Jointly Learning to Repair Code and Generate Commit Message

Jiaqi Bai[†‡*], **Long Zhou**[◇], **Ambrosio Blanco**[◇], **Shujie Liu**[◇],
Furu Wei[◇], **Ming Zhou**[◇], **Zhoujun Li**[†‡]
[†]School of Cyber Science and Technology, Beihang University, China
[‡]State Key Lab of Software Development Environment, Beihang University, China
[◇]Microsoft Research Asia
{bjq,lizj}@buaa.edu.cn
{lozhou,ambrosio.blanco,shujliu,fuwei,mingzhou}@microsoft.com

Abstract

We propose a novel task of jointly repairing program codes and generating commit messages. Code repair and commit message generation are two essential and related tasks for software development. However, existing work usually performs the two tasks independently. We construct a multilingual triple dataset including buggy code, fixed code, and commit messages for this novel task. We provide the cascaded models as baseline, which are enhanced with different training approaches, including the teacher-student method, the multi-task method, and the back-translation method. To deal with the error propagation problem of the cascaded method, the joint model is proposed that can both repair the code and generate the commit message in a unified framework. Experimental results show that the enhanced cascaded model with teacher-student method and multitask-learning method achieves the best score on different metrics of automated code repair, and the joint model behaves better than the cascaded model on commit message generation.

1 Introduction

Deep learning has been demonstrated remarkably adept at numerous natural language processing (NLP) tasks, such as machine translation (Bahdanau et al., 2014), relation extraction (Zhang et al., 2017), grammar error correction (Ge et al., 2018), and so on. The success of deep learning in NLP also promotes the development of which in programming languages (Clement et al., 2020; Lu et al., 2021). Recently, researchers have exploited deep learning to programming-language related tasks, such as code completion (Svyatkovskiy et al., 2020), automated code repair (Tufano et al., 2018), commit messages generation (Xu et al., 2019), code search (Gu et al., 2018), and so on. Among these tasks, automated code repair and commit message generation are the two most active and closely related tasks. The former is to repair software bugs automatically without the intervention of a human programmer. The latter aims to generate natural language descriptions of code changes, which act as a record of feature additions and bug repairs.

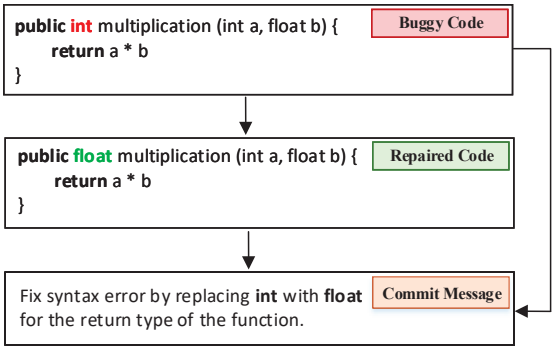


Figure 1: An illustrative example for our proposed task. Given a buggy code, the task is to generate its corresponding repaired version, as well as the commit message that describes their changes.

Because these two tasks can potentially reduce debugging costs in software development and helps programmers to understand the high-level rationale of changes, a lot of great work has been proposed to deal with automated program repair (Tufano et al., 2018; Chen et al., 2019; Dinella et al., 2020; Yasunaga and Liang, 2020; Tang et al., 2021) and commit message generation (Loyola et al., 2017; Liu et al., 2020; Nie et al., 2020), respectively. However, existing work tackles the two tasks independently, ignoring the underlying relationship between these two closely related tasks, e.g., after fixing the bug, commit message can record the process of code repair. Therefore it is crucial to explore how to bridge these two tasks and achieve the code repair and commit messages generation simultaneously.

In this paper, we formulate a novel task to jointly repair program code and generate commit message,

[*]Contribution during internship at Microsoft Research

Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, pages 9784–9795
November 7–11, 2021.

given the buggy code. To facilitate the study of this task, we create a dataset with multiple programming languages. The dataset is collected from commit and buggy-fixed histories of open-source software projects, where each example consists of buggy code, fixed code, and the corresponding commit message. We first introduce the cascaded methods as baseline. The cascaded model employs one model to repair code and the other to generate commit message successively. We enhance this cascaded model with three training approaches inspired by the low-resource machine translation, including the teacher-student method (Chen et al., 2017), the multi-task learning method (Domhan and Hieber, 2017), and the back-translation method (Sennrich et al., 2016a). To deal with the error propagation problem of the cascaded method, we propose a joint model which can achieve both code repair and commit message generation in a single model. We train and evaluate our model using the created triple (*buggy-fixed-commit*) dataset. The results demonstrate the validity of our proposed methods, which achieve a significant improvement over baseline in both qualities of code and commit messages. Particularly, the enhanced cascaded method obtains the best performance on code repair task, and the joint method behaves better than the cascaded method on commit message generation task.

