Break it - Message it - Fix it : Learning to Repair Python Programs using Error Messages without Labelled Data

Anonymous ACL submission

Abstract

In recent years there is an increasing demand to reduce the gap in development to deployment. It has been estimated that developers spend almost 20% of their time in fixed problems with their code. Therefore tools which can automatically repair code can help accelerate the DevOps cycles. In this work we build upon recent success of deploying neuro-symbolic approaches for automatic code repair. In our approach, we use a dataset of python code, viz, CodeNet, which represents data distribution for human generated code. We train two neural modules a breaker and a fixer, which are trained iteratively, along with a symbolic module Pylint. The breaker learns to introduce errors in the code, the symbolic module acts as a Critic and is able to fragment the error by identifying the line, as well provide the error type with a specific exception message. The Fixer utilizes the exception message to repair the erroneous line in the code. We are able to cover 32 different syntax errors, and iterative training based on back translation actually helps improve the performance of the Fixer.

1 Introduction

In the world of programming, if there exists a tool like a Fixer which is capable of fixing the bugs by generating the corresponding solutions to the errors, a lot of effort and time will be saved, thereby producing good quality error-free codes. The number of coding errors is directly proportional to the complexity and size of the code. While static analyzers can help to point out the errors, manually rectifying them is a very tedious task and very time consuming to understand those errors first and then fix them. This issue highlights the need of a fixer capable of fixing errors from the code, thereby transforming it to error-free code.

In this work, we propose to build a fixer for Python which uses the error message for a corrupted code in input and tries to output the correct fix for the mentioned error, leading to the error-free code. Inorder to further improve the fixer's performance, we also train a breaker capable of corrupting good code to produce bad code as output. These bad codes will be more realistic compared to synthetically generated corrupted codes. Our fixer is then retrained on these bad codes inorder to make it more capable to fix errors in codes. We update the fixer and breaker in an iterative manner to improve both their performances. Figure 1 denotes our overall approach. To evaluate this approach, we build our own Python dataset based on CodeNet data (Puri et al., 2021). We synthetically corrupt the data points using random insert/swap/delete operations inorder to deliberately create error in a line of code.

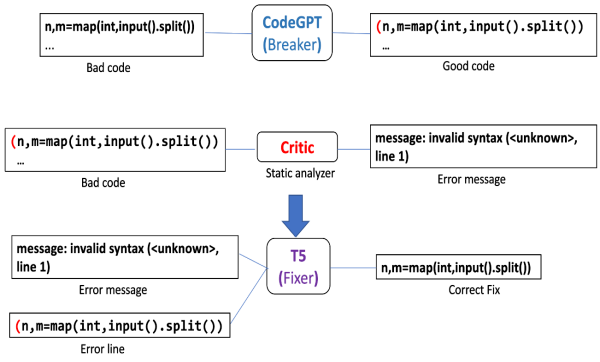

Figure 1: High-level diagram of our approach

We use a static code analysis tool, Pylint (code analysis for python, 2021), which serves as a critic in our work. Given a buggy code, it helps to detect the error and it provides the error line number, type of error detected and the corresponding error message associated with it. This information helps our fixer in fixing the detected bug easily.

Our contributions in this work are two-fold:

- We build a new code repair dataset in Python by adding perturbations to the CodeNet (Puri et al., 2021) Python benchmark data.

- We train a fixer by a critic-verified backtranslation (Yasunaga and Liang, 2021) approach to make it capable of fixing code errors.

This paper is organized in the following manner : Section 2 covers the brief literature survey, Section 3 talks about the dataset description, Section 4 covers our proposed methodology, Section 5 covers our results and analysis and finally we have conclusion in Section 6.

2 Related Works

There have been several works which learn to repair code errors, for example: (Just et al., 2014), (Bader et al., 2019), (Chen et al., 2021), (Mesbah et al., 2019) and so on. In (Yasunaga and Liang, 2021), the authors proposed a critic-verified backtranslation approach inorder to train a fixer on buggy codes. (Berabi et al., 2021) suggested an approach of using error messages to generate fixes for the bugs in code. (Yasunaga and Liang, 2020) solves the problem of learning to repair programs from diagnostic feedback. Motivated with the works of (Berabi et al., 2021) and (Yasunaga and Liang, 2021), we propose a novel critic-verified backtranslation-based approach which uses error messages to repair buggy codes in Python.

3 Dataset Description

The benchmark data in CodeNet consist of compilable coding solutions to different problems. The authors in (Puri et al., 2021) ensure that the solutions are not near-duplicates of other code samples. Since we require corrupted codes to train our fixer, we follow our corrupting procedure to convert these good codes into bad codes so that we can form a paired dataset. We try to insert errors in good codes such that they contain common programming errors like typo, punctuation mistake etc. so that the errors seem to be realistic. We modify the good codes released in CodeNet(Puri et al., 2021) Python benchmark data by adding perturbations in the form of insert/swap/delete operation. We randomly choose a line of code and decide one of the operations randomly with appropriate weights. Based on the operation chosen, we either insert noisy tokens in a line or swap tokens from two different indices in a line or delete a random token from a line of code.

