $T = \langle t_1, t_2, \dots, t_n \rangle$, pointer models first compute a score vector $S^{loc} = [s_1^{loc}, \dots, s_n^{loc}]$ where each score s_i^{loc} reflects the likelihood of token t_i to be the bug location. If $t_i \in Loc$, i.e., token t_i is a candidate bug location, a feedforward network $\pi^{loc} : \mathbb{R}^m \rightarrow \mathbb{R}$ is applied on the feature vector h_i to compute s_i^{loc} . Otherwise, it is unlikely that t_i is the bug location so a minus infinity score is assigned. Formally, s_i^{loc} is computed as follows:

$$s_i^{loc} = \begin{cases} \pi^{loc}(h_i) & \text{if } M^{loc}[i] = 1, \\ -\text{inf} & \text{otherwise,} \end{cases}$$

where M^{loc} is the localization candidate mask:

$$M^{loc}[i] = \begin{cases} 1 & \text{if } t_i \in Loc, \\ 0 & \text{otherwise.} \end{cases}$$

S^{loc} is then normalized to localization probabilities with the softmax function: $P^{loc} = [p_1^{loc}, \dots, p_n^{loc}] = \text{softmax}(S^{loc})$.

Depending on the bug type, the set of repair tokens, Rep , can be drawn from T (e.g., for `var-misuse` and `arg-swap`) or fixed (e.g., for `wrong-binop`). For the former case, P^{rep} is computed in the same way as computing P^{loc} , except that another feedforward network, π^{rep} , and the repair candidate mask, M^{rep} , are used instead of π^{loc} and M^{loc} . When Rep is a fixed set of l tokens, the repair prediction is basically an l -class classification problem. We treat the first token t_1 of T as the repair token $t_{[rep]}$ and apply $\pi^{rep} : \mathbb{R}^m \rightarrow \mathbb{R}^l$ over its feature vector $h_{[rep]}$ to compute scores $S = \pi^{rep}(h_{[rep]})$. Then, the final repair score is set to $S[i]$ if M^{rep} indicates that the i -th repair token is valid, or to minus infinity otherwise. Overall, the repair scores $S^{rep} = [s_1^{rep}, \dots, s_l^{rep}]$ are computed as follows:

$$s_i^{rep} = \begin{cases} S[i] & \text{if } M^{rep}[i] = 1, \\ -\text{inf} & \text{otherwise,} \end{cases}$$

S^{rep} is then normalized to repair probabilities with the softmax function: $P^{rep} = [p_1^{rep}, \dots, p_l^{rep}] = \text{softmax}(S^{rep})$.

B. Implementation, Model and Training Details

In this section, we provide details in implementation, models, and training.

Constructing the Test Sets in Figure 1 In Figure 1, we mention three test sets used to reveal distribution shift in existing learning-based `var-misuse` detectors. Test set I is a balanced dataset with synthetic bugs, created by randomly selecting 336 non-buggy samples from `real-test` and injecting one synthetic bug into each non-buggy sample. Test set II is a balanced dataset with real bugs, created by replacing the synthetic bugs in the first test set by the 336 real bugs in `real-test`. Test set III is `real-test`.

Three Bug Types Handled by Our Work The definition of `var-misuse` (resp., `wrong-binop` and `arg-swap`) can be found in (Allamanis et al., 2018; Vasic et al., 2019) (resp., in (Pradel & Sen, 2018)). To determine Loc and Rep , we mainly follow (Allamanis et al., 2021; Kanade et al., 2020) and add small adjustments to capture more real bugs:

- `var-misuse`: we include all appearances of all local variables in Loc , as long as the appearance is not in a function definition and the variable has been defined before the appearance. When constructing Rep for each bug location variable, we include all local variable definitions that can be found in the scope of the bug location variable, except for the ones that define the bug location variable itself.
- `wrong-binop`: we deal with three sets of binary operators: arithmetics $\{+, *, -, /, \%\}$, comparisons $\{=, !=, is, is\ not, <, <=, >, >=\}$, and booleans $\{\text{and}, \text{or}\}$. If a binary operator belongs to any of the three sets, it is added to Loc . The set that the operator belongs to, excluding the operator itself, is treated as Rep . The repair candidate mask M^{rep} is of size 17, i.e., it includes all the operators in the three sets. M^{rep} sets the operators in Rep to 1 and the other operators to 0.

