How to Train Your Neural Bug Detector: Artificial vs Real Bugs

Cedric Richter
University of Oldenburg
Oldenburg, Germany
cedric.richter@uol.de

Heike Wehrheim
University of Oldenburg
Oldenburg, Germany
heike.wehrheim@uol.de

***Abstract*—Real bug fixes found in open source repositories seem to be the perfect source for *learning* to localize and repair *real bugs*. Yet, the scale of existing bug fix collections is typically too small for training data-intensive neural approaches. Neural bug detectors are hence almost exclusively trained on *artificial bugs*, produced by mutating existing source code and thus easily obtainable at large scales. However, neural bug detectors trained on artificial bugs usually underperform when faced with real bugs.**

To address this shortcoming, we set out to explore the impact of training on real bug fixes at scale. Our systematic study compares neural bug detectors trained on real bug fixes, artificial bugs and mixtures of real and artificial bugs at various dataset scales and with varying training techniques. Based on our insights gained from training on a novel dataset of 33k real bug fixes, we were able to identify a training setting capable of significantly improving the performance of existing neural bug detectors by up to 170% on simple bugs in Python. In addition, our evaluation shows that further gains can be expected by increasing the size of the real bug fix dataset or the code dataset used for generating artificial bugs. To facilitate future research on neural bug detection, we release our real bug fix dataset, trained models and code.

***Index Terms*—program repair, bug detection, bug fixes, learning to debug**

I. Introduction

Finding and fixing software bugs are one of the most frequent tasks for software engineers [1]. Consequently, numerous tools have been developed to support software engineers in bug localization and repair [2]–[4], [6], [7], [11], [13], [22]–[24], [26], [29], [34]–[36], [43], [44]. Still, bugs find their way into open software repositories and then require bug fixing code changes. This raises the question whether we can make use of exactly these open source bugs, *learn* from common developer mistakes and thereby provide even more support for debugging.

Previous work [2], [3], [13], [26], [29], [36] addressed this question by designing automatic learning-based methods for bug localization and repair. These so called *neural bug detectors* are trained on millionth of examples of buggy and correct code to simultaneously detect, localize and repair bugs. Existing neural bug detectors already support a wide range of hard-to-find bugs such as variable misuses [2] and binary operator bugs [26]. These bug types are detected and repaired based on static bug patterns learned from source code only, without any execution. A key problem of neural bug detectors can be found in the training process. To obtain the necessary amount of data needed for training, neural bug detectors are typically trained on *code mutants* (i.e., *artificial bugs*) instead of real bug fixes. Mutants are generated by automatically injecting small code changes into existing code. As this process is fully automated, mutants can be easily obtained at the large scales required for training effective learning-based models. However, mutants do not necessarily represent *real* bugs which could ultimately limit the performance of learning based bug localization and repair. Indeed, studies have shown that neural bug detectors trained purely on mutants usually underperform when evaluated on real bugs [3], [13], [36].

In contrast, in this work we aim at exploring the effect of real bug fixes – obtained from mining open source repositories – on the training of learning-based bug localization and repair methods. Previous work [12] showed that by training neural bug detectors in a more realistic scenario with a strong bias towards non-buggy code can significantly improve the *precision* (i.e. the ratio of false alarms and bugs detected) of existing bug detectors. However, existing neural bug detectors are often severely limited in their ability to identify and repair real bugs. We hypothesize that – because of the scale of the employed datasets – the bug patterns needed for identifying real bugs lack *support* in the training sets. Therefore, it is unclear (1) how real bug fixes would impact the training process at scale (with sufficient support), (2) whether the size of the datasets (both mutants and real bug fixes) employed in previous studies is a limiting factor and (3) whether neural bug detectors still benefit from mutants when trained on a large set of real bug fixes.

To answer these questions, we performed a systematic study comparing neural bug detectors trained on real bug fixes, mutants, and mixtures of mutants and real bug fixes at various dataset scales. To evaluate the impact of real bug fixes at scale, we mined a novel dataset of 33k real-world bug fixes (more than 40 times larger than in previous studies with neural bug detectors [12]) crawled from public repositories. During our evaluation, we consider several models trained on various subsets of the bug fix dataset ranging from a few hundred examples to the full dataset size. To evaluate impact of mutants at different scales, we vary (1) the number of mutants generated per code snippet and (2) the size of the code dataset used for mutation. For evaluating the joint impact of training on real bug fixes and training on mutants, we employ a joint framework for training neural bug detectors. Similar to previous works [12], we train the neural bug detectors in two steps by first *pre-training* on a high number of mutants and then *fine-tuning* on real bug fixes. This design allows us to not only evaluate the impact of real bug fixes and mutants on the training process together, but also individually (by skipping either the pre-training or the fine-tuning phase). We however opted for a balanced setup to evaluate the impact of the training data and not of the buggy to correct code ratio. We emphasize that our goal is not a new method, but to provide insights on the effect of real bug fixes and mutants at scale on the training of neural bug detectors.

To measure the impact of the training process on the detection and repair of real bugs, we reimplemented three common baselines for fixing a variety of *single token bugs* in Python. Our implementation considers four common types of single token bugs – that can be fixed by changing a single program token like an operator, variable or literal. We evaluate the trained models on over 2000 real world bugs collected in the PyPIBugs benchmark [3].

We observe that integrating real bug fixes in the training process (especially at the provided scale) significantly improves the ability of neural bug detectors to identify and repair real bugs. In fact, the evaluated neural bug detectors can identify and repair between 20% to 50% more real bugs when fine-tuned on real bug fixes. In addition, we observe that (1) training on mutants is still crucial for obtaining high performing models and (2) both the scale of the mutant and real bug fix datasets can be a limiting factor. By increasing the size of the mutant dataset with our scaling methods, further gains can be achieved after fine-tuning on real bug fixes (up to 32% more real bugs found).

Contributions. Our main contributions can be summarized as follows:

- We perform a **systematic** study to evaluate the effect of training on real bug fixes and mutants on the performance of neural bug detectors at different dataset scales.
- We introduce a **novel** dataset of 33k real-world single token bug fixes in Python for training.
- We reimplemented several baselines for single token bugs in Python in a unified framework.
- We demonstrate that both the scale of the real bug fix dataset and the scale of the mutant dataset used during training have a **significant** impact on the capabilities of a neural bug detector to detect and repair real bugs. After training on high number of real bug fixes and mutants, the resulting models are the first neural bug detectors to repair a significant portion of a real bug benchmark by statically analyzing the code.

All data and software including all trained models, real-world datasets and the training code are publicly available on Zenodo [31] for inspection and reproduction. The trained neural bug detectors are also available on Github:

https://github.com/cedricrupb/nbfbaselines

II. Background

Next, we introduce the necessary background. To begin with, we start by describing the task of single token bug localization and repair and how neural bug detectors address this task by predicting token replacements.

A. Single token bug localization and repair

We focus on the localization and repair of *single token bugs*. Single token bugs are bugs that can be repaired by replacing only a single program token (e.g. a variable or binary operator). Single token bugs are frequent [2], [19] and often easy to repair (as they require only a single identifier change). However, because of their size they are often missed by developers [2]. Examples for single token bugs in Python are given in Table I.

Task description. Throughout this work, we view source code as a sequence of tokens $\mathcal{T} = t_0, t_1, t_2, \dots, t_n$. A single token bug can then be fixed by replacing a *single* token t_l with another token r in the same scope (i.e., $r = t_{l'}$ for some $l' \in \{1, \dots, n\}$) or coming from an external vocabulary V (i.e., $r \in V$). To effectively localize and repair a single token bug, the following three tasks have to be performed: (1) the program $\mathcal{T}$ has to be *classified* to contain a bug, (2) the bug location t_l has to be *localized* and (3) the correct *repair* r has to be identified. In practice, these three tasks are often modeled jointly as *token replacement* operations. Let $\mathcal{T}$ be a program containing a single token bug and $\mathcal{T}'$ be the corrected bug-free version, then the localization and repair model is trained to perform the following operations:

$$\mathcal{T} \xrightarrow{\text{replace}(t_l, r)} \mathcal{T}' \quad (1) \qquad\qquad \mathcal{T}' \xrightarrow{\text{noop}()} \mathcal{T}' \quad (2)$$

Here, we fix the buggy program $\mathcal{T}$ by replacing t_l with r and therefore translating it into $\mathcal{T}'$. Since $\mathcal{T}'$ is *bug-free*, a change is not required (*noop*). Selecting the replacement operation *noop* hence indicates that the code is considered correct and no bug fix is required.

