Fix-Filter-Fix: Intuitively Connect Any Models for Effective Bug Fixing

Haiwen Hong[1] **Jingfeng Zhang**[1] **Yin Zhang**[1]* **Yao Wan**[2] **Yulei Sui**[3]
[1] College of Computer Science and Technology, Zhejiang University, Hangzhou, China
[2]School of Computer Sci. & Tech., Huazhong University of Science and Technology, China
[3] School of Computer Science, University of Technology Sydney, Australia
{honghaiwen96, zhjf, zhangyin98}@zju.edu.cn,
wanyao@hust.edu.cn, yulei.sui@uts.edu.au

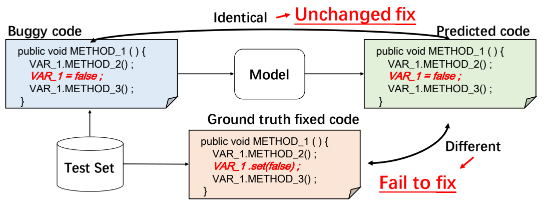
Abstract

Locating and fixing bugs is a time-consuming task. Most neural machine translation (NMT) based approaches for automatically bug fixing lack generality and do not make full use of the rich information in the source code. In NMT-based bug fixing, we find some predicted code identical to the input buggy code (called ***unchanged fix***) in NMT-based approaches due to high similarity between buggy and fixed code (e.g., the difference may only appear in one particular line). Obviously, unchanged fix is not the correct fix because it is the same as the buggy code that needs to be fixed. Based on these, we propose an intuitive yet effective general framework (called Fix-Filter-Fix or F^3) for bug fixing. F^3 connects models with our ***filter mechanism*** to filter out the last model's unchanged fix to the next. We propose an F^3 theory that can quantitatively and accurately calculate the F^3 lifting effect. To evaluate, we implement the Seq2Seq Transformer (ST) and the AST2Seq Transformer (AT) to form some ***basic*** $\boldsymbol{F}^3$ instances, called $\boldsymbol{F}^3_{ST+AT}$ and $\boldsymbol{F}^3_{AT+ST}$. Comparing them with single model approaches and many model connection baselines across four datasets validates the effectiveness and generality of F^3 and corroborates our findings and methodology.

1 Introduction

Locating and repairing bugs of programs automatically is important in software engineering. Many approaches (Tufano et al., 2019; Chen et al., 2019; Chakraborty et al., 2020) based on the Neural Machine Translation (NMT) have achieved promising performance for semantic bug fixing. The basic idea is to automatically translate a buggy code fragment into a fixed patch. However, there still exist some limitations. Most of them (1) do not fully exploit the information of source code and only partly using the textual or structured information; (2) are single model architectures with poor generality.

*Corresponding author: Yin Zhang.



```
public void METHOD_1 ( ) {
   VAR_1.METHOD_2() ;
   VAR_1 = false ;
   VAR_1.METHOD_3() ;
}
```

```
public void METHOD_1 ( ) {
   VAR_1.METHOD_2() ;
   VAR_1 = false ;
   VAR_1.METHOD_3() ;
}
```

```
public void METHOD_1 ( ) {
   VAR_1.METHOD_2() ;
   VAR_1 .set(false) ;
   VAR_1.METHOD_3() ;
}
```

Figure 1: A motivating example.

We find that many current NMT-based bug fixing models often predict exactly the same output code as the input buggy codes as Figure 1 shows (see red lines), which we call ***unchanged fix***. The input code is buggy, while the unchanged fix does not make any changes to the buggy code, so the unchanged fix is a failed fix obviously. It is because the buggy code and fixed code are always very similar and the vocabularies of the buggy and fixed code are closely overlapped in bug fixing. This phenomenon may also exist in many tasks with similar vocabularies before and after translation, such as automatic post-editing and text style transfer.

In fact, this is an ***unsupervised phenomenon***, i.e., without comparing the ground-truth fixed code, it is possible to determine directly whether the prediction is correct by knowing only the result of the model prediction and the input buggy code. This has led to the following question we aim to answer in this paper: "*Can we filter out those **unchanged fixes** and then refine the bug fixing process with a different model?*". A similar scenario also exists when revising a paper. Multiple revisions can always find more errors than single revisions. Intuitively, in bug fixing, multiple models in tandem can provide better fixing than a single model does.

Based on the above observation, we propose a general and intuitive framework for bug fixing (called Fix-Filter-Fix or F^3) with high performance and marginal extra cost. F^3 uses a ***filter mechanism*** to connect several different ***learners*** (individual models for bug fixing).

The filter mechanism needs to directly filter out some buggy code fragments that a learner fails to fix ***without checking the ground-truth fixed code in the dataset***, and then feed the filtered buggy code into the next learner. Each learner in F^3 should be able to fix a portion of the buggy code that others cannot fix. Our filter mechanism in this paper compares a learner's predicted results with the input buggy code, filters out the unchanged fix, and continues the corresponding buggy code to the next learner for processing, but it may not an optimal filter mechanism. An optimal filter mechanism can filter out all the buggy code fragments that a learner fails to fix.

It is intuitive that F^3 can improve performance. To make the intuition more precise, we propose a theory to ***precisely calculate*** the specific performance improvement of F^3 combined with multiple learners without experimental verification. Since source code contains textual and structural information, we apply Seq2Seq Transformer (ST) to fix bugs based on the textual representation of code, while the AST2Seq Transformer (AT) is based on abstract syntax tree (AST), the structural information of code. We connect these two learners in different orders to implement F^3 instances, called F^3_{ST+AT} and F^3_{AT+ST}.

We compare its performance with the single model baselines (Tufano et al., 2019) and model connection baselines on four datasets transformed by BFP and CodRep datasets (Chen and Monperrus, 2018). Experimental results demonstrate that our F^3_{ST+AT} outperforms all the baselines at a low cost. Then we experimentally investigate the effects when using different orders or number of learners on F^3 as experimental corroboration of our theoretical proof. Finally, we analyze the generality and broader impact of F^3.