Table 9: The number of epochs, learning rate (LR), and time cost for the two training phases.

Bug Type	First phase			Second phase		
	Epochs	LR	Time	Epochs	LR	Time
<code>var-misuse</code>	1	10^{-6}	15h	2	10^{-6}	10h
<code>wrong-binop</code>	1	10^{-5}	15h	2	10^{-6}	6h
<code>arg-swap</code>	1	10^{-5}	15h	1	10^{-6}	2h

- `arg-swap`: We handle most function arguments but exclude keyworded and variable-length arguments that are less likely to be mistaken. In contrast to the other bug types, we also support the swapping of arguments that consist of more than a single token (e.g., an expression), by simply marking the first token as the bug location or the repair token. Moreover, we consider only functions that have two or more handled arguments. We put all candidate arguments in *Loc*. For each argument in *Loc*, *Rep* is the other candidate arguments used in the same function.

For bug injection and real bug extraction, we apply the bug-inducing rewriting rules in (Allamanis et al., 2021) given the definitions of *Loc* and *Rep* above.

Implementation with CuBERT Here, we describe the implementation of our techniques with CuBERT. CuBERT tokenizes the input program into a sequence of sub-tokens. When constructing the masks M^{loc} , C^{loc} , M^{rep} , and C^{rep} from *Loc* and *Rep*, we set the first sub-token of each token to 1. As standard with BERT-like models (Devlin et al., 2019), the first sub-token of the input sequence to CuBERT is always [CLS] used as aggregate sequence representation for classification tasks. We also use this token and its corresponding feature embedding for bug classification (all three tasks) and repair (only `wrong-binop`). CuBERT consists of a sequence of BERT layers and thus naturally aligns with our task hierarchy. Our two-phase training is technically a two-phase fine-tuning procedure when applied to pre-trained models like CuBERT. CuBERT requires the input sequence to be of fixed length, meaning that shorted sequences will be padded and longer sequences will be truncated. We chose length 512 due to constraints on hardware: CuBERT is demanding in terms of GPU memory and longer lengths caused out-of-memory errors on our machines. When extracting real bugs and injecting bugs into open source programs, we only consider bugs for which the bug location and at least one correct repair token are within the fixed length. This includes most real bugs we found.

Model Details CuBERT is a BERT-Large model with 24 hidden layers, 16 attention heads, 1024 hidden units, and in total 340M parameters. Our classification head π^{cls} is a two-layer feedforward network. The localization head π^{loc} is just a linear layer. The repair head π^{rep} is a linear layer for `var-misuse` and `arg-swap` and a two-layer feedforward network for `wrong-binop`. The size of the hidden layers is 1024 for all task heads. The implementation of our model is based on Hugging Face (Wolf et al., 2019) and PyTorch (Paszke et al., 2019).

Training Details Our experiments were done on servers with NVIDIA RTX 2080 Ti and NVIDIA TITAN X GPUs. As described in Section 3.3, our training procedure consists of two phases. In the first phase, we load a pretrained CuBERT model provided by the authors (Kanade et al., 2020) and fine-tune it with `syn-train`. In the second phase, we load the model trained from the first phase and perform fresh fine-tuning with `real-train`. The number of epochs, learning rate, and the time cost of the two training phases are shown in Table 9. Both training phases require at most two epochs to achieve good performance, highlighting the power of pretrained models to quickly adapt to new tasks and data distributions. In each batch, we feed two samples into the model as larger batch size will cause out-of-memory errors.

For fair comparison, when creating synthetic bugs with BugLab, we do not perform their data augmentation rewrite rules for all models. Those rules apply to all models and can be equally beneficial. When training GNN models with `syn-train` and `real-train`, we follow (Allamanis et al., 2021) to use early stopping over `real-val`. When training with BugLab, we use 80 meta-epochs, 5k samples (buggy/non-buggy ratio 1:1) per meta-epoch, and 40 model training epochs within one meta-epoch. This amounts to a total of around 6 days of training time for GNN.