In practice, we train models to estimate the likelihood of each *possible* token replacement and select the most likely replacement to fix $\mathcal{T}$.

B. Mutation

Motivated by the unavailability of real bug fixes at a sufficient scale, previous neural bug detectors [3], [13], [29], [36] mainly focused on training on *mutants*. Mutants are artificially generated (pseudo-)bugs that are introduced into a correct program via a mutation operator. For single token bugs, the mutation operator can be seen as a token replacement operator which can be inverted by a localization and repair model:

$$\mathcal{T} \xrightarrow{\text{mutate}(t_l, r)} \mathcal{T}' \xrightarrow{\text{replace}(t_l, r^{-1})} \mathcal{T} \quad (3)$$

For a dataset of bug-free programs (e.g. mined from open source projects), the mutation operator first introduces a token mutation by replacing a random token with a random other token. The token types are often specified (e.g. binary

TABLE I: Examples of single token bugs taken from PyPIBugs [3]

Example	Description

`1 # VarMisuse: applied instead of patch`
`2 applied = self.db.applied_patches()`
`3 for patch in applied:`
`4     if patch in patches:`
`5         patches.remove(applied)`
	All `applied` patches should be removed from the `patches` list. However, the developer mistakenly tries to remove `applied` instead of a single `patch`. **Fix:** replace `applied` in Line 5 by `patch` defined in Line 3.

`1 # BinOp: != instead of ==`
`2 def updateRefractionParameters(self):`
`3     ...`
`4     if self.checkRefracNoTrack.isChecked():`
`5         if self.app.mount.status != 0:`
`6             return False`
`7     ...`
	The function `updateRefractionParameters` performs an update and returns true if the update was successful. Prior to the update, the function checks if the `mount` is ready and aborts if not. We can conventionally expect that we abort if the status is zero. However, the function checks whether the status is not zero. **Fix:** replace != in Line 7 by ==.

`1 # Negation: namespace instead of not namespace`
`2 if namespace:`
`3     self.namespacesFilter = [ "prymatex", "user" ]`
`4 else:`
`5     self.namespacesFilter = namespace.split()`
	A default `namespacesFilter` should be used if no `namespace` is given. However, the condition checks the inverse. **Fix:** replace `namespace` in Line 2 by `not namespace`.

operators) such that programs remain interpretable after the transformation. Afterwards, the localization and repair model is trained to invert the mutation process to obtain the original program.

While traditionally mutation operator are designed as a random process, previous work also tried to design more realistic mutation operators by learning from real bug fixes [25], by training an adversary to the repair model [3] or by finding replacements that naturally fit the context [29].

C. Real bug fixes

Real bug fixes are often obtained by scraping the commit history of public open source projects. During this process, commits are typically classified as ”bug fixing” based on certain keywords in the commit message [19]. Even though this process cannot guarantee that every collected commit is a real bug fix, it has been empirically shown [19] that the process is highly precise (e.g. over 90% of all collected code changes were real bug fixes). In this work, we are interested in *single token bug fixes*. Here, a bug in version $\mathcal{T}_i$ is fixed by replacing only a single token:

$$\mathcal{T}_i \xrightarrow{\text{replace}(t_l, r)} \mathcal{T}_{i+1} \tag{4}$$

Note that a (bug fixing) commit only represents a snapshot of the project at time i. Therefore, while it is highly likely $\mathcal{T}_i$ contains a single token bug which can be fixed by replace(t_l, r), we cannot guarantee that the bug fix is complete and $\mathcal{T}_{i+1}$ is bug-free.

III. METHODOLOGY

Our intention is to explore the impact of learning from mutants (as to exploit the large scale of artificial bugs producible by mutation) and learning from real bugs (as to take advantage of their ”realness”). In the following, we describe the training technique used for training neural bug detectors which will allow us to isolate the effect of mutants and real bug fixes. Then, we explain the employed neural architectures and inference strategy in more detail.

A. Training with mutants and real bug fixes

We perform the training in two phases (as shown in Figure 1), a *pre-training* and a *fine-tuning* phase. In the first *pre-training phase*, we train on artificially generated mutants introduced into source code obtained by mining public open source repositories. Afterwards, in the second *fine-tuning* phase, we either continue the training on real bug fixes with our pre-trained models (to measure the joint impact of mutants and real bug fixes) or start the training from scratch.

Pre-training with code mutants. During pre-training, we train our model similar to the way current localization and repair models are trained [13]. Here, the training objective is not to identify real bugs but rather to identify and transform mutated code snippets back into the real code snippets. For this task, we naturally start with a general corpus of *Github code* snippets (e.g. function implementations). We randomly *mutate* each code snippet in our corpus at max k times which produces a dataset of at max k *unique mutants* per code snippet[1]. During our experiments, we vary the number of mutants per code snippet to measure the effects of the scale of mutations applied during pre-training. We employ the original code corpus as training examples of unmutated *correct* code. Based on the two datasets, the localization and repair models are then trained to (1) distinguish mutants from real code, (2) identify the mutated location (if any) and (3) find the original replaced token. Since the dataset of mutants is up to k times larger than the set of correct code snippets (by construction), we additionally *supersample* the correct code snippets to match their frequency with the mutants. In other words, for each mutant seen during training we sample a correct code snippet from our training

[1]The number of different mutants per code snippet is limited (e.g. by the number of binary operators found in the code snippet) and might be lower than k. We only consider unique mutants and never introduce mutation duplicates.

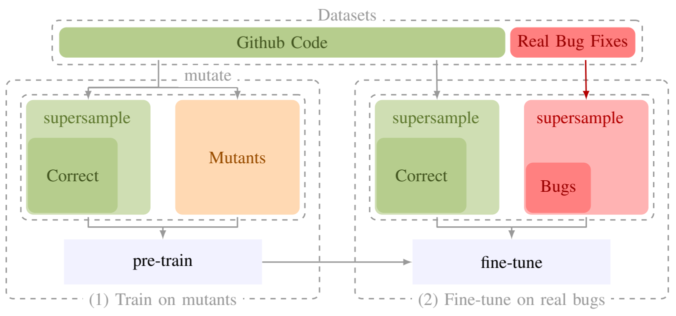


Fig. 1: Overview over the training process.

set. This avoids biasing the models toward the dominant class (e.g., correct code or mutants), which allows us to measure the effects of mutants more directly.

Fine-tuning with real bug fixes. In the second phase, we continue (or restart) the training on realistic buggy and bug-free code. As examples for realistic buggy code, we employ the code related to a real bug fix *before* the fix is applied. Bug-free code is again obtained by using the original Github corpus. During our experiments, we vary the number of real bugs seen during training ranging from a few hundred examples to a large scale dataset of multiple thousand real bug fixes. In this phase, the neural bug detectors are fine-tuned to (1) distinguish *real buggy code* from bug-free code, (2) identify the bug location (if any) and (3) imitate the original bug fix. Since now, the code corpus is usually much larger than the set of bug fixes, we supersample the buggy programs to match the correct programs in their frequency.

B. Model architecture

Next, we discuss the neural model architecture employed to learn localization and repair of single token bugs. Since our main focus is to study the effect of mutants and real bug fixes on the training process, we employ a common model architecture [13] for learning bug localization and repair.