In summary, the key contributions are as follows:

- We, for the first time, reveal and study the ***unchanged fix*** issue existing in NMT-based bug fixing tasks. This is an unsupervised phenomenon. We present and analyze the causes and functions of the unchanged fix scenarios in detail. In addition, we analyze the ways in which the unchanged fix phenomenon can be used in a broader domain.

- We propose an intuitive framework called F^3 based on unchanged fix to comprise multiple learners through a filter mechanism for iterative bug fixing. We provide a theory that can accurately calculate the specific improvement of F^3 for each task and each backbone, thus validating that F^3 can be useful in any area where unchanged fix exists.

- We connect Seq2Seq Transformer (ST) with AST2Seq Transformer (AT) to form basic F^3 instances and evaluate their performance on four datasets. Experimental results show that our F^3 outperforms all the single model baselines and model connection baselines. We also provide analysis for the generality of F^3.

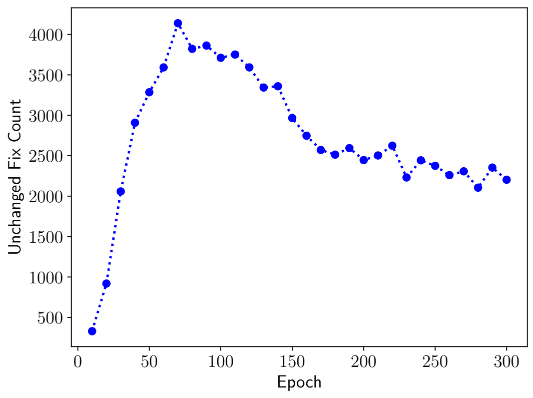


Figure 2: Trend plot of the number of unchanged fixes as a function of training epochs, where the dataset is BFP_{small} and the model is Seq2Seq Transformer.

2 Unchanged Fix Issue

When applying NMT to the problem of bug fixing, the buggy code is translated into the fixed code for the purpose of fixing. In this process, as shown in Figure 1, it often happens that the sequence predicted by the model, and the sequence of buggy code at the time of input, are exactly the same, a phenomenon that we call unchanged fix.

The input code is buggy, and the predicted unchanged fix is exactly the same as the buggy code, which means that the unchanged fix must also be buggy, and therefore it must not be a successful fix. In other words, we do not need to actually test whether the predicted code can run, or know how the code that is actually fixed should look like. Just by comparing the sequences predicted by the model, with the input sequences, we can filter out a batch of cases where the fix obviously fails, so unchanged fix contains unsupervised properties.

This phenomenon is caused by the fact that the input and output before and after translation are highly similar, or the vocabularies are highly similar, which may cause the model to "***accidentally***" generate exactly the same results as the input.

According to our tests on the Seq2Seq Transformer, as shown in Figure 2, when the training epoch increases from 1 to 300, the proportion of unchanged fix in the test set increases sharply and then decreases. The proportion is low at the beginning because the initialized sequence is completely chaotic. As the epoch increases, it slowly learns the approximate distribution of the dataset, and thus the phenomenon of unchanged fix starts to appear. After that, the distribution learned by the model becomes more and more accurate, and the number of model prediction errors gradually decreases, thus the number of unchanged fixes decreases.

Therefore, unchanged fix can be considered as a kind of lapse phenomenon when the model is not in a perfect state, just like a new painting student who wants to draw a tiger but accidentally draws a cat. The observed lapse scenario suggests that generative models, like humans, may learn a general generative logic first and then continuously improve and refine the learned knowledge. Unchanged fix implies that the model itself has a general learning of the distribution of the data, but does not have the particularly precise details. This type of model is more common in many complex tasks of NLP based on NMT, which means that the unchanged fix issue may have a generality that is not limited to the bug fixing task.

3 Preliminaries

For the purpose of quantitative proof for the properties of F^3, we denote all the buggy codes in the test set as $\mathcal{T}$, the multiple learners in F^3 as $\mathcal{M} = \{M_1, M_2, \ldots, M_{|\mathcal{M}|}\}$, where $|\mathcal{M}|$ is the number of learners. The part of F^3 before the M_i (including the M_i) is called F_i^3. In particular, given a buggy program $x \in \mathcal{T}$, the learner M_i will generate a fixed program y. In this paper, we classify y into the following four different sets:

- ***Correct fix***: It represents a code fragment produced by a learner successfully fixes the bug. That means, after being fixed by M_i, the fixed programs are identical to the correct code in the ground-truth dataset. We denote these fixed programs as $\mathcal{C}(M_i)$.
- ***Changed but wrong fix***: It represents a code fragment that is inconsistent with the input code fragment and the ground truth correct code. We denote these programs after being fixed by the learner M_i as $\mathcal{CW}(M_i)$.
- ***Unchanged fix***: It represents the fixes produced by a learner have not modified/changed anything of a buggy code fragment. We denote these fixes by the learner M_i as $\mathcal{U}(M_i)$.
- ***Wrong fix***: It represents a fix that is inconsistent with the ground-truth fixing programs, including unchanged fix and changed but wrong fix. We denote these programs after being fixed by the learner M_i as $\mathcal{W}(M_i)$.

The goal of our filter mechanism is to filter out those unchanged fixes from each learner's predicted/generated fixes and feed them into the next learners. We can obtain the following rules for M_1:

$$|\mathcal{T}| = |\mathcal{C}(M_1)| + |\mathcal{W}(M_1)| = |\mathcal{C}(M_1) \cup \mathcal{W}(M_1)| \quad (1)$$

and for any M_i:

$$|\mathcal{W}(M_i)| = |\mathcal{CW}(M_i)| + |\mathcal{U}(M_i)| = |\mathcal{CW}(M_i) \cup \mathcal{U}(M_i)| \quad (2)$$

Similarly, we define $\mathcal{C}(F_i^3)$, $\mathcal{U}(F_i^3)$, $\mathcal{CW}(F_i^3)$, $\mathcal{W}(F_i^3)$ for the F^3 results.

