Program Analysis using Unsupervised Learning *

Ashwin Murali[1]

University of Stuttgart, 70569 Stuttgart, Germany
st181872@stud.uni-stuttgart.de

Abstract. Program repair involves detecting and fixing bugs in computer programs using automated techniques. Deep Learning models can learn patterns and relationships within code by analysing large datasets of code. Due to the limited availability of labelled datasets, many works have created buggy codes by perturbing the good samples using some heuristics. However, a model trained on such synthetically-generated data do not extrapolate well to the real distribution of bad inputs. It is thus highly important to build robust techniques with Unsupervised Learning and Self-Supervised Learning paradigms due to the inadequacy of labelled datasets. This work extends an Unsupervised Learning approach for automatic program repair by improving the accuracy of the code error fixer for invalid syntactical errors.

Keywords: Program Repair · Code Analysis

1 Introduction

Code bases now are extremely complex and updated frequently. Code repair refers to the process of detecting and fixing bugs in software code. The problem of fixing bugs, yet remains mostly a developer's task. There exists patterns in source code such as variable and function names, comments and data flow which can be informative. This information can be used to identify and fix certain bugs. Such a learnt program repair tool using Deep Learning techniques can offer the promise to improve how developers develop software. The code for this work is found in this repository.

The structure of the following sections in this report is organised as: Background, Solution Design, Implementation, Evaluation and Conclusion and Next steps.

2 Background

Previous works use labeled datasets comprising of source code edits made by code creators. Since labeled datasets are costly, various methods for code repair rely on preparing synthetic generated paired data by perturbing good code to obtain

* Supported by The Software Lab, University of Stuttgart

perturbed code. Specifically, [4] has perturbed good code to prepare synthetic paired data and has trained a bug fixer using a seq2seq model. [5] improved on it by incorporating perturbations with common errors made by programmers. However, [1] found that such synthetically perturbed data do not match the distribution of real bad examples. They mainly focused on parenthesis errors and invalid syntax errors while there exists in practice other kind of bugs including argument swapping, wrong binary operator, wrong boolean operator and variable misuse bug. [6] showed an alternative technique for learning to generate realistic bugs. My work extends [1] technique by augmenting the data with inclusion of other bug kinds. For more detailed review about automatic code repair, [7] can be referred.

Availability of labelled datasets consisting of (bad, good) pairs of code is not adequate to train a complex model. There have been works that create training data by corrupting good examples of code using heuristics (e.g., dropping tokens). However, it has been shown that such a model trained on data generated through perturbations do not fit well to the real distribution of bad inputs. To circumvent this, [1] proposes a new training approach, Break-It-Fix-It (BIFI) which has two main ideas. Given a critic that assesses the quality of an input, the goal is to train a fixer that converts a bad example into a good one and add good (fixed) code back to the training data. The second idea is to train a breaker to generate realistic bad code from good code. The fixer and breaker are then trained recursively by exchanging the outputs of each other. [1] is referred as BIFI in the rest of the report.

3 Problem Analysis

The code error fixer should map a buggy code into a good code such that no error results from a critic (compiler). It is very important for a good code error fixer to handle wide range of error categories to perform well in real-world scenarios. BIFI focuses mainly on unbalanced parenthesis errors, indentation errors, missing colon errors, missing newline errors, missing parenthesis errors and redundant comma errors. However, BIFI cannot handle specific error categories including variable misuse errors, argument swapping errors, wrong operator errors and wrong literal errors with the same accuracy. This is because the model was not exposed to codes involving these error categories during training. This work focuses on preparing a limited labelled dataset involving the above mentioned errors. Later, the trained BIFI model was continued training with the collected data to improve its code error fixing accuracy.

4 Solution Design

4.1 Overview of BIFI

The BIFI approach has a fixer and a breaker as its two main components. Initially, the two components are trained with synthetically paired data. Then,

both are improved simultaneously through rounds of data generation and training. Breaker generates buggy code from clean code and the generated buggy code is later used as input to the fixer. Fixer generates clean code by correcting the bug from the buggy input. This fixed bug free code is also used as input to the Breaker in the next rounds. BIFI uses a critic that can distinguish good code from buggy code which is also used to verify that the fixer produces good code and the breaker produces buggy code. Figure 1 shows the overall architecture of BIFI.