Probabilistic model. For the task of single token localization and repair, programs are represented as a sequence of tokens $\mathcal{T} = t_0, t_1, t_2, \dots, t_n$ where each token t_l represents a potential bug location. Single token bugs are fixed by only replacing a single token t_l with another token r (replace(t_l, r)). In the following, we model localization and repair as a joint probability distribution over all potential bug locations and repairs $\{\langle l, r\rangle \mid t_l \in \mathcal{T} \cup \{\texttt{NOOP}\} \text{ and } r \in \mathcal{T} \cup V\}$:

$$p(\langle l, r\rangle \mid \mathcal{T}) = p_{\text{loc}}(l \mid \mathcal{T}) \cdot p_{\text{repair}}(r \mid l, \mathcal{T}) \qquad (5)$$

Here, localization and repair is factorized into first localizing a bug location ($p_{\text{loc}}(l \mid \mathcal{T})$) and then finding a repair dependent on the bug location ($p_{\text{repair}}(r \mid l, \mathcal{T})$). For localization, we include a special NOOP location that indicates that $\mathcal{T}$ is bug-free and no repair is necessary. In practice, we implement the probability distributions similar to pointer networks [10] (with the addition of an external vocabulary for repair).

Neural code representation. To learn a neural code representation, we learn a neural encoding function $\mathbf{e}(t_i)$ that maps each token t_i to a vector representation. For this, we employ a BPE subtoken encoder [32] (with a vocabulary of 10K subtokens) to obtain an initial token embedding by averaging the embedding of its subtokens. Afterwards, we encode the sequence of token embeddings via a neural encoder to obtain a contextualized vector representation $\mathbf{r}_i = \mathbf{e}(t_i)$. During our evaluation, we evaluate both *graph based* neural encoders [13] that learn from a graph representation of the program and *Transformer based* neural encoders [8] that directly operate on the token sequence.

Localization & Repair module. To finally compute the probability distribution over bug locations and repairs, we employ individual modules for localization and repair based on the computed vector representation $\mathbf{r}_i$. The *localization module* is a multilayer perceptron that computes a bugginess score for each potential bug location based on the vector representation $\mathbf{r}_i$ and the original token embedding. The objective of the localization module is to learn how likely the token t_i does not fit its surrounding context (represented by $\mathbf{r}_i$). The localization probability is computed as a softmax distribution over all potential bug locations. The *repair module* is designed similar to CopyNet [10]. Given the vector representation $\mathbf{r}_i$ of a potential bug location, the repair module computes a repairing score between the bug location and each repair candidate at token t_j (represented by $\mathbf{r}_j$). In addition, a similar score is obtained based on token embeddings of an external vocabulary V (e.g. other binary operators). The repair probability score is then computed as a softmax distribution over all repair candidates.

C. Finding and repairing real bugs

After the successful training process, the localization and repair models are confronted with new unseen programs with the objective of identifying a potential bug and repair it. This is typically done by finding the *most likely* repair for the *most likely* bug location (according to the model) [36]. However, the *most likely* repair at the *most likely* bug location might not always be *meaningful*. For example, while the model might be confident that a bug is located at a certain location, there might not be a suitable repair candidate that can actually fix the bug. For this reason, we filter the candidate repairs for meaningful repairs: If the model predicts a bug location, then this should be fixed by a different token of the same type. Combinations of the special NOOP operation and a repair are always meaningful (since nothing will be changed). Finally, we compute the most likely meaningful repair operation:

$$\text{replace}(t_{l'}, r') \text{ with } \langle l', r'\rangle = \text{argmax } p(\langle l, r\rangle \mid \mathcal{T}) \qquad (6)$$

D. Implementation

To effectively measure the impact of mutants and real bug fixes on the training process and to exclude variances due to implementation details, we implemented several baselines considered during our evaluation in a unified framework. In particular, we followed the design of Hellendoorn et al. [13] by implementing a common localization and repair framework with exchangeable components (e.g. token encoder, localization and/or repair modules). In the process, we reimplemented or reused state-of-the-art components for all employed subcomponents. For example, all Transformer-based baselines are built upon the official BERT implementation from the `transformer` library [37]. The localization and repair modules together with the graph-based baselines are implemented closely to the implementation of the PyBugsLab model [3]. In addition, we reimplemented the code preprocessing pipelines for tokenization [13] and graph construction [3] (used for the graph-based baselines) in independent libraries to facilitate reuse. Finally, we release the implementation code, all trained checkpoints and evaluated models on Zenodo [31] and on Github[2]. We think that these models are not only valuable for reproducing our results but also provide easy access to effective models in neural bug detection.

IV. Evaluation

We evaluate the impact of real bugs and mutants at scale on the performance of neural bug detectors for single token bugs in Python. To guide our evaluation, we designed individual experiments to answer the following two research questions:

RQ1 How does training on real bug fixes at scale impact the localization and repair performance on real bugs?

RQ2 How does training on mutants at scale impact the localization and repair performance on real bugs?

For answering the research questions, we investigate the correlation between dataset scale and the performance of neural bug localization and repair models by varying the number of mutants and real bug fixes available during training.

A. Evaluation Tasks

During our evaluation, we evaluate our localization and repair models on two types of tasks: (1) the detection and repair of real world bugs and (2) the identification of correct code.

Real bug detection and repair. We focus our study on the detection and repair of single token bugs in Python. More precisely, we consider four common types of bugs including variable misuses [2], binary operator bugs [26], unary operator bugs [3] and literal replacements [3]. To evaluate the performance of our models on these bug types, we employ the PyPIBugs benchmark [3], consisting of 2374 real-world single statement bugs and their fixes derived from open source projects. The benchmark is hand-filtered and therefore it is likely that each included bug represents a real world bug. We only considered single token bugs (which excludes argument swaps) in functions where the implementation is still publicly accessible[3]. This produced a real world test benchmark of 2028 real-world bugs. Examples of bug fixes contained in the PyPIBugs benchmark are shown in Table I.

Correct code identification. We employ the test split of the ETH Py150k [28] dataset containing 50k Python files for evaluating the performance of our models to detect correct code. We extracted all top level functions and deduplicated the resulting dataset. This process led to a test benchmark of more than 200k deduplicated Python functions. During our evaluation, we assume that the parsed function implementations are correct and that any bug reported by the models is a false alarm.

Metrics. To evaluate the effectiveness of our models in identifying and repairing real bugs, we measure the bug localization and repair performance in terms of the *joint recall* of localizing and repairing bugs in addition to the *localization recall* of identifying the bug location and *repair recall* of finding the correct bug fix given the bug location. For quantifying the precision of our models (i.e. the ability to identify correct code), we measure the *false positive rate*[4] as the ratio of reported false positives and the number of correct code examples.

B. Neural bug detector baselines

For answering our research questions, we train and evaluate bug localization and repair models based on *transformers* [13] (with relative position encodings [33]) , *graph neural networks* (GNN[5]) [3] and *GREAT* [13]. Graph based approaches like GNNs and GREAT have been shown to be highly effective in the detection of single token bugs [3]. Here, we employ recent graph based models that learn to localize and repair single token bugs based on structural-, control flow- and data flow information. For a fair comparison, all models considered in our experiments are not pre-trained prior to our training process, share a similar size (25M - 34M parameters) and see the same number of training examples during training. All models are trained to support code snippets with up to 1024 *program* tokens. During our evaluation, we consider bugs in larger functions as undetected. To support the repair of our four bug types, we initialize the vocabulary V to the set of unary and binary operators defined in Python together with a set of boolean and numeric literals ($x \in \{-2, -1, 0, 1, 2\}$). Finally, we measure the impact of real bug fixes and mutants on the training in relation to a training in a standard setup: The baselines are trained on a training dataset purely consisting of mutants (with $k = 5$ mutants injected) for 300 epochs (a 200k examples per epoch) with early-stopping on the validation set. In our experiments, we vary the number of mutants injected and the number of real bug fixes seen during training.

[2]https://github.com/cedricrupb/nbfbaselines

[3]The benchmark consists of references to bug fixing commits. We found that not all bug fixing commits were publicly accessible at the time of writing.