4 Fix-Filter-Fix (F^3) Framework

4.1 An Overview

Figure 3 shows the workflow of F^3. In the first stage, the first learner M_1 digests the buggy programs $\mathcal{T}$ and outputs the first stage results. Then our filter mechanism classifies those results into ***unchanged fix*** $\mathcal{U}(M_1)$ and the others (***correct fix*** $\mathcal{C}(M_1)$ and ***changed but wrong fix*** $\mathcal{CW}(M_1)$ but we cannot distinguish each other). The $\mathcal{C}(M_1)$ and $\mathcal{CW}(M_1)$ are sent to the final results, while the $\mathcal{U}(M_1)$ are filtered out and fed into the learner M_2 in the next stage. The following stages will follow a similar process. Note that all results from the learner $M_{|\mathcal{M}|}$ in $|\mathcal{M}|$ stage are passed to the final results of F^3, since there is no latter learners.

We implement two basic F^3 instances, named F$^3_{ST+AT}$ and F$^3_{AT+ST}$, which are composed of two Transformer-based learners, Seq2Seq Transformer (ST) to represent the textual sequence of code tokens and AST2Seq Transformer (AT) to represent the ASTs extracted from codes. To verify the extensibility, in the experiments of RQ2, we add another Seq2Seq RNN (SR) learner to the end of them to achieve a better performance than the two-stage F^3. In the following sections, we will elaborate the

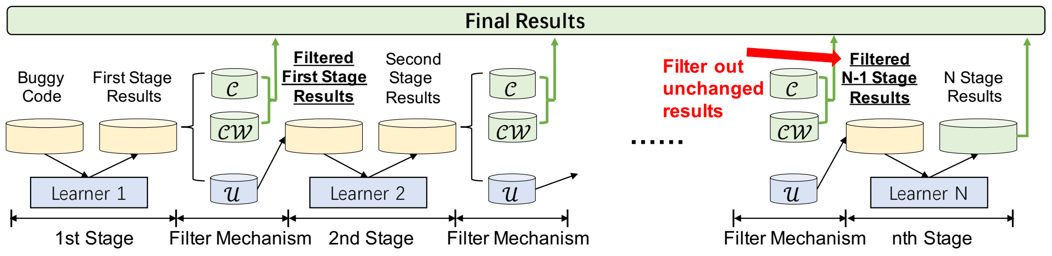


Figure 3: Workflow of the F^3 framework with our Filter Mechanism (filter out the unchanged fix). $\mathcal{C}$ is the correctly fixed programs set, $\mathcal{CW}$ is the changed but wrong programs set, $\mathcal{U}$ is the unchanged programs set.

main components of F$^3_{ST+AT}$ and F$^3_{AT+ST}$ framework.

4.2 Learners

We can choose any existing bug fixing model as a F^3 learner. In this paper, we implement Seq2Seq Transformer (ST) and AST2Seq Transformer (AT) as learners for experiments, and the outputs of these models are as follows:

$$\boldsymbol{o}(ST) = Transformer\,(\boldsymbol{e}_t) \qquad t \in buggy\_seq \tag{3}$$

$$\boldsymbol{o}(AT) = Transformer\,(\boldsymbol{e}_t) \qquad t \in AST\_seq \tag{4}$$

where $\boldsymbol{e}_t$ is the token embedding for the buggy code token t sampled from a buggy token sequence in Eq.3, and for AST token t in the AST token sequence AST_seq generated from the AST parsed from a buggy program in Eq.4. Through DFS (Depth-First Search), we get the traversed sequence of AST and feed it into our AST2Seq Transformer. The two learners fix bugs from different perspectives i.e., textual information and structural information of code.

4.3 Our Filter Mechanism

F^3 is so understandable that we can intuitively determine that it improves performance because subsequent learners fix bugs that the previous ones could not. We provide quantitative calculations to make the intuition more precise, so that we can determine the amount of improvement in F^3's performance by a direct calculation, ***without the need for tedious experimental testing***. In this section, we theoretically prove the effects of the number and order of learners on the performance of F^3 with our proposed filter mechanism.

With the two learners, F^3 will keep correctly fixed programs $\mathcal{C}(M_1)$ from the first learner and give unchanged programs $\mathcal{U}(M_1)$ to the next. Among $\mathcal{U}(M_1)$, the next learner will fix those programs in its correctly fixed set $\mathcal{C}(M_2)$:

$$|\mathcal{C}(F^3_2)| = |\mathcal{C}(M_1)| + |\mathcal{C}(M_2) \cap \mathcal{U}(M_1)| \tag{5}$$

When adding a new learner M_{i+1}, it will fix codes that it can fix correctly in the F^3_i unchanged set:

$$|\mathcal{C}(F^3_{i+1})| = |\mathcal{C}(F^3_i)| + |\mathcal{C}(M_{i+1}) \cap \mathcal{U}(F^3_i)| \tag{6}$$

We need to know $\mathcal{U}(F^3_i)$. The M_{i+1} works among the $\mathcal{U}(F^3_i)$, and can only leave the unchanged programs that are both in $\mathcal{U}(F^3_i)$ and $\mathcal{U}(M_{i+1})$, thus:

$$\mathcal{U}(F^3_{i+1}) = \mathcal{U}(M_{i+1}) \cap \mathcal{U}(F^3_i) \tag{7}$$

$$|\mathcal{C}(F^3_{i+1})| - |\mathcal{C}(F^3_i)| = |\mathcal{C}(M_{i+1}) \cap \bigcap_{n=1}^{i} \mathcal{U}(M_n)| \tag{8}$$

It shows that when we add a new learner to the F^3, the updated F^3 outperforms the old F^3 as long as there are programs in the intersection of previous learners' sets of unchanged programs that the new learner can fix, that means ***the newly added learners should have the ability to fix programs with different granularities***. That is why we implement two different learners, i.e., the Seq2Seq Transformer and AST2Seq Transformer. However, with our filter mechanism, we cannot establish a deterministic quantitative relationship between the unchanged programs set $\mathcal{U}$ and the correctly fixed set $\mathcal{C}$. Therefore, we cannot determine whether the new learner performs better or worse than the old F^3. With our filter mechanism, we can guarantee that the new F^3 will not perform worse than the old F^3, but there is no guarantee that the new F^3 will perform better than the new learner added to the old F^3.