C. More Evaluation Results

In this section, we present additional evaluation results.

Evaluation Results for `arg-swap` We repeat the experiments in Sections 5.1 and 5.2 for `arg-swap`. The results are shown in Tables 10 to 12 and Figures 9 to 12. Most observations that we can make from those results are similar to what we discussed in Sections 5.1 and 5.2 for `var-misuse` and `wrong-binop`. We highlight two difference: first, Our Full Method does not have a clear advantage over Only Synthetic and Mix in terms of AP (see Figure 9); second, the data imbalance and the amount of training data do not clearly improve the AP (see Figures 10 and 11). These different points are likely due to the distinct characteristics of `arg-swap` bugs. We leave it as an interesting future work item to further improve the performance of `arg-swap` detectors.

Parameter Selection for Task Hierarchy and Contrastive Learning In Tables 15 to 17 (resp., Tables 18 to 20), we show the model performance by only changing weight β of the contrastive loss (resp., the task order in our task hierarchy). For `var-misuse` and `wrong-binop`, Our Full Method (highlighted with \star) performs the best among all the configurations. In terms of β and `var-misuse`, Our Full Method ($\beta = 0.5$) is less precise but has significantly higher recall than $\beta = 8$. For `arg-swap`, Our Full Method performs the best on the validation set but not on the test set.

Table 10: Changing training phases (arg-swap).

Method	cls		cls-loc		cls-loc-rep	
	P	R	P	R	P	R
Only Synthetic	1.31	39.84	1.00	30.49	0.79	23.98
Only Real	0	0	0	0	0	0
Mix	2.02	32.93	1.69	27.64	1.59	26.02
Two Synthetic	44.19	7.72	44.19	7.72	44.19	7.72
Our Full Method	73.68	5.69	73.68	5.69	73.68	5.69

Table 11: Applying our two-phase training on GNN and BugLab (Allamanis et al., 2021) (arg-swap).

Model	Training Phases	cls		cls-loc		cls-loc-rep	
		P	R	P	R	P	R
GNN	Only Synthetic	0.99	50.00	0.68	34.15	0.43	21.95
GNN	Synthetic + Real	83.33	4.07	83.33	4.07	75.00	3.66
GNN	Only BugLab	0.81	51.63	0.50	32.11	0.37	23.58
GNN	BugLab + Real	81.82	3.66	81.82	3.66	72.73	3.25
Our Model	Synthetic + Real	73.68	5.69	73.68	5.69	73.68	5.69

Table 12: Evaluating other techniques (arg-swap).

Method	cls		cls-loc		cls-loc-rep	
	P	R	P	R	P	R
No cls Head	34.21	5.28	34.21	5.28	34.21	5.28
No Hierarchy	61.29	7.72	61.29	7.72	61.29	7.72
No Focal Loss	73.68	5.69	73.68	5.69	73.68	5.69
No Contrastive	46.15	7.32	46.15	7.32	46.15	7.32
Our Full Method	73.68	5.69	73.68	5.69	73.68	5.69

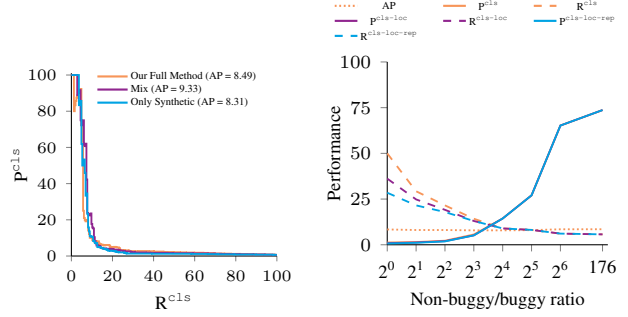
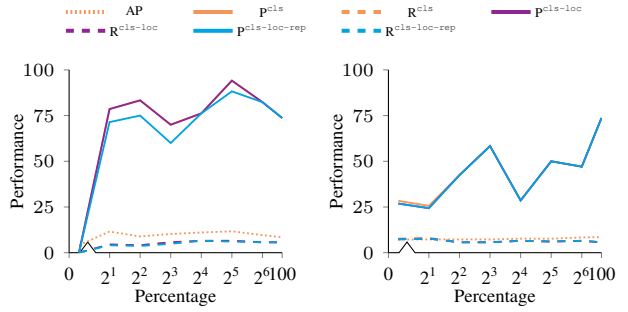


Figure 9: Precision-recall curve and AP for methods in Table 10 (arg-swap).