[4]Measuring precision directly would require that we assume some distribution of buggy and non-buggy code (which is not naturally given in our setup).

[5]Our evaluation setup differs slightly from [3] in that only individual function implementations are considered. Therefore, graph level information that would require access to the implementation context cannot be computed.

C. *Datasets*

To train the neural bug detection models, we employ two types of datasets: a general *Github corpus* of code snippets and a dataset of *real bug fixes*. To achieve comparable results, we employ existing datasets (if available). For the same reason, we also decided to focus on bugs in Python function implementations.

Github code corpus. As a general corpus of Python code, we again employ the ETH Py150k dataset [28] containing over 150k program files from popular Python projects. For training, we consider the train split consisting of 100k files and train only on Python functions obtained from the train split. We hold out 10k files which we use as validation set for evaluating the correct code identification performance of our models during the training process. We extract all top level functions and deduplicate the datasets such that the Python functions used for training only occur once in the training dataset and do not occur in our test benchmarks. In total, our training corpus contains more than 360k Python function implementations (after filtering). During our evaluation, we randomly inject mutants representing one of our four bug types. For this, we always generate a complete list of all unique mutants that can be injected and we randomly sample up to k mutants from this list. In total, we generate up to 22M mutants (dependent on the choice of k) which we use for training the neural bug detectors.

Real bug fixes. For obtaining real bug fixes at a sufficient scale, we mined a novel dataset of 33k real bug fixes from Github. As a starting point for the mining process, we employ the SSB-9M dataset [30] of over 9M general single statement bug fixes in Python. The dataset does not include the necessary implementation code itself but references the original commits in the repositories in addition to other useful metadata. This includes information about the code change as a Unix diff and whether the code change appears inside a function. Based on this information, we first pre-filtered the dataset for bug fixes that likely fall into one of our bug categories. After the filtering process, we then mined the function code from the original repositories. Since not all bug types can be identified purely on the code difference (e.g. a variable misuse requires that all variables are defined in scope), we filtered and deduplicated the resulting dataset of buggy Python functions for a second time. This process has lead to around 35k examples of real bug fixes that match at least one of our bug types. Finally, we use 33k examples for training and hold out around 2k examples as a validation set used during training.

V. Results

In this section, we discuss our evaluation results with the ultimate goal of answering our research questions.

A. *RQ1: Impact of real bug fixes*

To answer the first research question, we evaluate the impact of real bug fixes on the training process by evaluating the performance of models trained solely on real bug fixes and models fine-tuned from our baselines. We consider subsamples of 0% (no fine-tuning), 1% (334), 3% (996), 5% (1.658), 10% (3.314), 30% (9.936), 50% (16.559), 70% (23.180), 90% (29.802), 100% (33.113) of the real bug dataset. We train and fine-tune individual models for each subsample size. To obtain more stable results, the models are trained on three subsamples per sample size and evaluated on our validation sets. Averaged results are reported in Figure 2.

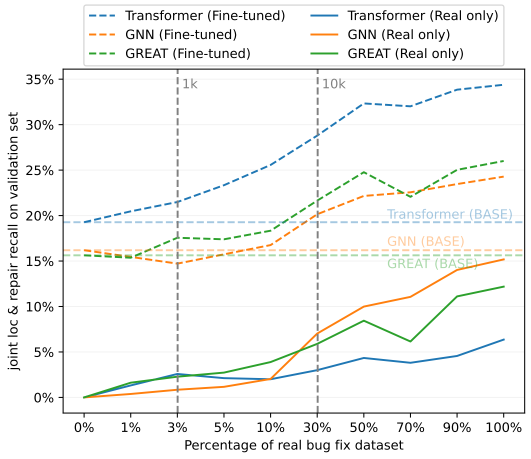


Fig. 2: Effect of real bug fixes on the performance on the validation set. The x-axis is the percentage of the bug fix dataset used for fine-tuning. The horizontal dashed lines represent our baselines purely trained on mutants and the vertical gray dashed lines mark datasets that exceed 1k and 10k examples, respectively.

Training on real bug fixes only. We find that training on real bug fixes only is insufficient for achieving high performing models. The bug localization and repair models clearly underperform when trained on real bug fixes only. The GNN is the only model that achieves comparable recall performance when trained on the complete real bug dataset. We still observe that *scaling* the number of real bugs has a significant impact on the performance of the neural bug detectors.

Fine-tuning from baselines. When fine-tuned on real bug fixes (dashed lines), the performance of our baselines continues to improve as the size of the real bug fix dataset increases. Surprisingly, the performance already improves significantly (about 3-5%) for small fine-tuning datasets of about 1000 (less than 5% of the original data set size) real bug fixes. This is encouraging as real bug fix collections of this size can be easily obtained by mining the top 1000 most popular Github projects for the respective languages [17], [19]. However, increasing the dataset size from 5% to 100% (e.g. by scaling by a factor of 20) can further improve the bug detector by up to 152%.

Real world performance. To evaluate the impact of training on real bug fixes on the benchmark performance, we also evaluate the best performing models (fine-tuned on 100% of our real bugs dataset) on our real bugs and correct code

TABLE II: Impact of training with real bug fixes on bug detection and repair on our real bugs benchmark

	Transformer				**GNN**				**GREAT**			
	FPR↓	Joint↑	Loc.↑	Repair↑	FPR↓	Joint↑	Loc.↑	Repair ↑	FPR↓	Joint↑	Loc.↑	Repair↑
Only real bugs	**12.1**	10.9	14.1	40.4	**14.8**	15.4	22.4	42.7	**10.0**	12.9	16.5	50.1
Only mutants	25.2	21.4	25.8	59.9	26.2	18.4	24.2	53.3	27.2	19.1	23.7	56.2
Mixed	27.0	24.5	29.6	65.5	20.9	18.9	24.6	56.7	27.6	21.7	25.6	60.5
Fine-tuned	26.7	**32.2**	**37.5**	**68.6**	17.4	**24.1**	**29.9**	**59.3**	21.3	**27.0**	**31.7**	**64.4**

benchmark against our baselines trained solely on mutants and the models trained solely on real bug fixes. In addition, we also consider models fine-tuned on a mix of real bug fixes and mutants (Mixed). Our evaluation results are shown in Table II. We find that the fine-tuned models significantly outperform all other training variants in terms of localization (Loc.), repair (Repair) and joint localization and repair recall (Joint). In addition, we find that training on real bug fixes generally improves the false positive rate. In fact, the lowest false positive rate is achieved by models trained *solely* on real bug fixes. Models that are fine-tuned achieve a comparable or improved false positive rate as models trained on mutants only.

Altogether, we can conclude for RQ1:

The localization and repair performance of neural bug detectors on real bugs can significantly be improved by scaling the number of real bug fixes seen during training. In comparison to techniques purely trained on mutants, the fine-tuned models show a superior performance in the detection and repair of real bugs (up to 150% improvement) and a comparable false positive rate on correct programs.

B. RQ2: Impact of mutants

While increasing the size of our real bug fix dataset is difficult, we can easily increase the number of mutants by increasing the number of mutants injected (mutation frequency). To evaluate the impact of mutants at different scales, we trained several versions of the neural bug localization and repair models by varying the mutation frequency (up to 1, 3, 5, 10, 100 and 1000 unique mutants per code snippet). For the comparison, we measured the performance of each trained model before and after fine-tuning on real bug fixes. The models are evaluated on our real bugs validation set. Figure 3 gives an overview of our results for the joint bug localization and repair recall. The configuration 0x represents a version of the evaluated models only trained on real bug fixes.

Training on more mutants. Contrary to common belief [13], we observe that increasing the number of mutations (up to a critical point of 100x mutants) leads to a performance improvement for both localization and repair[6]. This is surprising as the number of unique mutants per code snippet is limited (with an average of 85 unique mutants per code snippet) and, henceforth, buggy programs with more mutant candidates are oversampled. However, increasing the limit of mutants beyond 100 (and thereby oversampling code snippets in our dataset that provide up to 200k unique mutants) does not always yield further improvements.