To explore the effects of the order of learners on F^3, we consider the F^3 containing two learners: the first learner M_1 and the last learner M_2. Then we change the order of the two learners to obtain a framework denoted as $F^3_{reversed}$. We have:

$$|\mathcal{C}(F^3)| = |\mathcal{C}(M_1)| + |\mathcal{C}(M_2) \cap \mathcal{U}(M_1)| \tag{9}$$

Datasets	BFP$_{small}$	BFP$_{medium}$	CodRep$_{real}$	CodRep$_{abstract}$
Training Set	46,680	52,364	11,868	11,868
Validation Set	5,835	6,546	1,483	1,483
Test Set	5,835	6,545	1,483	1,483
Total	58,350	65,455	14,834	14,834

Table 1: The number of programs in the four datasets.

$$|\mathcal{C}(F^3_{reversed})| = |\mathcal{C}(M_2)| + |\mathcal{C}(M_1) \cap \mathcal{U}(M_2)| \quad (10)$$

$$\begin{aligned}|\mathcal{C}(F^3)| - |\mathcal{C}(F^3_{reversed})| = |\mathcal{C}(M_1) \cap \mathcal{CW}(M_2)| \\ - |\mathcal{C}(M_2) \cap \mathcal{CW}(M_1)|\end{aligned} \quad (11)$$

With our filter mechanism, $\mathcal{CW}(M_1)$ and $\mathcal{CW}(M_2)$ are indeterminate so that learners' order is likely to affect the performance of F^3.

4.4 Is our Filter Mechanism Optimal?

Our filter mechanism is not optimal and the optimal filter mechanism should be able to find all the wrong fixes without checking the ground-truth fixed code. We rewrite the Eq. 6 and Eq. 7 as:

$$|\mathcal{C}(F^3_{i+1})| = |\mathcal{C}(F^3_i)| + |\mathcal{C}(M_{i+1}) \cap \mathcal{W}(F^3_i)| \quad (12)$$

$$\mathcal{W}(F^3_{i+1}) = \mathcal{W}(F^3_i) \cap \mathcal{W}(M_{i+1}) \quad (13)$$

Based on some set-theoretic derivations, we get:

$$|\mathcal{C}(F^3_{i+1})| = |\mathcal{T}| - |\bigcap_{n=1}^{i+1} \mathcal{W}(M_n)| = |\bigcup_{n=1}^{i+1} \mathcal{C}(M_n)| \quad (14)$$

With this optimal filter mechanism, the more different learners involved in the F^3, the better performance F^3 will gain. Moreover, the learners' order has no effect on the final results, since the $\bigcup_{n=1}^{i+1} \mathcal{C}(M_n)$ is the same for any order of learners.

These equations above mean that as long as we know the individual performance of learners, we can calculate the performance for all F^3 with different learners' order and quickly find the best F^3 based on these learners without experimental validations. We will validate the theoretical calculations with experimental results in RQ2.

5 Experiment and Analysis

Our paper is biased towards verifying the theoretical validity of the F^3, and the experiments are just one of the supporting evidences. F^3 may improve performance in any area where unchanged fixes exist, such as automatic post-editing and text style transfer, which is intuitive and theoretically proven by us. Here we just pick the bug fixing as a typical task for experimental validation, and these experiments should also hold for other F^3-compliant domains. In the experiments, we focus on investigating the following research questions:

- ***RQ1 (Performance Boost):*** How much of the performance boost does F^3 provide?
- ***RQ2 (Impact of Learner Order and Count):*** Is the theoretical performance of F^3 accurate under different orders and number of learners?
- ***RQ3 (Cost Evaluation):*** How much will F^3 increase the cost?
- ***RQ4 (Generality Analysis):*** How to combine learners with different input and output?

5.1 Performance Boost (RQ1)

5.1.1 Baselines

There are a variety of NMT-based bug fixing methods. SUQUENCER (Chen et al., 2019) only conducts bug fixing without localization, and Graph2diff (Tarlow et al., 2020) is mainly designed for compilation errors while we focus on semantic bugs. Our approach translates the entire buggy code into correct code, including bug location and fixing, which is similar to the Seq2Seq RNN model in (Tufano et al., 2019), hence we pick it as our baseline. In fact, F^3 can fuse existing models and what it needs to verify most is its enhancement to existing models rather than a direct comparison with existing models. Therefore, comparing F^3 with the learners within it (Seq2Seq Transformer and AST2Seq Transformer) is what matters most.

Besides, considering that F^3 is a method of model connection, in order to reflect the superiority of F^3 and the usefulness of the unchanged fix, we design a variety of different connection methods as the baselines for comparison. These connections are based on learners who have been trained individually, and the difference is in the strategy of decision making, not in the training method. Taking the connection between two learners as an example, these connection methods are designed as follows ("Parallel" stands for two models arranged in parallel, accepting all input cases separately and outputting the results. "Series" stands for connecting two learners in series in order of overall performance from highest to lowest.):

Parallel Random For each input case, the output of two learners is randomly taken as the final output of the framework.

Parallel Prior For each output case, there is a 75% probability that the output of the overall better performing learner is taken and a 25% probability that the output of the overall worse performing learner is taken as the output of the final framework. That is, the decision is biased in favor of the model with better performance.

Series Random After the first learner accepts all input cases, according to the output results, 50% of the original input cases are randomly selected to enter the second learner, and the output of the second learner overwrites the output of the corresponding cases of the previous learner. That is, for all cases received by the second learner, the final output of the framework is the output of the second learner, otherwise, it is the output of the first learner.