After the perturbation procedure, we find that there can exist a paired dataset (Good code, Bad code) consisting of 208921 data points. From this paired dataset D, we keep 10% of the data reserved as the test set for evaluating our fixer model. From the remaining dataset, we equally divide it into two splits - forming D_1 and D_2 respectively each of size 94025. We use D_1 for initial round of training for both fixer and breaker.

$$\text{Fixer} : Train^{Bad->Good}(D_1)$$

$$\text{Breaker} : Train^{Good->Bad}(D_1)$$

Fixer is responsible for transforming a bad code into a good code and the breaker does the opposite.

4 Methodology

This section describes in brief the critic used, the neural architectures used for breaker and fixer, the training procedure and the experimental details.

4.1 Critic

In our work, we require an error detector which will help us to identify good codes from the bad codes. We use Pylint (code analysis for python, 2021) as a static code analyzer for python. Pylint takes a python code as input and generates the corresponding error report. Figure 2 highlights a sample bad code and the corresponding error report produced by Pylint. Each report mainly consists of a symbol, error message, message id and the location of the error in the code snippet. We only restrict the Pylint output to error messages so that we can filter out codes having only errors and not focusing on any warnings or the ones violating any standard coding convention and so on. This error detector serves as our critic, examining whether the fixer has successfully fixed the error and thus the output can be added to the D_1 and whether the breaker has succesfully added noise to the good code, helping the generated output to get added to D_2. Pylint has been able to capture 32 different types of coding errors which have occurred due to the added perturbations in our dataset. Three most frequent error types are *syntax error*, *used-before-assignment* message and *undefined-variable message*.

4.2 Fixer

We leverage the Text-to-Text Transfer Transformer (T5) (Raffel et al., 2020) architecture as our fixer. We follow (Berabi et al., 2021) to build a fixer by providing the inputs based on the critic's error report in the following manner :

```
def main():
    N = int(input())
    A = list(map(int, input().split(' ')))

    insertion_sort(A, N)

def insertion_sort(A, N):
    print(' '.join([str(n) for n in A]))
    for i in range(1, N):
        v = A[i]
        j = i – 1
        while j >= 0 and A[j] > v:
            A[j+1] = A[j]
            j -= 1
        A[j+1] = v

        print(' '.join([str(n) for n in A]))

    return A
if __name__ == '__main__':
    main()
```

(a) Corrupted code

```
{
  type: error,
  module: s020476450_corrupted,
  obj: ,
  line: 3,
  column: 42,
  path: p02255/s020476450_corrupted.py,
  symbol: syntax-error,
  message: unindent does not match any outer indentation level (<unknown> line 3),
  message-id: E0001
}
```

(b) Error Report produced by the critic

Figure 2: Pylint producing error report for the given bad code

fix error type error message error line : error context

Here error type refers to the type of error occurred due to the error line, error message refers to the message given by the static analyzer to understand the error and error context refers to a single line of code both before and after the error line. We consider the *symbol* as shown in Figure 2 as the error type.

We frame this task of fixing code bugs as a text-to-text task and hence use this sequence-to-sequence based Transformer (Vaswani et al., 2017) architecture, T5. The output of the fixer model is the correct line of code, which when replaced with the error line, will convert the bad code to a good code. The T5 model is pretrained on a large corpus of English data for various NLP tasks. We finetune the pretrained T5 model on our code repair dataset. Since the code fragments use different natural language identifiers and keywords, our intuition is that this model will understand the coding terminologies better when finetuned on the code repair data.

4.3 Breaker

We use the codeGPT (Lu et al., 2021) model pretrained on Python programming language as the neural architecture for the breaker. CodeGPT has the same architecture and training objective like GPT-2 (Radford et al., 2019). Our intuition behind using this model to implement the breaker function is to generate proper syntactic code as output with proper indents (if required). This can be possible only if the pretraining has been done on code corpus. CodeGPT is pretrained from scratch on python functions so that the BPE (Byte Pair Encoder) (Sennrich et al., 2016) vocabulary is obtained on the code corpus. We feed the correct code as input and auto-regressively generate the bad code as output.

4.4 Training Procedure

We apply a critic verified back-translation procedure like in (Yasunaga and Liang, 2021). Our breaker model helps in back-translation by generating bad codes from good codes. Initially, we train the breaker and the fixer on the paired D_1 dataset. Algorithm 1 highlights our training procedure. We restrict the breaker to run prediction on only the data points in the val split of D_1 due to computational constraints. As a result of this retraining, breaker and fixer learn better based on the outputs of each other in an iterative manner, which improves the overall performance of the fixer.

4.5 Experimental details

For the fixer, we choose the T5-small model (Raffel et al., 2020) which consists of 60 million parameters with 6 layers in the encoder and decoder blocks. We finetune the T5 model for 4 epochs and choose the best model in terms of evaluation loss for prediction. For the breaker network, we choose the codeGPT-small (Lu et al., 2021) model which consists of 12 layers of Transformer decoders. We download the pretrained models released by Hugging Face [1]. We finetune it for 10 epochs and choose the best model similarly for prediction. We keep the batch size of 1 for both the networks due to computational constraints. We keep the maximum sequence length of 256 and 512 for fixer and breaker respectively.