[6]We observe the same trend for localization and repair independently which is not shown here for brevity.

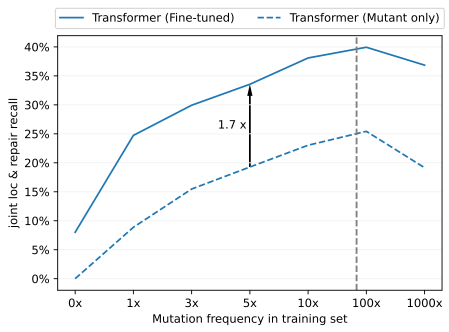
Impact on fine-tuning performance. We find that fine-tuning has an orthogonal effect on the localization and repair recall, resulting in a consistent performance improvement of 1.5x to 1.7x. This indicates that the models can effectively exploit the patterns learned during pre-training while improving their performance during fine-tuning.

Benchmark performance. We evaluate the impact of training with a higher number of mutants (100x) on the performance on our real bug benchmark. Results are reported in Table III. We find that pre-training on a larger set of mutants significantly improves the detection and repair of real bugs. However, increasing the mutation frequency comes also at the cost of an increased false positive rate. Fine-tuning on real bug fixes can mitigate this problem.

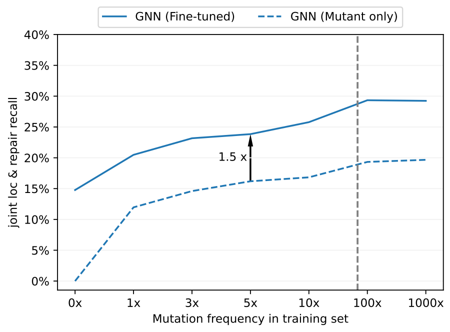
Starting from a larger Github corpus. As increasing the mutation frequency further does not yield consistent performance gains, we investigate increasing the underlying Github corpus as an alternative scaling strategy. For this, we considered the Python part of the CodeSearchNet [14] corpus which we deduplicated with respect to our training data and test benchmarks. Including the data increased our Github corpus by 384k examples. We mutated the corpus 100x, trained our models on the joint dataset and reported our evaluation results again in Table III. We find that increasing the Github corpus can further improve the localization and repair performance. Surprisingly, the training can significantly decrease the false positive rate for models trained on mutants. While this is not in the scope of this work, we expect further improvements by further increasing the corpus size. We leave the evaluation of training on larger corpora open.

We conclude for RQ2:

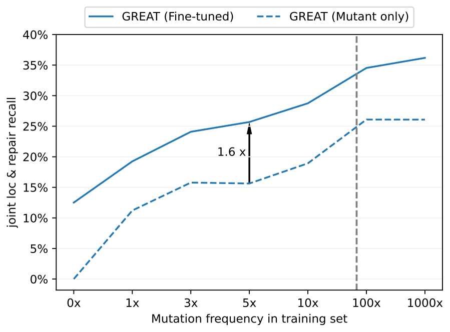
Pre-training on mutants can significantly boost the localization and repair performance of neural bug detectors. Increasing the number of mutants is a simple and effective way to further improve the performance (by up to 132%). However, models pre-trained on mutants produce a significant higher false positive rate which can only partly be mitigated by fine-tuning on real bug fixes or training on larger code corpora.



(a) Transformer



(b) GNN



(c) GREAT

Fig. 3: Effect of mutation frequency during training on the performance on the validation set. The gray dashed line represents the average number of unique mutants that can be generated per code snippet.

TABLE III: Evaluation results for bug detection and repair on our real bugs benchmark

	Transformer				**GNN**				**GREAT**			
	FPR↓	Joint↑	Loc.↑	Repair↑	FPR↓	Joint↑	Loc.↑	Repair ↑	FPR↓	Joint↑	Loc.↑	Repair↑
Baseline	25.2	21.4	25.8	59.9	26.2	18.4	24.2	53.3	27.2	19.1	23.7	56.2
+ 100x Mutants	30.0	24.9	30.4	65.6	20.4	18.7	24.2	58.3	28.2	23.8	29.0	63.9
+ Real Bug Fixes	22.0	36.7	41.9	73.5	**13.9**	25.9	31.1	64.1	22.4	33.2	38.6	68.8
+ Larger corpus	**16.5**	**38.7**	**42.7**	**76.7**	14.1	**28.9**	**34.9**	**66.5**	**17.4**	**35.6**	**40.9**	**73.6**

VI. Discussion

We now take a more qualitative look on our evaluation results by manually inspecting the raised warnings on our real bug benchmark. During this process, we mainly focus on the results produced by the best performing model[7]. In addition, we also evaluate our results in the context of the state of the art in neural bug detection.

Impact of training on real bug fixes. While neural bug detectors trained purely on mutants can only exploit how developers implement code, models fine-tuned on real bug fixes can also learn how developers make mistakes and fix them. When analyzing bugs detected only by the fine-tuned models, we find that this difference is most visible in the following two cases: for (1) *type-related* bugs and for (2) bugs occurring in code snippets with *multiple* weaknesses (e.g. potential bug locations and code quality issues at multiple locations). An example for a type-related bug only detected after fine-tuning is shown in Figure 5a. Here, the implementer intends to build the union of two sets by applying the += operator instead of the |= operator. These operators for sets are commonly confused which is also visible in our training data. For bugs occurring in code snippets with *multiple* weaknesses, we find that the fine-tuned models are better in prioritizing real bugs over code quality issues (such as unused variables). In other words, the fine-tuned models are better aligned with our understanding of real bugs.

[7]According to our results, the best performing model is a Transformer based model trained on the larger dataset with 100x mutants injected and fine-tuned on real bug fixes.

Impact of real bug fixes at scale. Some bugs that are more often discovered and fixed during development are very unlikely to appear in code repositories – even though the bug itself is common in practice. Large scale bug fix collections such as TSSB-3M [30] that contain bug fixes from a wide range of professional and hobby projects increase the chance of including these types of bugs. The bug shown in Figure 5a is an example of a common bug that is seldomly recorded in repositories. Because it is type-related, the bug can be more easily detected during code execution. Still, a good neural bug detector should warn the developer before the buggy code is submitted to a repository. Smaller bug fix collections such as the 872 wrong binary operator bugs used for training in previous work [12] do not include this type of bug. Larger collections such as PySStuBs [17] only collected two instances related to set operators. Our dataset includes more than 20x more bug fixes that match the same bug pattern. Therefore, the fine-tuned bug detectors are able to detect this type of bug as there is sufficient *support* for learning this bug pattern in the training distribution. In other words, by increasing the scale of the real bug fix dataset, we can capture unique bug types that appear seldom in code repositories.

Impact of mutants at scale. Complementary to real bug fixes, our evaluation results suggest that mutants and the size of the mutant dataset used in training can have a significant impact on the capabilities of a neural bug detector. While mutants are often not representative for real bugs, a *random* mutator can be a source for bugs that are otherwise infrequent in code repositories (as the type of bugs we discussed before). We analyze which real bugs found in PyPIBugs (including the set operator bug shown in Figure 5a) can be reproduced by our mutation strategy. For this, we mutate the code after the bug fix and we determine the percentage of bugs that can be reproduced within the first k mutants. Results are averaged across 10 runs and shown in Figure 4. We can observe that at a mutation frequency of 1000x mutants nearly all real bugs can be reproduced. This also holds true for infrequent bug types such as the set operator bug discussed before. In fact, analyzing our mutant training sets, we find that the datasets include between 31 to 3970 set operator bugs that are generated via mutation. A key problem however is the low signal-to-noise ratio (i.e. the number of real bugs relative to the number of uninformative mutants) which decreases with the scale of the mutant dataset. Therefore, as our experiments suggests, neural bug detectors can only effectively utilize these bug patterns when fine-tuned on real bug fixes.