Series Prior Only 25% of the original input cases will be picked into the second learner, the rest of the design is the same as Series Random.

Parallel Unchanged Random For the current case, if the output of the current learner is unchanged fix, the output of the other learner is directly adopted as the final output of the framework, and if both are unchanged fix, one is randomly selected as the output. For the remaining cases (i.e., cases for which neither learner outputs unchanged fix), the outputs of two learners are randomly selected as the final output of the framework.

Parallel Unchanged Prior Exactly the same as the Parallel Unchanged Random design, except that for remaining cases, there is a 75% probability that the output of the overall better performing learner is taken and a 25% probability that the output of the overall worse performing learner is taken as the output of the final framework.

Parallel Unchanged Order Exactly the same as the Parallel Unchanged Random design, except that for all remaining cases, the output of the learner with better overall performance is taken as the final output of the framework.

Series Unchanged Random As the first learner accepts all input cases, it inputs all cases of unchanged fix to the second learner, and then takes 50% of the remaining cases and inputs them to the second learner. The output of the second learner overwrites the output of the corresponding case of the first learner, as long as it is not an unchanged fix of second learner.

Series Unchanged Prior Only 25% of the remaining input cases will be picked into the second learner, the rest of the design is the same as Series Unchanged Random.

5.1.2 Dataset and Preprocessing

We conduct all our experiments on BFP and CodRep divided as Table 1.

- **BFP** (Tufano et al., 2019). BFP is derived from the commits of some Java projects on github. We use abstract BFP with two collections, BFP$_{small}$ (token length $<=$ 50) and BFP$_{medium}$ (50 $<$ token length $<=$ 100).

- **CodRep** (Chen and Monperrus, 2018). CodRep is from open-source Java projects of (Hata et al., 2012; Li et al., 2019; Monperrus and Martinez, 2012; Scholtes et al., 2016; Tufano et al., 2017; Zhong and Su, 2015; Zhou et al., 2012). We filter out the methods with token length $>$ 25 and $<$ 100, and we call it CodRep$_{real}$. Then we do abstraction on it and get an abstract dataset called CodRep$_{abstract}$.

5.1.3 Implementation Details

For AST2Seq Transformer and Seq2Seq Transformer, we follow the implementation of Fairseq (Ott et al., 2019). For AST2Seq Transformer, we first parse the buggy methods to ASTs, and use the ASTs as input, fixed method sequences as output. For Seq2Seq RNN, we implement it using PyTorch and set the hyperparameters according to (Tufano et al., 2019). We train all models separately on the training set of BFP$_{small}$, BFP$_{medium}$, CodRep$_{real}$ and CodRep$_{abstract}$.

During inference, we connect Seq2Seq Transformer and AST2Seq Transformer with our filter mechanism to be the F^3_{ST+AT} and F^3_{AT+ST} for testing. The programs are fixed correctly only if they are identical to their ground-truth fixed programs in the test set. The evaluation metric is accuracy.

5.1.4 Results and Analysis

Table 2 shows the accuracy comparison among single models, different connections methods and our F^3_{ST+AT} and F^3_{AT+ST} on BFP$_{small}$, BFP$_{medium}$, CodRep$_{real}$ and CodRep$_{abstract}$ datasets.

It is worth mentioning that baselines containing the “Prior” field are given a higher priority to the learner with better performance. For example, with BFP$_{small}$, in the baseline containing the “Series” field, ST is first, and the corresponding F^3 is

Approach	BFP$_{small}$	BFP$_{medium}$	CodRep$_{real}$	CodRep$_{abstract}$
Seq2Seq RNN (SR)	530 / 5835 (9.08%)	209 / 6545 (3.19%)	45 / 1483 (3.03%)	105 / 1483 (7.08%)
Seq2Seq Transformer (ST)	822 / 5835 (14.09%)	252 / 6545 (3.85%)	86 / 1483 (5.80%)	145 / 1483 (9.78%)
AST2Seq Transformer (AT)	749 / 5835 (12.84%)	383 / 6545 (5.58%)	52 / 1483 (3.51%)	152 / 1483 (10.25%)
Parallel Random	780 / 5835 (13.68%)	315 / 6545 (4.81%)	67 / 1483 (4.52%)	146 / 1483 (9.84%)
Parallel Prior	809 / 5835 (13.86%)	353 / 6545 (5.39%)	76 / 1483 (5.12%)	149 / 1483 (10.05%)
Series Random	786 / 5835 (13.47%)	311 / 6545 (4.75%)	69 / 1483 (4.65%)	146 / 1483 (9.84%)
Series Prior	800 / 5835 (13.71%)	359 / 6545 (5.49%)	79 / 1483 (5.33%)	151 / 1483 (10.18%)
Parallel Unchanged Random	879 / 5838 (15.06%)	401 / 6545 (6.13%)	75 / 1483 (5.06%)	179 / 1483 (12.07%)
Parallel Unchanged Prior	901 / 5835 (15.44%)	413 / 6545 (6.31%)	85 / 1483 (5.73%)	180 / 1483 (12.14%)
Parallel Unchanged Order	947 / 5835 (16.23%)	438 / 6545 (6.69%)	90 / 1483 (6.07%)	184 / 1483 (12.41%)
Series Unchanged Random	869 / 5835 (14.89%)	395 / 6545 (6.04%)	72 / 1483 (4.86%)	177 / 1483 (11.94%)
Series Unchanged Prior	905 / 5835 (15.51%)	414 / 6545 (6.33%)	83 / 1483 (5.60%)	179 / 1483 (12.07%)
F$^3_{ST+AT}$	947 / 5835 (16.23%)	424 / 6545 (6.48%)	90 / 1483 (6.07%)	201 / 1483 (13.55%)
F$^3_{AT+ST}$	854 / 5835 (14.64%)	438 / 6545 (6.69%)	59 / 1483 (3.98%)	184 / 1483 (12.41%)

Table 2: The accuracy comparison among individual models, different connections methods and our F$^3_{ST+AT}$ and F$^3_{AT+ST}$ on BFP$_{small}$, BFP$_{medium}$, CodRep$_{real}$ and CodRep$_{abstract}$ datasets.