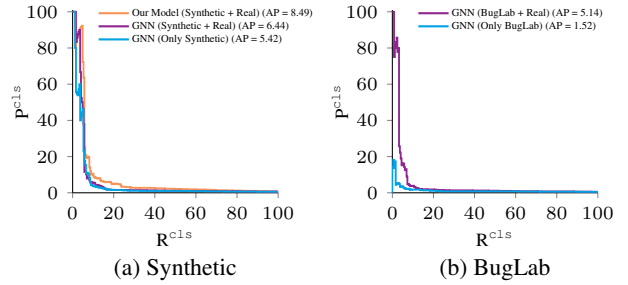
Figure 10: Varying data skewness in the second training phase (arg-swap).



(a) synthetic-train

(b) real-train

Figure 11: Model performance with subsampled syn-train or real-train (arg-swap).



(a) Synthetic

(b) BugLab

Figure 12: Precision-recall curves and AP for GNN and Our Model with different training phases (arg-swap).

Table 15: Different weight β (var-misuse).

Weight β of Contrastive Loss	cls		cls-loc		cls-loc-rep	
	P	R	P	R	P	R
0	60.00	14.29	57.50	13.69	53.75	12.80
0.25	59.21	13.39	56.58	12.80	55.26	12.50
* 0.5	64.79	13.69	61.97	13.10	56.34	11.90
1	61.90	11.61	57.14	10.71	55.56	10.42
2	61.67	11.01	58.33	10.42	58.33	10.42
4	58.62	10.12	55.17	9.52	51.72	8.93
8	71.43	1.49	71.43	1.49	71.43	1.49
16	63.64	6.25	63.64	6.25	60.61	5.95

 Table 16: Different weight β (wrong-binop).

Weight β of Contrastive Loss	cls		cls-loc		cls-loc-rep	
	P	R	P	R	P	R
0	47.96	43.18	47.06	42.36	46.15	41.55
0.25	47.92	44.60	47.05	43.79	46.17	42.97
0.5	47.62	44.81	46.75	43.99	46.10	43.38
1	48.51	46.44	47.66	45.62	46.60	44.60
2	51.54	44.20	50.36	43.18	49.64	42.57
* 4	52.30	43.99	51.09	42.97	49.64	41.75
8	48.35	41.75	47.41	40.94	46.70	40.33
16	50.23	43.79	49.30	42.97	48.36	42.16

 Table 17: Different weight β (arg-swap).

Weight β of Contrastive Loss	cls		cls-loc		cls-loc-rep	
	P	R	P	R	P	R
0	46.15	7.32	46.15	7.32	46.15	7.32
0.25	60.00	4.88	60.00	4.88	60.00	4.88
* 0.5	73.68	5.69	73.68	5.69	73.68	5.69
1	69.57	6.50	69.57	6.50	69.57	6.50
2	63.64	5.69	63.64	5.69	54.55	4.88
4	72.73	6.50	72.73	6.50	72.73	6.50
8	76.47	5.28	76.47	5.28	76.47	5.28
16	86.67	5.28	86.67	5.28	86.67	5.28

Table 18: Different task order (var-misuse).

Task Order	cls		cls-loc		cls-loc-rep	
	P	R	P	R	P	R
No Hierarchy	51.82	16.96	50.91	16.67	48.18	15.77
* <i>cls, loc, rep</i>	64.79	13.69	61.97	13.10	56.34	11.90
<i>cls, rep, loc</i>	57.53	12.50	54.79	11.90	54.79	11.90
<i>loc, cls, rep</i>	53.66	13.10	50.00	12.20	47.56	11.61
<i>loc, rep, cls</i>	57.69	13.39	53.85	12.50	51.28	11.90
<i>rep, cls, loc</i>	51.69	13.69	49.44	13.10	47.19	12.50
<i>rep, loc, cls</i>	52.38	13.10	50.00	12.50	46.43	11.61

Table 19: Different task order (wrong-binop).