Failure cases. While the training process significantly improved the bug detection capabilities of the neural bug detectors, they still fail to recognize a significant number of real bugs. To provide insights for future studies on neural bug detection, we analyze common failure cases. A key problem of all neural bug detectors is the limited context (consisting of the implementation of a single function in isolation). An example where the neural bug detectors fail to detect the bugs because of the limited context is depicted in Figure 5b. Here, the implementer intends to complete a given `relation` but instead uses an identified `fuser_relation`. The context is insufficient to determine that completing `relation` is preferred over `fuser_relation`. While this can be partly mitigated by including further context such as code from the same class or project, the detection of some bugs requires implicit knowledge not provided by the implementer. In general, we find that neural bug detectors are most effective for well-documented, idiomatic code.

Comparison to PyBugLab [3]. We evaluate the best performing model against PyBugLab [3] on our real bugs benchmark. PyBugLab employs a more advanced learning-based mutator and additional augmentations of the training data. As the PyBugLab models are evaluated on the same benchmark (PyPIBugs) and PyBugLab models are not publicly available, we report the results for each bug type of the original authors. Table IV provides an overview of our evaluation results. Combining a high mutation rate with fine-tuning on a large set on real bug fixes significantly improves the localization and repair performance over PyBugLab which is purely trained on mutants. Interestingly, we find that our training process has the strongest impact on the bug localization performance yielding improvements of 120% to 212%. Our evaluation results however indicate that fine-tuning on real bug fixes has an orthogonal effect on the training performance and fine-tuning PyBugLab with real bug fixes could further boost its performance.

Comparison to He et al. [12]. Finally, we compare our model against CuBERT [18], a 340M parameter masked language model, which was fine-tuned by He et al. [12] to detect real bugs. Similar to our work, CuBERT is fine-tuned in two steps by first training on mutants and then on real bug fixes. The training process is tailored towards precision: Multiple versions of CuBERT are trained that support only a single bug type. Only variable misuses and binary operator bugs are supported. For this reason, we evaluated the approach proposed by He et al. [12] on a subset of our real-world benchmarks that tackle variable misuses and binary operator bugs. While their models achieve in our experiments a false positive rate of close to zero, it only detects and repairs around 2.82% of all variable misuses and 2.23% of all binary operator bugs. In contrast, our results show that by training on a significantly higher number of real bugs and mutants we can train significantly smaller models that can achieve a more than ten times higher localization and repair performance than the method by He et al. In addition, we can confirm that training on real bug fixes can significantly decrease the false positive rate. In the end, it is highly dependent on the application scenario whether a low false positive rate or high localization and repair performance is preferred. For example, we envision future applications of high recall models in scenarios where false repairs can be easily rejected via patch validation techniques [39] or by other elimination techniques for false positives [5], [20].

TABLE IV: Comparison to PyBugLab

Bug type	PyBugLab [3]		Transformer (ours)	
	Loc.	Repair	Loc.	Repair
Wrong Assign Op	20.0	68.9	**27.3**	**77.3**
Wrong Binary Op	27.2	54.3	**57.7**	**87.3**
Wrong Boolean Op	27.6	**96.9**	**39.6**	96.2
Wrong Comparison Op	33.7	66.1	**45.6**	**69.2**
Wrong Literal	21.6	78.4	**30.2**	**82.6**
Variable Misuse	35.3	70.5	**42.7**	**75.2**

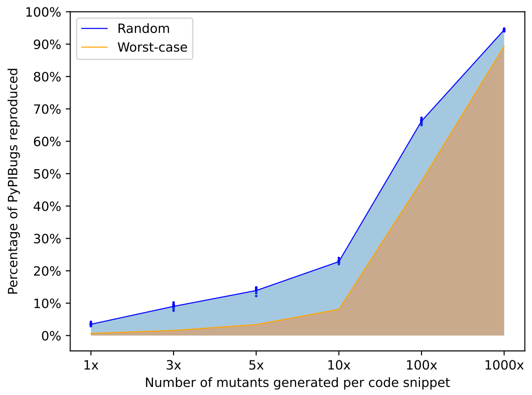


Fig. 4: Inversion test on PyPIBugs for reproducing real bugs via mutation. The x-axis is the number of mutants generated and y-axis is the percentage of real bugs regenerated within the first k mutants. Mutants are sampled randomly and we average over 10 samples (blue line). The orange line is the worst-case score assuming that the real bug is always sampled last.

```python
# AssignOp: |= instead of +=
if event == 'highstate-start':
    minions += set(data['minions'])
elif event == 'highstate':
    minions.discard(data['minion'])
```

(a) A wrong assignment operator bug that can only be fixed after training on real bug fixes. Developer often confuse += and |= for set unions.

```python
# VarMisuse: relation instead of fuser_relation
def _find_completion(fuser, relation):
    for fuser_relation in fuser.fusion_graph.get_relations([...]):
        if fuser_relation._id == relation._id:
            return fuser.complete(fuser_relation)
    return None
```

(b) A variable misuse not detected by any localization and repair model. The context is insufficient to decide that the implementer intends to complete the relation and not the fuser relation.

Fig. 5: Bugs taken from the PyPIBugs benchmark. Code is reformatted to fit the figure.

VII. Threats to Validity

Although a variety of neural bug detectors have been developed in recent years, there does not exist a universally accepted setup for training and benchmarking these models. Therefore, even though we implemented our baselines close to the reference implementation, the resulting trained models might behave differently than in the setup they were originally developed for. To still achieve comparable results, we designed our evaluation to match prior studies [13] on neural bug detectors as close as possible. For example, we adopted the same publicly accessible Github corpus ETH Py150k, a similar architectural design and similar baselines as employed by Hellendoorn et al. [13]. To further support a wider range of bug types which allowed us to exploit the hand validated benchmark PyPIBugs, we adjusted the architecture and mutation process similar to Allamanis et al. [3]. Still, our evaluation results for the baseline algorithms are slightly different than the results of prior studies. For example, we found that while graph-based models such as GNNs and GREAT still perform better in a low data regime (when trained only on real bug fixes), Transformers show a surprisingly strong performance when trained on mutants. We mainly attribute this difference in performance to our choice of a *relative* attention mechanism over an *absolute* attention mechanism used in prior work (which led to a boost of up to 20% on our validation set). Note however that Hellendoorn et al. [13] already anticipated this result (even though they only evaluated on variable misuses) when the models are trained for a longer duration – which we did by training on approximately 2.4x more training examples.

In contrast to the results of Allamanis et al. [3], we observe that graph-based models underperform in our evaluation setup which we attribute to two main differences: (1) for a fair comparison, all models only have access to the function implementation without the implementation context which prohibits the computation of type related or call structure related information exploited by the graph-based models and (2) all models were trained on a different (potentially smaller) dataset. Although integrating the type of information and training on a larger dataset would potentially benefit all baselines, the performance ranking between architectures might differ. However, since our experiments showed that the performance gain due to training on real bug fixes is orthogonal to the effect of training on mutants, we expect that adapting our evaluation setup has little to no influence on our evaluation outcome. Finally, while our training approach is general enough to be applied to other languages, our approach is evaluated on specific single token bugs in Python. The effect of training on real bug fixes might vary for other programming languages or other bug types.

VIII. Related Work

We discuss the most closely related previous or concurrent works that (1) tackle neural bug detection with alternative training strategies, (2) exploit real bug fixes for automatic program repair or code mutations and (3) consider alternative pre-train-and-fine-tune techniques.