[1] https://huggingface.co/models

Algorithm 1 Training Procedure

1: Train the breaker and fixer models on D_1 initially.
2: *loop* :
3: Apply the current fixer to real bad examples in D_2 and keep outputs that are fixed.
4: Train the breaker on this new paired dataset from the earlier checkpoint.
5: Apply the current breaker to real good examples in D_1 and keep outputs that are broken.
6: Retrain the fixer from the earlier checkpoint on the paired dataset resulting from step 5.
7: **goto** *loop*

5 Results and Analysis

	Round	EM	Repair
t5-small	Round-0	84.87%	87.62%
	Round-1	85.43%	87.96%

Table 1: Performance of the Fixer Model. EM refers to the exact match Accuracy and Repair refers to the Repair Accuracy.

We choose *Exact Match accuracy* and *Repair accuracy* (Yasunaga and Liang, 2021) as the two metrics for evaluating the performance of our fixer. We calculate Exact match accuracy by the number of data points whose ground truths and predictions have matched completely on token level, divided by the number of data points in the test set. Repair accuracy is calculated by the number of data points which have been considered as error-free by the critic divided by the size of test set.

Table 1 refers to the performances of the fixer model on the test set. Based on our hypothesis, we see that the performance of T5-small model as a fixer improves in Round-1. There is a gain of 0.65% in exact match accuracy and a gain of 0.38% in repair accuracy. We believe that retraining the fixer on predictions of the breaker on both the train and val splits of D_1 would improve the performance significantly. Our results also indicate that the fixer model is able to generate more compilable patches as the probable fix which do not always match the ground truth, thus the repair accuracy metric is always higher than the exact match accuracy metric.

6 Conclusion

In this work, we attempt to automatically fix syntactic bugs in Python codes using error messages. We use Transformer-based architecture as our Fixer module to generate possible fix for the bug. Retraining the fixer on critic-verified data points which have been generated as a result of the predictions by the breaker module yields a jump in performance. As a part of this work, we create a code repair dataset in Python using the error messages, based on the CodeNet data. We also show how the breaker can generate buggy codes, which upon usage by the fixer during retraining, makes it more robust. In our future work, we plan to analyse the performance of the fixer on individual types of syntactic error and also try to incorporate other types of error apart from the syntactic bugs.

References

Johannes Bader, Andrew Scott, Michael Pradel, and Satish Chandra. 2019. Getafix: Learning to fix bugs automatically. *Proc. ACM Program. Lang.*, 3(OOPSLA).

Berkay Berabi, Jingxuan He, Veselin Raychev, and Martin Vechev. 2021. Tfix: Learning to fix coding errors with a text-to-text transformer. In *Proceedings of the 38th International Conference on Machine Learning*, volume 139 of *Proceedings of Machine Learning Research*, pages 780–791. PMLR.

Zimin Chen, Steve Kommrusch, Michele Tufano, Louis-Noël Pouchet, Denys Poshyvanyk, and Martin Monperrus. 2021. Sequencer: Sequence-to-sequence learning for end-to-end program repair. *IEEE Transactions on Software Engineering*, 47(9):1943–1959.

Pylint : code analysis for python. 2021. https://www.pylint.org/.

René Just, Darioush Jalali, and Michael D. Ernst. 2014. Defects4j: A database of existing faults to enable controlled testing studies for java programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014, page 437–440, New York, NY, USA. Association for Computing Machinery.

Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin B. Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, Ming Gong, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie

Liu. 2021. Codexglue: A machine learning benchmark dataset for code understanding and generation. *CoRR*, abs/2102.04664.

Ali Mesbah, Andrew Rice, Emily Johnston, Nick Glorioso, and Edward Aftandilian. 2019. Deepdelta: Learning to repair compilation errors. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2019, page 925–936, New York, NY, USA. Association for Computing Machinery.

Ruchir Puri, David Kung, Geert Janssen, Wei Zhang, Giacomo Domeniconi, Vladmir Zolotov, Julian Dolby, Jie Chen, Mihir Choudhury, Lindsey Decker, Veronika Thost, Luca Buratti, Saurabh Pujar, Shyam Ramji, Ulrich Finkler, Susan Malaika, and Frederick Reiss. 2021. Codenet: A large-scale ai for code dataset for learning a diversity of coding tasks.

Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. 2019. Language models are unsupervised multitask learners.

Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. 2020. Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of Machine Learning Research*, 21(140):1–67.

Rico Sennrich, Barry Haddow, and Alexandra Birch. 2016. 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)*, pages 1715–1725, Berlin, Germany. Association for Computational Linguistics.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc.

Michihiro Yasunaga and Percy Liang. 2020. Graph-based, self-supervised program repair from diagnostic feedback. In *International Conference on Machine Learning (ICML)*.

Michihiro Yasunaga and Percy Liang. 2021. Break-it-fix-it: Unsupervised learning for program repair. In *International Conference on Machine Learning (ICML)*.