F$^3_{ST+AT}$, while with BFP$_{medium}$, AT is first, and the corresponding F^3 is F$^3_{AT+ST}$.

Across all four datasets, we can combine a F^3 framework, making it outperform any single-model baselines and multi-model baselines, which fully illustrates the performance advantage of F^3. Among them, the performance of Parallel Unchanged Order can do the same as F^3, the reason is that the two are similar for the use of unchanged fix, but in the subsequent experiments of RQ3 in Table 5, we can see that the cost of F^3 is smaller. Compared to the single-model (SR, ST, and AT), F^3 has a significant degree of improvement, but this is slightly lacking in the CodRep$_{real}$ dataset. This may be due to the fact that CodRep$_{real}$ has not undergone any abstraction process and its vocabulary is too huge, resulting in the unchanged fix not being obvious enough. This suggests that to fully utilize the F^3 framework, the vocabulary size needs to be controlled, as in many existing approaches (Chen et al., 2019), which is not the focus of this paper.

In addition, comparing baselines with and without the "Unchanged" field, such as Parallel Random and Parallel Unchanged Random, we can find that the introduction of the unchanged fix phenomenon can steadily improve the performance of the baseline. For example, in BFP$_{small}$, Series Prior has lower performance than ST, but the introduction of unchanged fix allows it to make better decisions compared to the single model. This also illustrates the enhancement of unchanged fix for the decision phase.

We also find that although there is a difference in performance of single models, for example, in BFP$_{small}$, ST has higher performance than AT, they combine as F$^3_{ST+AT}$ and are able to improve performance. It means that there still exist input cases where ST cannot fix but AT can fix though the overall performance of ST is better. As long as the two learners are not identical, they will possess the possibility to be joined as F^3 and improve the performance.

5.2 Impact of Learners' Order and Count (RQ2)

We compare the performance of F$^3_{AT+ST}$ and F$^3_{ST+AT}$ and we add the Seq2Seq RNN (SR) after the F$^3_{ST+AT}$ and F$^3_{AT+ST}$ to form F$^3_{ST+AT+SR}$ and F$^3_{AT+ST+SR}$.

The optimal filter mechanism should filter out all the wrong fixes by comparing them with the input buggy programs. To compare our filter mechanism and the optimal filter mechanism, we artificially select the wrong fixes of the learners by comparing them with the ground-truth fixed programs in the test set to simulate it. The dataset in this section is BFP$_{small}$.

Next, we count four sets, i.e., ***correct fixes*** $\mathcal{C}$, ***changed but wrong fixes*** $\mathcal{CW}$, ***unchanged fixes*** $\mathcal{U}$, and ***wrong fixes*** $\mathcal{W}$ defined above of the Seq2Seq Transformer and AST2Seq Transformer, to calculate the theoretical performance of these F^3 based on our equations above and compare it with the experimental results to validate our theory.

5.2.1 Results and Analysis

Our filter mechanism In Table 3, with our filter mechanism, the accuracy of F$^3_{ST+ST}$ is the same

Approach	Accuracy with two Filter Mechanism	
	Our Filter Mechanism	Optimal Filter Mechanism
SR	530 / 5835 (**9.08%**)	
ST	822 / 5835 (**14.09%**)	
AT	749 / 5835 (**12.84%**)	
F^3_{ST+AT}	947 / 5835 (**16.23%**)	1090 / 5835 (**18.68%**)
F^3_{AT+ST}	854 / 5835 (**14.64%**)	1090 / 5835 (**18.68%**)
$F^3_{ST+AT+SR}$	949 / 5835 (**16.26%**)	1097 / 5835 (**18.80%**)
$F^3_{AT+ST+SR}$	856 / 5835 (**14.67%**)	1097 / 5835 (**18.80%**)
F^3_{ST+ST}	822 / 5835 (**14.09%**)	822 / 5835 (**14.09%**)

Table 3: The accuracy with our filter mechanism and optimal filter mechanism of the Seq2Seq RNN (SR), Seq2Seq Transformer (ST), AST2Seq Transformer (AT) , F^3_{ST+AT} and the various F^3 transformed from it. The dataset of these results is BFP_{small}.

	$\mathcal{T}$	$\mathcal{C}$ (AT)	$\mathcal{W}$ (AT)	$\mathcal{CW}$ (AT)	$\mathcal{U}$ (AT)
$\mathcal{T}$	5,835	749	5,086	3,309	1,777
$\mathcal{C}$ (ST)	822	481	341	236	105
$\mathcal{W}$ (ST)	5,013	268	4,745	3,073	1,672
$\mathcal{CW}$ (ST)	2,510	143	2,367	1,894	473
$\mathcal{U}$ (ST)	2,503	125	2,378	1,179	1,199

Table 4: Counts of $\mathcal{C}$, $\mathcal{CW}$, $\mathcal{U}$ and $\mathcal{W}$ of Seq2Seq (ST) and AST2Seq (AT) Transformer on the test set $\mathcal{T}$.

as Seq2Seq Transformer, demonstrating that connecting identical models is not helpful.

The accuracy results of Seq2Seq Transformer, F^3_{ST+AT}, $F^3_{ST+AT+SR}$ are increasing. A similar pattern appears in AST2Seq Transformer, F^3_{AT+ST} and $F^3_{AT+ST+SR}$. It shows that adding new different learners to F^3 can improve the performance of F^3. Moreover, we can see that the accuracy improvement from Seq2Seq Transformer to F^3_{ST+AT} is greater than that from F^3_{ST+AT} to $F^3_{ST+AT+SR}$. This may be because both Seq2Seq RNN and Seq2Seq Transformer are based on token sequences and the bugs they can fix are similar. These results above can verify our theory that with our filter mechanism when adding a new learner to the original F^3, we can guarantee that the new F^3 will performs better than the original F^3. As to learner' order, we can see that F^3_{ST+AT} performs better than F^3_{AT+ST}, which shows that changing the order of learners may affect the performance of F^3 with our filter mechanism as we mentioned above.