Neural Bug Detection. Detection and repair of single token bugs has been explored in previous work [2], [3], [13], [25], [26], [29], [36]. Allamanis et al. [2] addressed the detection and repair of variable misuse bugs (which we also considered in this work) by representing programs as graphs. Vasic et al. [36] proposed a joint model for the same task and Hellendoorn et al. [13] explored alternative program representations. These techniques all have in common that they do not learn from real bug fixes but from artificially mutated code. In contrast, in this work, we showed that integrating real bug fixes in the training process is crucial for the localization and repair of real bugs. More recent work [3], [25], [29] also showed that the quality of training data is important for effective bug localization and repair. For example, employing a more realistic mutator [25], [29] (i.e. a mutator that is more likely to reproduce a real bug) or learning to inject hard to find bugs [3] can both improve the localization and repair performance. However, the integration of these approaches often increases the complexity by requiring to learn a mutation operator either prior or concurrent to the training process. With our systematic study, we showed that integrating real bug fixes, while relying on simpler and easier to implement mutation operators, can be sufficient to obtain a significant improvement in real bug localization and repair performance. He et al. [12] explored whether adjusting the training distribution to a more realistic distribution improves the *precision* of neural bug detectors. Similar to our work, their model is trained both on mutants and real bug fixes. While their method significantly improved the precision (i.e. the number of correct programs classified as buggy), they did not consider bug datasets at the scale needed for obtaining high recall models (i.e. models that can detect and repair a significant number of real bugs). Our evaluation shows that training both on mutants and real bug fixes at a large scale can significantly improve the number of bugs detected and repaired *while* training on real bug fixes can also decrease the false positive rate.

Automatic program repair. While first approaches in neural bug detection focused on code mutants, there is a long history of learning-based methods trained purely on real bug fixes in automatic program repair (APR) [4], [6], [7], [11], [16], [22]–[24], [34], [35], [43], [44]. SequenceR [7], for example, learns from thousands of real bug fixes to predict one-line bug patches. Dlfix [22] and CoCoNuT [24] improved the repair performance by proposing more effective learning strategies. In this work, we evaluated the impact of real bug fixes both on the bug detection *and* repair performance. We found that training on real bug fixes significantly improves the ability to detect and localize real bugs. In addition, APR techniques are often trained on real bug fixes only without considering mutants for the training process. Training additionally on mutants could further improve the repair performance. This observation is also supported by DrRepair [40] and Self-APR [42] which showed that pre-training repair models on artificial errors improved the repair performance on real bugs. Also utilizing large language models [15] and fine-tuning them for the APR task can improve the repair performance. Still, their approaches rely on a compiler or user-provided unit tests to detect bugs. Neural bug detectors are designed to complimentary with the goal of finding bugs that are typically missed by a compiler or during testing.

Code mutation. Code mutation addresses the inverse problem of injecting a bug into a correct program. Tufano et al. [34] and Patra and Pradel [25] showed that bug fixes can be effectively leveraged to learn code mutations by learning to replicate the original bug. Interestingly, Yasunaga et al. [41] showed that repeatedly training a breaker and fixer which initially learn from real bug fixes but then provide training data for each other actually improves the performance of the fixer to repair syntactic bugs. While our work showed that real bug fixes are also crucial for bug detection, we believe that exploiting real bug fixes in the mutation process for training bug detectors can be a promising direction for future work.

Pre-training and fine-tuning. Pre-training on large corpora of fuzzy data and then fine-tuning on a specific task with a smaller dataset has been shown to be highly successful in domains such as natural language processing [8], [27], image processing [21] and most recently programming language processing [9], [15], [18], [38]. In contrast to our models, these techniques are often pre-trained on a generic unrelated task where data is available before fine-tuning them on a specific task. We showed that training with the same architecture with largely the same objective of identifying and repairing buggy (or mutated) code can significantly improve the performance of neural bug detectors.

CuBERT [18] showed that pre-training on a generic corpus of Python code can improve the detection performance on variable misuses. However, the authors employed mutants instead of real bug fixes in the fine-tuning phase. In contrast, we pre-train on mutants and then fine-tune on real bug fixes. A combination of these two approaches would be interesting and we leave this open for future work.

IX. Conclusion

In this work, we have investigated the effect of training on mutants plus real bug fixes at a large scale on the performance of bug localization and repair models. For this, we systematically evaluated and compared neural bug localization and repair models trained on mutants, real bug fixes and mixtures thereof at different dataset scales. We found that existing neural bug detectors can effectively utilize both mutants *and* real bug fixes during training by first pre-training on mutants and then fine-tuning on real bug fixes. Our evaluation on thousands of real bugs obtained from real Python projects showed that existing neural bug detectors were limited by the scale of the employed training data and that training with real bug fixes and mutants on sufficiently large datasets can significantly improve the localization and repair of real bugs. As future work we see the integration of more realistic data in the training process of neural bug localization and repair models. For example, training on more realistic mutants could boost the detection performance and reduce the number of false positives even before fine-tuning on real bug fixes. In addition, it might also be interesting to integrate even larger code corpora in the training process or start from pre-trained models of code. Finally, to conclude, we demonstrate in this study that neural bug localization and repair models can effectively learn from real bug fixes to localize and repair real bugs.

Data Availability Statement

The implementation of all evaluated neural bug localization and repair models are open source and available. To improve the replication of our work and facilitate future work, we are releasing all trained models together with the pre-training and fine-tuning code on Zenodo [31]. In addition, our replication package also includes novel datasets produced in this work.

References

[1] A. Alaboudi and T. D. LaToza, “An exploratory study of debugging episodes,” *CoRR*, vol. abs/2105.02162, 2021. [Online]. Available: https://arxiv.org/abs/2105.02162

[2] M. Allamanis, M. Brockschmidt, and M. Khademi, “Learning to represent programs with graphs,” in *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net, 2018. [Online]. Available: https://openreview.net/forum?id=BJOFETxR-

[3] M. Allamanis, H. Jackson-Flux, and M. Brockschmidt, “Self-supervised bug detection and repair,” *Advances in Neural Information Processing Systems*, vol. 34, pp. 27 865–27 876, 2021.

[4] J. Bader, A. Scott, M. Pradel, and S. Chandra, “Getafix: Learning to fix bugs automatically,” *Proceedings of the ACM on Programming Languages*, vol. 3, no. OOPSLA, pp. 1–27, 2019.

[5] A. Bella, C. Ferri, J. Hernández-Orallo, and M. J. Ramírez-Quintana, “Calibration of machine learning models,” in *Handbook of Research on Machine Learning Applications and Trends: Algorithms, Methods, and Techniques*. IGI Global, 2010, pp. 128–146.

[6] S. Chakraborty, Y. Ding, M. Allamanis, and B. Ray, “Codit: Code editing with tree-based neural models,” *IEEE Transactions on Software Engineering*, 2020.

[7] Z. Chen, S. Kommrusch, M. Tufano, L.-N. Pouchet, D. Poshyvanyk, and M. Monperrus, "Sequencer: Sequence-to-sequence learning for end-to-end program repair," *IEEE Transactions on Software Engineering*, vol. 47, no. 9, pp. 1943–1959, 2019.

[8] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," in *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. Minneapolis, Minnesota: Association for Computational Linguistics, Jun. 2019, pp. 4171–4186. [Online]. Available: https://aclanthology.org/N19-1423

[9] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou, "Codebert: A pre-trained model for programming and natural languages," in *Findings of the Association for Computational Linguistics: EMNLP 2020, Online Event, 16-20 November 2020*, ser. Findings of ACL, T. Cohn, Y. He, and Y. Liu, Eds., vol. EMNLP 2020. Association for Computational Linguistics, 2020, pp. 1536–1547. [Online]. Available: https://doi.org/10.18653/v1/2020.findings-emnlp.139

[10] J. Gu, Z. Lu, H. Li, and V. O. K. Li, "Incorporating copying mechanism in sequence-to-sequence learning," in *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics, ACL 2016, August 7-12, 2016, Berlin, Germany, Volume 1: Long Papers*. The Association for Computer Linguistics, 2016. [Online]. Available: https://doi.org/10.18653/v1/p16-1154

[11] R. Gupta, S. Pal, A. Kanade, and S. Shevade, "Deepfix: Fixing common c language errors by deep learning," in *Thirty-First AAAI conference on artificial intelligence*, 2017.