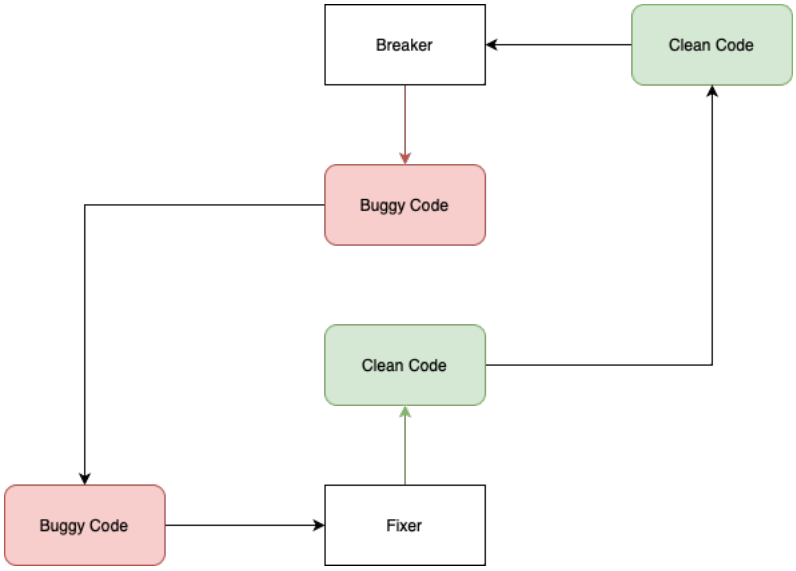

Fig. 1. BIFI Architecture

4.2 Improving Fixer's accuracy

The accuracy of the fixer can be further improved by exposing it to bug kinds it has not been trained on. The design of this work's implementation is shown in Figure 2. After extracting PyPIBugs data in the required format for BIFI, the BIFI model can be retrained on the extracted data to learn to fix invalid syntactical bug kinds. The results of the fixer after retraining is tabulated in the Evaluation section.

5 Implementation

5.1 PyPIBugs Dataset

This dataset has been collected by [2] by crawling all commits across all PyPi python packages found in the libraries.io v1.6.0 dataset. A candidate bug fix was

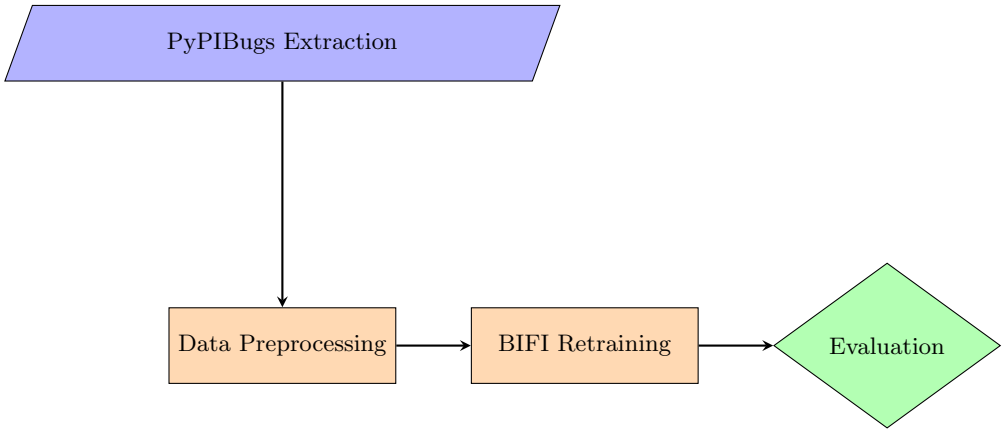

Fig. 2. Flowchart showing solution design

considered when change that is identical to the observed commit. There are a total of 2374 samples with wide range of real-world "stupid simple bugs". A jsonl file providing the Github git URLs of the projects along with the commit SHAs, file paths, and bug types has been released. The breakdown of the different kind of bugs in the dataset includes,

Table 1. Bug Kinds in PyPIBugs

Bug Kind	Count	Percentage
`Argument Swapping`	282	11.9
`Wrong Assignment`	45	1.9
`Wrong Binary Operator`	81	3.4
`Wrong Boolean Operator`	192	8.1
`Wrong Comparison Operator`	407	17.1
`Wrong Literal`	88	3.7
`Wrong Misuse`	1278	53.8
Total	2374	100

A python script with code extracting the appropriate representation of the code for each of the bugs in PyPIBugs dataset has been written. The script not only just extracts the bugs but also preprocesses and adapts each bug to the BIFI dataset format. As the preprocessing step, a chunk selection technique has been implemented to limit each sample with an upper bound to maximum code tokens. A chunk for each code sample is selected by locating the line where the bug is present and also including few lines before and after the bug line to provide some context. 371 samples were removed due to errors originating while pulling the git commit. This leaves a total of 2003 samples for fine-tuning of the

BIFI fixer model. Each sample is then saved in the format required for the BIFI model training.

5.2 BIFI Retraining

The PyPIBugs dataset has been adapted to the BIFI data format and has been split into train, test and validation data. The training of the BIFI model was resumed from the checkpoint and carried out with the PyPIBugs dataset. Similar to the flow in [1], three rounds of data generation and model training was performed recursively and the results are evaluated.