Our main contributions are as follows:

- We propose the novel task of jointly repairing code and generating commit message. Moreover, we collect and release a multilingual *buggy-fixed-commit* dataset for the task.
- We perform an empirical study of different machine learning-based methods for code repair and commit message generation.
- To the best of our knowledge, this is the first work to investigate the effectiveness of joint modeling with the code repair process and commit message generation.

2 Task Definition

We aim to generate the repaired code and the commit message based on the given buggy code. For the example in Figure 1, our goal is to replace syntax bug "int" with "float" for the return type of the "multiplication" method, then generate a piece of commit message to describe the change.

Formally, given a triple (*buggy-fixed-commit*) dataset $\mathcal{D} = \{(B_i, F_i, C_i)\}_{i=1}^{K}$, where the i-th sample consists of a buggy code snippet B_i, its fixed version F_i, and a commit message C_i that is used to describe the changes from B_i to F_i, our goal is to learn probability distribution $P(C, F|B)$. In practice, the commit message C is hard to estimate without the full consideration of both B and F. Therefore, it is a reasonable way to firstly predict the fixed code F based on B. Then learn how to generate an appropriate message C according to B and F. The probability $P(C, F|B)$ can be decomposed as:

$$P(C, F|B) = \sum_{F} P(F|B)\, P(C|B, F) \tag{1}$$

Thus, given a new buggy code B, we can generate its fixed version F, and the commit message C following the conditional probability $P(F|B)$ and $P(C|B, F)$, respectively.

3 Approach

In this section, we firstly introduce the cascaded models enhanced by the teacher-student method, the multi-task learning method, and the back-translation method (Section 3.1), which generate repaired code and commit message in a two-stage manner. Then, we propose a joint model (Section 3.2), which is capable of jointly optimizing the generation of repaired code and commit message in an end-to-end manner. The models described in this section are all build on the Transformer model (Vaswani et al., 2017), where we devise the model to take in some representation of input and then yield a distribution over output vocabulary.

3.1 Cascaded Model

Cascaded model is one of the most straight-forward methods to tackle this problem, where F is used as a hinge to build bridges between B and C. Formally, given the buggy code B, the generation of commit message C can be conducted in two steps. The first step aims to generate F conditioned on B, which can be defined by minimizing the following negative log-likelihood loss:

$$\mathcal{L}_{\mathcal{F}}(\theta) = -\sum_{(B,F)\in\mathcal{D}} \log P(F|B) \tag{2}$$

The second step is to generate commit message C based on B and previous generated F, and it can be formally expressed as:

$$\mathcal{L}_{\mathcal{C}}(\theta) = -\sum_{(B,F,C)\in\mathcal{D}} \log P(C|g(B, F)) \tag{3}$$

where $g(B, F)$ is a function to combine B and F as model input, which could be concatenating them, or using their changing information[1]. In the following section, the training loss of commit message generation is optimized by Equation (3), unless explicitly specified.

To further enhance the modeling capabilities of the cascaded model, we introduce three alternative methods by incorporating the teacher-student framework, multi-task learning method, and back-translation method, respectively.

Teacher-student Method We attempt to improve the performance of code repair with the help of commit message. Different from the previous works that directly used comments (Guo et al., 2020) or compiler error messages (Yasunaga and Liang, 2020) as the prior information, we utilize the commit message as the posterior information, to supervise the generation of F in code repair. Specifically, the teacher-student framework (Hinton et al., 2015) is employed to distill knowledge from teacher model to student model, which first learns a teacher model $P(F|B, C)$ with the use of C, where C is the truth commit message . Then, the teacher model teaches the student model $P(F|B)$ by minimizing the KL divergence (Kullback and Leibler, 2006), which is defined by

$$\mathcal{L}_{KL}(\theta) = \sum_{(B,F,C)\in\mathcal{D}} Q(F|B, C) \cdot \log \frac{Q(F|B,C)}{P(F|B)} \quad (4)$$

where $Q(F|B, C)$ represents the teacher's sequence distribution over the sample space of all possible sequences. When optimizing $\mathcal{L}_{KL}$, the posterior distribution $Q(F|B, C)$ can be regarded as labels, so that our model is instructed to use prior distribution $P(F|B)$ to approximate $Q(F|B, C)$ accurately. During the training stage of code repair, the student model not only learns from the output probabilities of teacher model, but also learns from the correct context, which is formulated by

$$\mathcal{L}^{T}_{\mathcal{F}}(\theta) = \mathcal{L}_{\mathcal{F}}(\theta) + \mathcal{L}_{KL}(\theta) \quad (5)$$