Task Order	cls		cls-loc		cls-loc-rep	
	P	R	P	R	P	R
No Hierarchy	48.70	45.62	47.83	44.81	46.52	43.58
<i>cls, loc, rep</i>	49.43	44.40	48.30	43.38	47.39	42.57
<i>cls, rep, loc</i>	48.29	43.18	47.38	42.36	46.01	41.14
<i>loc, cls, rep</i>	49.66	44.20	48.74	43.38	47.60	42.36
<i>loc, rep, cls</i>	46.44	46.44	45.42	45.42	44.81	44.81
<i>rep, cls, loc</i>	50.82	44.20	49.88	43.38	48.71	42.36
* <i>rep, loc, cls</i>	52.30	43.99	51.09	42.97	49.64	41.75

Table 20: Different task order (arg-swap).

Task Order	cls		cls-loc		cls-loc-rep	
	P	R	P	R	P	R
No Hierarchy	61.29	7.72	61.29	7.72	61.29	7.72
<i>cls, loc, rep</i>	94.12	6.50	94.12	6.50	88.24	6.10
<i>cls, rep, loc</i>	53.57	6.10	53.57	6.10	53.57	6.10
* <i>loc, cls, rep</i>	73.68	5.69	73.68	5.69	73.68	5.69
<i>loc, rep, cls</i>	55.56	6.10	55.56	6.10	55.56	6.10
<i>rep, cls, loc</i>	76.47	5.28	76.47	5.28	76.47	5.28
<i>rep, loc, cls</i>	63.64	5.69	63.64	5.69	63.64	5.69

D. Case Studies of Inspected Warnings

In the following we present case studies of the warnings we inspect in Section 5.3. We showcase representative bugs and code quality issues raised by our models. Further, we also provide examples of false positives and discuss potential causes for the failure. We visualise the bug location with and the repair token with .

D.1. var-misuse: bug in repository aleju/imgaug

Our bug detector model correctly identifies the redundant check on `x_px` instead of `y_px`.

```
def translate(self, x_px, y_px):
    if x_px < 1e-4 or x_px > 1e-4 or y_px < 1e-4 or x_px > 1e-4:
        matrix = np.array([[1, 0, x_px], [0, 1, y_px], [0, 0, 1]], dtype=np.float32)
        self._mul(matrix)
    return self
```

D.2. var-misuse: bug in repository babelsberg/babelsberg-r

The model identifies that `w_read` was already checked but not `w_write`.

```
def test_pipe(self, space):
    w_res = space.execute("""
return IO.pipe
""")
    w_read, w_write = space.listdir(w_res)
    assert isinstance(w_read, W_IOObject)
    assert isinstance(w_write, W_IOObject)
    w_res = space.execute("""
r, w, r_c, w_c = IO.pipe do |r, w|
  r.close
  [r, w, r.closed?, w.closed?]
end
return r.closed?, w.closed?, r_c, w_c
""")
    assert self.unwrap(space, w_res) == [True, True, True, False]
```

D.3. var-misuse: code quality issue in repository JonnyWong16/plexpy

The model proposes to replace `c` by `snowman`, since `snowman` is otherwise unused. Even though this replacement does not suggest a bug, the warning remains useful as the unused variable `snowman` must be considered a code quality issue.

```
def test_ensure_ascii_still_works(self):
    # in the ascii range, ensure that everything is the same
    for c in map(unichr, range(0, 127)):
        self.assertEqual(
            json.dumps(c, ensure_ascii=False),
            json.dumps(c))
    snowman = u'\N{SNOWMAN}'
    self.assertEqual(
        json.dumps(c, ensure_ascii=False),
        '"' + c + '"')
```