[12] J. He, L. Beurer-Kellner, and M. T. Vechev, "On distribution shift in learning-based bug detectors," in *International Conference on Machine Learning, ICML 2022, 17-23 July 2022, Baltimore, Maryland, USA*, ser. Proceedings of Machine Learning Research, K. Chaudhuri, S. Jegelka, L. Song, C. Szepesvári, G. Niu, and S. Sabato, Eds., vol. 162. PMLR, 2022, pp. 8559–8580. [Online]. Available: https://proceedings.mlr.press/v162/he22a.html

[13] V. J. Hellendoorn, C. Sutton, R. Singh, P. Maniatis, and D. Bieber, "Global relational models of source code," in *International conference on learning representations*, 2019.

[14] H. Husain, H. Wu, T. Gazit, M. Allamanis, and M. Brockschmidt, "Codesearchnet challenge: Evaluating the state of semantic code search," *CoRR*, vol. abs/1909.09436, 2019. [Online]. Available: http://arxiv.org/abs/1909.09436

[15] N. Jiang, K. Liu, T. Lutellier, and L. Tan, "Impact of code language models on automated program repair," in *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*. IEEE, 2023, pp. 1430–1442. [Online]. Available: https://doi.org/10.1109/ICSE48619.2023.00125

[16] N. Jiang, T. Lutellier, and L. Tan, "Cure: Code-aware neural machine translation for automatic program repair," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 1161–1173.

[17] A. V. Kamienski, L. Palechor, C.-P. Bezemer, and A. Hindle, "Pysstubs: Characterizing single-statement bugs in popular open-source python projects," in *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*. IEEE, 2021, pp. 520–524.

[18] A. Kanade, P. Maniatis, G. Balakrishnan, and K. Shi, "Learning and evaluating contextual embedding of source code," in *Proceedings of the 37th International Conference on Machine Learning, ICML 2020, 13-18 July 2020, Virtual Event*, ser. Proceedings of Machine Learning Research, vol. 119. PMLR, 2020, pp. 5110–5121. [Online]. Available: http://proceedings.mlr.press/v119/kanade20a.html

[19] R. Karampatsis and C. Sutton, "How often do single-statement bugs occur?: The manysstubs4j dataset," in *MSR*. ACM, 2020, pp. 573–577. [Online]. Available: https://doi.org/10.1145/3379597.3387491

[20] A. Kharkar, R. Z. Moghaddam, M. Jin, X. Liu, X. Shi, C. B. Clement, and N. Sundaresan, "Learning to reduce false positives in analytic bug detectors," in *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*. ACM, 2022, pp. 1307–1316. [Online]. Available: https://doi.org/10.1145/3510003.3510153

[21] A. Kolesnikov, L. Beyer, X. Zhai, J. Puigcerver, J. Yung, S. Gelly, and N. Houlsby, "Big transfer (bit): General visual representation learning," in *European conference on computer vision*. Springer, 2020, pp. 491–507.

[22] Y. Li, S. Wang, and T. N. Nguyen, "Dlfix: Context-based code transformation learning for automated program repair," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 602–614.

[23] F. Long, P. Amidon, and M. Rinard, "Automatic inference of code transforms for patch generation," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, 2017, pp. 727–739.

[24] T. Lutellier, H. V. Pham, L. Pang, Y. Li, M. Wei, and L. Tan, "Coconut: combining context-aware neural translation models using ensemble for program repair," in *Proceedings of the 29th ACM SIGSOFT international symposium on software testing and analysis*, 2020, pp. 101–114.

[25] J. Patra and M. Pradel, "Semantic bug seeding: a learning-based approach for creating realistic bugs," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021, pp. 906–918.

[26] M. Pradel and K. Sen, "Deepbugs: A learning approach to name-based bug detection," *Proceedings of the ACM on Programming Languages*, vol. 2, no. OOPSLA, pp. 1–25, 2018.

[27] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, P. J. Liu *et al.*, "Exploring the limits of transfer learning with a unified text-to-text transformer." *J. Mach. Learn. Res.*, vol. 21, no. 140, pp. 1–67, 2020.

[28] V. Raychev, P. Bielik, and M. Vechev, "Probabilistic model for code with decision trees," *ACM SIGPLAN Notices*, vol. 51, no. 10, pp. 731–747, 2016.

[29] C. Richter and H. Wehrheim, "Learning realistic mutations: Bug creation for neural bug detectors," in *2022 IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2022, pp. 162–173.

[30] ——, "TSSB-3M: Mining single statement bugs at massive scale," in *19th IEEE/ACM International Conference on Mining Software Repositories, MSR 2022, Pittsburgh, PA, USA, May 23-24, 2022*. ACM, 2022, pp. 418–422. [Online]. Available: https://doi.org/10.1145/3524842.3528505

[31] ——. (2023) How to train your neural bug detector: Artificial vs real bugs. Zenodo. [Online]. Available: https://doi.org/10.5281/zenodo.7900059

[32] R. Sennrich, B. Haddow, and A. Birch, "Neural machine translation of rare words with subword units," in *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Berlin, Germany: Association for Computational Linguistics, Aug. 2016, pp. 1715–1725. [Online]. Available: https://aclanthology.org/P16-1162

[33] P. Shaw, J. Uszkoreit, and A. Vaswani, "Self-attention with relative position representations," in *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT, New Orleans, Louisiana, USA, June 1-6, 2018, Volume 2 (Short Papers)*, M. A. Walker, H. Ji, and A. Stent, Eds. Association for Computational Linguistics, 2018, pp. 464–468. [Online]. Available: https://doi.org/10.18653/v1/n18-2074

[34] M. Tufano, C. Watson, G. Bavota, M. Di Penta, M. White, and D. Poshyvanyk, "Learning how to mutate source code from bug-fixes," in *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2019, pp. 301–312.

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

[36] M. Vasic, A. Kanade, P. Maniatis, D. Bieber, and R. Singh, "Neural program repair by jointly learning to localize and repair," in *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net, 2019. [Online]. Available: https://openreview.net/forum?id=ByloJ20qtm

[37] T. Wolf, L. Debut, V. Sanh, J. Chaumond, C. Delangue, A. Moi, P. Cistac, T. Rault, R. Louf, M. Funtowicz, J. Davison, S. Shleifer, P. von Platen, C. Ma, Y. Jernite, J. Plu, C. Xu, T. L. Scao, S. Gugger, M. Drame, Q. Lhoest, and A. M. Rush, "Transformers: State-of-the-art natural language processing," in *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*. Online: Association for Computational Linguistics, Oct. 2020, pp. 38–45. [Online]. Available: https://www.aclweb.org/anthology/2020.emnlp-demos.6

[38] C. S. Xia and L. Zhang, "Less training, more repairing please: revisiting automated program repair via zero-shot learning," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and*

Symposium on the Foundations of Software Engineering, 2022, pp. 959–971.

[39] J. Yang, A. Zhikhartsev, Y. Liu, and L. Tan, "Better test cases for better automated program repair," in *Proceedings of the 2017 11th joint meeting on foundations of software engineering*, 2017, pp. 831–841.

[40] M. Yasunaga and P. Liang, "Graph-based, self-supervised program repair from diagnostic feedback," in *International Conference on Machine Learning*. PMLR, 2020, pp. 10 799–10 808.

[41] ——, "Break-it-fix-it: Unsupervised learning for program repair," in *International Conference on Machine Learning*. PMLR, 2021, pp. 11 941–11 952.

[42] H. Ye, M. Martinez, X. Luo, T. Zhang, and M. Monperrus, "Selfapr: Self-supervised program repair with test execution diagnostics," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, 2022, pp. 1–13.

[43] H. Ye, M. Martinez, and M. Monperrus, "Neural program repair with execution-based backpropagation," in *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*. IEEE, 2022, pp. 1506–1518.

[44] Q. Zhu, Z. Sun, Y.-a. Xiao, W. Zhang, K. Yuan, Y. Xiong, and L. Zhang, "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*, 2021, pp. 341–353.