Quantitatively, according to the counts of the four sets in Table 4, we can calculate theoretical number of F^3_{ST+AT} correct fixes with our filter mechanism based on Eq. 6 as:

$$|\mathcal{C}(F^3_{ST+AT})| = |\mathcal{C}(ST)| + |\mathcal{C}(AT) \cap \mathcal{U}(ST)| = 947 \tag{15}$$

and the theoretical accuracy is 16.23% , which

Approach	Accuracy	Cost	Second Learner Cost
Parallel Random	780 / 5835 (13.68%)		
Parallel Prior	809 / 5835 (13.86%)		
Parallel Unchanged Random	879 / 5838 (15.06%)	11670	5835
Parallel Unchanged Prior	901 / 5835 (15.44%)		
Parallel Unchanged Order	947 / 5835 (16.23%)		
Series Random	786 / 5835 (13.47%)	8750	2915
Series Prior	800 / 5835 (13.71%)	7293	1458
Series Unchanged Random	869 / 5835 (14.89%)	10004	4169
Series Unchanged Prior	905 / 5835 (15.51%)	9171	3336
F^3_{ST+AT}	947 / 5835 (16.23%)	8338	2503
F^3_{AT+ST}	854 / 5835 (14.64%)	7612	1777

Table 5: The cost of baselines and F^3 in BFP_{small}.

is consistent with experimental results. Similarly, it is also consistent for F^3_{AT+ST}. Therefore, the equations proposed above have been verified.

Optimal filter mechanism As we have proved, the learners' order does not have effect on F^3, and F^3 outperforms all the internal learners in Table 3.

Moreover, F^3 with the optimal filter mechanism all outperform these with our filter mechanism. However, improvements between the two filter mechanism are not really significant, because the performance of the learners is also an important bottleneck of F^3.

We can calculate the theoretical number of fixes corrected by F^3_{ST+AT} with the optimal filter mechanism based on Eq. 12 as:

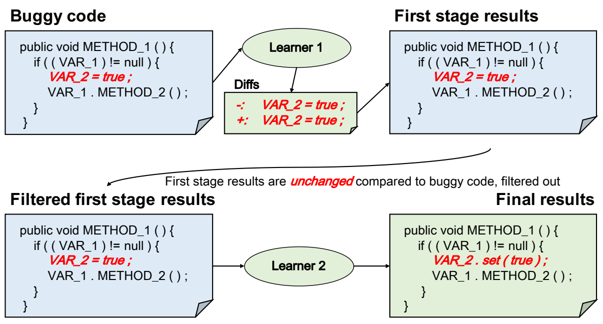
$$|\mathcal{C}(F^3_{ST+AT})| = |\mathcal{C}(ST)| + |\mathcal{C}(AT) \cap \mathcal{W}(ST)| = 1090 \tag{16}$$

and the theoretical accuracy is 18.68%, which is also identical to our experiment.

5.3 Cost Evaluation (RQ3)

In order to facilitate the cost statistics, we record the inference cost consumed by each input case into the Transformer (including ST and AT, both of which are transformers and have the same number of parameters, similar time consumption to process the same case) as one unit of cost. For example, if a buggy code, after passing ST, is filtered and then passes the second learner of F^3_{ST+AT}, AT, then it consumes 2 units of cost. According to this, we have recorded the amount of cost consumed by all connection methods in the BFP_{small} dataset, as shown in Table 5.

Overall, compared to other baselines, F^3 has the best performance and almost the lowest consumption. Series Prior has a lower cost than F_{ST+AT}, but at the cost of a much lower accuracy. The two


Buggy code
Learner 1
Diffs
First stage results
First stage results are *unchanged* compared to buggy code, filtered out
Filtered first stage results
Learner 2
Final results

Buggy code:

```
public void METHOD_1 ( ) {
  if ( ( VAR_1 ) != null ) {
    VAR_2 = true ;
    VAR_1 . METHOD_2 ( ) ;
  }
}
```

Diffs:

```
-:  VAR_2 = true ;
+:  VAR_2 = true ;
```

First stage results:

```
public void METHOD_1 ( ) {
  if ( ( VAR_1 ) != null ) {
    VAR_2 = true ;
    VAR_1 . METHOD_2 ( ) ;
  }
}
```

Filtered first stage results:

```
public void METHOD_1 ( ) {
  if ( ( VAR_1 ) != null ) {
    VAR_2 = true ;
    VAR_1 . METHOD_2 ( ) ;
  }
}
```

Final results:

```
public void METHOD_1 ( ) {
  if ( ( VAR_1 ) != null ) {
    VAR_2 . set ( true ) ;
    VAR_1 . METHOD_2 ( ) ;
  }
}
```

Figure 4: The case analysis. The first learner fails to fix the bug because its output diffs do not change the buggy code. This buggy code is filtered into the second learner which generates the whole code piece to fix it.

learners in all parallel methods accept all cases and so have the maximum cost. While in Series methods, the introduction of unchanged fix, which increases the accuracy, also leads to a part of the cost increase. This illustrates that the essence of unchanged fix is to reduce the randomness in the decision process by additional unsupervised trial and error, thus improving performance.

5.4 Generality Analysis (RQ4)

F^3 is a general framework that can combine a wide variety of bug fixing methods with different inputs and outputs. Figure 4 is a case for analysis. The first learner produces a predicted diff and we incorporate the diff into the original buggy code to get ***first stage results***, which is ***unchanged*** because it is the same as the buggy code, so our filter mechanism filters it out and passes it to the second learner to continue the fixing, and the second learner completes the fixing successfully.