D.4. var-misuse: false positive in repository ceache/treadmill

The model proposes to replace `fmt` with `server`, although the surrounding code clearly implies that `server` is `None` at this point in the program. Therefore, we consider it as a false positive. In this case, the two preceding method calls with `_server` in their name, were given the `server` variable as second argument. This may have affected the prediction of the model, disregarding the surrounding conditional statement with respect to `server`.

```
def server_cmd(server, reason, fmt, clear):
    """Manage server blackout."""
    if server is not None:
        if clear:
            _clear_server_blackout(context.GLOBAL.zk.conn, server)
        else:
            _blackout_server(context.GLOBAL.zk.conn, server, reason)
    else:
        _list_server_blackouts(context.GLOBAL.zk.conn, fmt)
```

D.5. wrong-binop: bug in repository Amechi101/concepteur-market-app

The model detects the presence of the string formatting literal %s in the string and in consequence raises a warning about a wrong binary operator.

```
+ * - / %
def buildTransform(inputProfile, outputProfile, inMode, outMode,
    renderingIntent=INTENT_PERCEPTUAL, flags=0):

    if not isinstance(renderingIntent, int) or not (0 <= renderingIntent <=3):
        raise PyCMSError("renderingIntent must be an integer between 0 and 3")

    if not isinstance(flags, int) or not (0 <= flags <= _MAX_FLAG):
        raise PyCMSError("flags must be an integer between 0 and %s" + _MAX_FLAG)

    try:
        if not isinstance(inputProfile, ImageCmsProfile):
            inputProfile = ImageCmsProfile(inputProfile)
        if not isinstance(outputProfile, ImageCmsProfile):
            outputProfile = ImageCmsProfile(outputProfile)
        return ImageCmsTransform(inputProfile, outputProfile, inMode, outMode,
            renderingIntent, flags=flags)
    except (IOError, TypeError, ValueError) as v:
        raise PyCMSError(v)
```

D.6. wrong-binop: bug in repository maestro-hybrid-cloud/heat

The model correctly raises a warning since the comparison must be a containment check instead of an equality check.

```
== != is is not < <= > >= in not in
def suspend(self):
    # No need to suspend if the stack has been suspended
    if self.state == (self.SUSPEND, self.COMPLETE):
        LOG.info(_LI('%s is already suspended'), six.text_type(self))
        return

    self.updated_time = datetime.datetime.utcnow()
    sus_task = scheduler.TaskRunner(
        self.stack_task,
        action=self.SUSPEND,
        reverse=True,
        error_wait_time=cfg.CONF.error_wait_time)
    sus_task(timeout=self.timeout_secs())
```

D.7. wrong-binop: code quality issue in repository tomspur/shedskin

The model identifies the unconventional use of the != operator when comparing with None.

```
== != is is not < <= > >= in not in
def getFromEnviron():
    if HttpProxy.instance is not None:
        return HttpProxy.instance
    url = None
    for key in ('http_proxy', 'https_proxy'):
        url = os.environ.get(key)
        if url: break
    if not url:
        return None
    dat = urlparse(url)
    port = 80 if dat.scheme == 'http' else 443
    if dat.port != None: port = int(dat.port)
    host = dat.hostname
    return HttpProxy((host, port), dat.username, dat.password)
```

D.8. wrong-binop: false positive in repository wechatpy/wechatpy

Our model mistakenly raises a `wrong-binop` warning with the `==` operator and proposes to replace it with `>` operator. In this case, the log message below the conditional check may have triggered the warning.

```
== != is is not < <= > >= in not in
def add_article(self, article):
    if len(self.articles) == 10:
        raise AttributeError("Can't add more than 10 articles in an ArticlesReply")
    articles = self.articles
    articles.append(article)
    self.articles = articles
```

D.9. arg-swap: bug in repository sgiavasis/nipype

Our model identifies the invalid use of the NumPy function `np.savetxt(streamlines, out_file + '.txt')`², which expects first the file name, i.e., `out_file + '.txt'` in this case.

```
def _trk_to_coords(self, in_file, out_file=None):
    from nibabel.trackvis import TrackvisFile
    trkfile = TrackvisFile.from_file(in_file)
    streamlines = trkfile.streamlines

    if out_file is None:
        out_file, _ = op.splitext(in_file)

    np.savetxt(streamlines, out_file + '.txt')
    return out_file + '.txt'
```

D.10. arg-swap: false positive in repository davehunt/bedrock

Our model mistakenly raises an argument swap warning with the `SpacesPage` constructor. In fact, with this specific repository our model repeatedly raised issues at similar code locations where the Selenium library is used. This is likely due to not having encountered similar code during training and hence due to lack of repository-specific information.