5.3 Hyperparameter Tuning

Many hyperparameters were available in the BIFI code base. Learning rate, epochs, number of layers in encoder, number of layers in decoder and update frequency were the hyperparameters that have been tuned for this work.

6 Evaluation

A repair is considered to be successful if the output code has no compiler errors. The fixer's repair accuracy on the test set containing only mainly invalid syntactical errors is the evaluation metric considered for this problem.

Table 2. Fixer Performance on PyPIBugs

Method	Accuracy Percentage
`Vanilla BIFI`	2.1
`BIFI+PyPIBugs` Round 0	4.375
`BIFI+PyPIBugs` Round 1	7.375
`BIFI+PyPIBugs` Round 2	8.0

The results show that the accuracy of the fixer for the PyPIBugs dataset is not satisfactory. There is only a very minimal improvement in accuracy percentage after retraining the BIFI model with the dataset. Each code sample is converted into code tokens where few tokens are masked into the unkown, COMMENT, NUMBER and STRING categories. A possible reason for the low accuracy could be that many tokens in the test dataset were masked into unknown category and thus did not match the ground truth. Another reason could have been the low number of training samples in the PyPIBugs dataset. Nevertheless, the extracted and preprocessed dataset can be used for other potential code fixing problems.

7 Conclusion and Next Steps

7.1 Conclusion

The accuracy of the BIFI model with and without exposing to PyPIBugs dataset is not that great. It is important for the code error fixer to perform well against a wide range of bugs in software code. Hence to improve the accuracy of BIFI model for wide bug kinds, other Deep Learning paradigms like prompt-based techniques can be employed. The dataset can also be extended to contain more realistic code snippets containing syntactical errors and their fixes and can be used as to fine-tune similar models like BIFI for program analysis.

7.2 Next Steps

Figure 3 shows where PyPIBugs component fits in the overall flow of BIFI. Next steps would be to also train Breaker with the PyPIBugs dataset and follow data generation and training steps repeatedly. If the breaker performs well, training data can be augmented with the inclusion of its output and there is no need for more labelled data. PyPIBugs dataset can also be extended to include more samples containing less lines of code. The BIFI model can be tested on a different programming language dataset using Transfer Learning or few-shot learning paradigms.

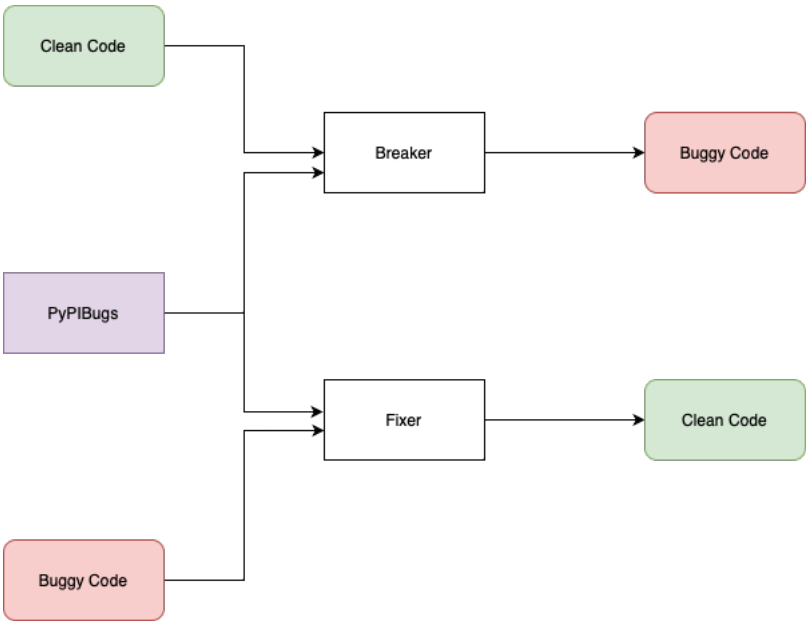

Fig. 3. Next steps

References

1. Break-It-Fix-It: Unsupervised Learning for Program Repair. Michihiro Yasunaga, Percy Liang. ICML 2021.
2. Self-Supervised Bug Detection and Repair. M. Allamanis, H. Jackson-Flux, M.Brockschmidt. NeurIPS 2021.
3. Automated Code Repair. Claire Le Goues, Michael Pradel and Abhik Roychoudhur. CACM 2019.
4. DeepFix: Fixing Common C Language Errors by Deep Learning. R. Gupta, S. Pal, A. Kanade, and S. Shevade. AAAI 2017.
5. Graph-based, Self-Supervised Program Repair from Diagnostic Feedback. Yasunaga, M and Liang, P. ICML 2020.
6. Semantic bug seeding: a learning-based approach for creating realistic bugs. J Patra and M Pradel. 2021
7. Automatic Software Repair. Martin Monperrus. 2017.
8. Getafix: Learning to Fix Bugs Automatically. J Bader, M Pradel. 2019.