Obviously, the two learners can essentially be replaced by most existing state-of-the-art methods because no matter how the existing model changes the input and output, its final fix still needs to be verified on the original buggy code, which inevitably can produce complete first stage results, thus allowing our filter mechanism works. F^3 may be suitable for many tasks that make changes to the original input, such as image denoising.

6 Related Work

We refer the reader to (Monperrus, 2018) for a comprehensive review of program repair. There are many bug fixing works (Jiang et al., 2018; Lutellier et al., 2020) recently. DeepFix (Gupta et al., 2017), TRACER (Ahmed et al., 2018), DeepDelta (Mesbah et al., 2019) and Graph2Diff (Tarlow et al., 2020) are the important works related to ours, which use machine learning or NMT for compiler program repair, while our work focuses on logical or semantic bugs. Furthermore, (Tufano et al., 2019) investigate the feasibility of NMT for bug fixing via Seq2Seq RNN model, it takes a buggy method token sequences as input and the fixed method token sequences as output, which is similar to our work. In contrast, we use AST as input to the Transformer model and focus on exploring the links between learners in the F^3 rather than single models. (Chen et al., 2019) propose SUQUENCER using a Seq2Seq model with attention and copy mechanism for bug fixing without locating. In contrast, our work includes bug locating and fixing. CODIT (Chakraborty et al., 2020) is a tree-based NMT system to model source code changes and learn code change patterns from the wild, it uses the AST to model code changes while we use it to model the buggy code.

In general, the focus of our work differs from all of the above in that our F^3 focuses on the connections between models. Our works are orthogonal to many of the above, and the F^3 can connect them to address more comprehensive tasks.

7 Conclusion

We reveal and study the unchanged fix issue in learning-based bug fixing tasks. Based on our findings, we propose an intuitive yet effective general framework called F^3 to concatenate different learners with the filter mechanism to filter out unchanged fixes. We demonstrate the considerable performance and generality of F^3 from both theoretical and experimental perspectives. In the future, we will the design better filter mechanism and apply F^3 to different learning tasks.

Acknowledgements

This work is supported by the NSFC projects (No. 62072399, No. U19B2042, No. 61402403), Chinese Knowledge Center for Engineering Sciences and Technology, MoE Engineering Research Center of Digital Library, National Engineering Research Center for Big Data Technology and System, the Fundamental Research Funds for the Central Universities, and partially supported by Australian Research Grant DP21010134.

References

Umair Z Ahmed, Pawan Kumar, Amey Karkare, Purushottam Kar, and Sumit Gulwani. 2018. Compilation error repair: for the student programs, from the student programs. In *Proceedings of the 40th International Conference on Software Engineering: Software Engineering Education and Training*, pages 78–87.

S. Chakraborty, Y. Ding, M. Allamanis, and B. Ray. 2020. Codit: Code editing with tree-based neural models. *IEEE Transactions on Software Engineering*, pages 1–1.

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

Zimin Chen and Martin Monperrus. 2018. The codrep machine learning on source code competition. *arXiv preprint arXiv:1807.03200*.

Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish Shevade. 2017. Deepfix: Fixing common c language errors by deep learning. In *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence*, pages 1345–1351.

Hideaki Hata, Osamu Mizuno, and Tohru Kikuno. 2012. Bug prediction based on fine-grained module histories. In *2012 34th international conference on software engineering (ICSE)*, pages 200–210. IEEE.

Jiajun Jiang, Yingfei Xiong, Hongyu Zhang, Qing Gao, and Xiangqun Chen. 2018. Shaping program repair space with existing patches and similar code. In *Proceedings of the 27th ACM SIGSOFT international symposium on software testing and analysis*, pages 298–309.

Daoyuan Li, Li Li, Dongsun Kim, Tegawendé F Bissyandé, David Lo, and Yves Le Traon. 2019. Watch out for this commit! a study of influential software changes. *Journal of Software: Evolution and Process*, 31(12):e2181.

Thibaud Lutellier, Hung Viet Pham, Lawrence Pang, Yitong Li, Moshi Wei, and Lin Tan. 2020. Coconut: Combining context-aware neural translation models using ensemble for program repair. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 101–114.

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*, pages 925–936.

Martin Monperrus. 2018. The living review on automated program repair. *HAL/archives-ouvertes. fr, Tech. Rep. hal-01956501*.

Martin Monperrus and Matias Martinez. 2012. Cvs-vintage: A dataset of 14 cvs repositories of java software.

Myle Ott, Sergey Edunov, Alexei Baevski, Angela Fan, Sam Gross, Nathan Ng, David Grangier, and Michael Auli. 2019. fairseq: A fast, extensible toolkit for sequence modeling. In *Proceedings of NAACL-HLT 2019: Demonstrations*.

Ingo Scholtes, Pavlin Mavrodiev, and Frank Schweitzer. 2016. From aristotle to ringelmann: a large-scale analysis of team productivity and coordination in open source software projects. *Empirical Software Engineering*, 21(2):642–683.

Daniel Tarlow, Subhodeep Moitra, Andrew Rice, Zimin Chen, Pierre-Antoine Manzagol, Charles Sutton, and Edward Aftandilian. 2020. Learning to fix build errors with graph2diff neural networks. In *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*, pages 19–20.

Michele Tufano, Gabriele Bavota, Denys Poshyvanyk, Massimiliano Di Penta, Rocco Oliveto, and Andrea De Lucia. 2017. An empirical study on developer-related factors characterizing fix-inducing commits. *Journal of Software: Evolution and Process*, 29(1):e1797.

Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. 2019. An empirical study on learning bug-fixing patches in the wild via neural machine translation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 28(4):1–29.

Hao Zhong and Zhendong Su. 2015. An empirical study on real bug fixes. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 913–923. IEEE.

Jian Zhou, Hongyu Zhang, and David Lo. 2012. Where should the bugs be fixed? more accurate information retrieval-based bug localization based on bug reports. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 14–24. IEEE.