²NumPy Documentation, <https://numpy.org/doc/stable/reference/generated/numpy.savetxt.html>


```
@pytest.mark.nondestructive
def test_spaces_list(base_url, selenium):
    page = SpacesPage(base_url, selenium).open()
    assert page.displayed_map_pins == len(page.spaces)
    for space in page.spaces:
        space.click()
        assert space.is_selected
        assert space.is_displayed
        assert 1 == page.displayed_map_pins
```

E. Bug Reports to the Developers

We report a number of bugs found during our manual inspection as pull requests to the developers. For forked repositories, we trace the buggy code in the original repository. If the original repository has the same code, we create a bug report in the original repository. Otherwise, we do not report the bug. We also found that 7 bugs are already fixed in the latest version of the repository. The links to the pull requests are listed below. We also mark the pull requests for which we received a confirmation from the developers before the deadline for the final version of this paper (two days after we reported them).

var-misuse:

```
(merged) https://github.com/numpy/numpy/pull/21764
(merged) https://github.com/frappe/erpnext/pull/31372
(merged) https://github.com/spirali/kaira/pull/31
(merged) https://github.com/pyro-ppl/pyro/pull/3107
(merged) https://github.com/nest/nestml/pull/789
(merged) https://github.com/cupy/cupy/pull/6786
(merged) https://github.com/funkring/fdoo/pull/14
(confirmed) https://github.com/apache/airflow/pull/24472
https://github.com/topazproject/topaz/pull/875
https://github.com/inspirehep/inspire-next/pull/4188
https://github.com/CloCkWeRX/rabbitvcs-svn-mirror/pull/6
https://github.com/amonapp/amon/pull/219
https://github.com/mjirik/io3d/pull/9
https://github.com/jhogsett/linkit/pull/30
https://github.com/aleju/imgaug/pull/821
https://github.com/python-diamond/Diamond/pull/765
https://github.com/python/cpython/pull/93935
https://github.com/orangeduck/PyAutoC/pull/3
https://github.com/damonkohler/sl4a/pull/332
https://github.com/vyrus/wubi/pull/1
https://github.com/shon/httpagentparser/pull/89
https://github.com/midgetspy/Sick-Beard/pull/991
https://github.com/sgala/gajim/pull/3
https://github.com/tensorflow/tensorflow/pull/56468
```

wrong-binop:

```
(merged) https://github.com/python-pillow/Pillow/pull/6370
(merged) https://github.com/funkring/fdoo/pull/15
(false positive) https://github.com/kovidgoyal/calibre/pull/1658
https://github.com/kbase/assembly/pull/327
https://github.com/maestro-hybrid-cloud/heat/pull/1
https://github.com/gramps-project/gramps/pull/1380
https://github.com/scikit-learn/scikit-learn/pull/23635
https://github.com/pupeng/hone/pull/1
https://github.com/edisonlz/fruit/pull/1
https://github.com/certsocietegenerale/FIR/pull/275
```

On Distribution Shift in Learning-based Bug Detectors

<https://github.com/MediaBrowser/MediaBrowser.Kodi/pull/117>
<https://github.com/sgala/gajim/pull/4>
<https://github.com/mapsme/omim/pull/14185>
<https://github.com/tensorflow/tensorflow/pull/56471>
<https://github.com/catatult-project/catatult-csm/pull/2>

arg-swap:

(merged) <https://github.com/clinton-hall/nzbToMedia/pull/1889>
(merged) <https://github.com/IronLanguages/ironpython3/pull/1495>
(false positive) <https://github.com/python/cpython/pull/93869>
<https://github.com/google/digitalbuildings/pull/646>
<https://github.com/quodlibet/mutagen/pull/563>
<https://github.com/nipy/nipype/pull/3485>