Multi-task Learning Inspired by previous work which shows that given the buggy lines can significantly improve the performance of code repair (Chen et al., 2019; Wen et al., 2018; Saha et al., 2017), we use an alternative way to improve code repair, which is the multi-task learning method. Specifically, we introduce a line-level binary sequence classification task as an auxiliary learning task to assist code repair, which reduces the difficulties for the model to locate the buggy lines[2]. To help the model distinguish from the line-level information and the token-level information, we add the "*[CLS]*" token at the beginning of each line of buggy code B, which is used to align with the tagging label T, where $T \in \{0, 1\}$, in which tag 0 means the line is error-free, and tag 1 means the line is buggy. To identify the buggy lines, we build a sequence classifier based on encoder output to implement the line-level binary sequence tagging task. The line-level sequence classification loss can be defined as:

$$\mathcal{L}_{\mathcal{T}}(\theta) = -\sum_{B\in\mathcal{D};T\in\{0,1\}} \log P(T|B) \quad (6)$$

At the stage of code repair, we jointly optimize the objective of sequence classification task and sequence generation task, i.e.,

$$\mathcal{L}^{M}_{\mathcal{F}}(\theta) = \mathcal{L}_{\mathcal{F}}(\theta) + \mathcal{L}_{\mathcal{T}}(\theta) \quad (7)$$

Back-translation Method Back translation has been demonstrated as an effective way on data augmentation (Sennrich et al., 2016a; Lachaux et al., 2020), and it leverages monolingual data to expand as pseudo-parallel data in a weakly-supervised manner. More precisely, we first train a back-directional model, that is a repaired code to buggy code model parameterized by $P(B|F, \theta_{F\to B})$. Then, the pseudo-parallel data is created by the back-directional model, in which the repaired code is regarded as the model input, and the goal is to predict its corresponding buggy version, which is formulated by

$$\hat{B} = \underset{B}{\operatorname{argmax}} P(B|F, \theta_{F\to B}) \quad (8)$$

where $\theta_{F\to B}$ is the parameter learned by maximum likelihood estimation on $\mathcal{M}$. $\mathcal{M}$ is a non-parallel corpus of fixed code, which is used to build the pseudo-parallel data. After obtaining $\hat{B}$, the pseudo parallel data $\mathcal{P} = \{(\hat{B}, F)\}$ is created to merge with the parallel data $\mathcal{D}$ to obtain the augmented parallel data $\mathcal{D}'$, which is used to train the code repair model according to Equation 2.

[1]We use `difflib` to represent code changes. The tool can be found in https://docs.python.org/3/library/difflib.html. In this paper, we use the code changes to build the model input, instead of their concatenation, since the latter will result in overlong sequence length, which drops the performance of model by a significant margin.

[2]To obtain the buggy lines, we employ the `difflib` to extract the line-level changes from buggy code to its fixed version. We maintain the lines only exist in the buggy version (i.e., remove the lines started with "+" and "?").

3.2 Joint Model

Although the above three methods can boost the performance of cascaded method, they still suffer from three challenges: (1) the generated fixed code may contain errors, and those errors will be propagated to the next step of commit generation, (2) they lose the inter-dependencies among global features to represent the changing details of code repair during commit message generation, and (3) the two-stage method results in low decoding efficiency. These problems may lead to the poor performance of commit generation. To this end, we propose a joint method that incorporates with a novel changes-aware dynamic attention mechanism to jointly decode fixed code and commit message.

Model Architecture The overview of our model is shown in Figure 2. Our model consists of three components: a buggy encoder, a fixed decoder, a message decoder with a changes-aware dynamic attention module. At first, the buggy encoder is deployed to encode the buggy code B, and map it into a sequence of output $\mathbf{z_b}$, where $\mathbf{z_b} \in \mathbb{R}^{n \times H}$, n and H are the length of B and the hidden size of model, respectively. $\mathbf{z_b}$ is used for line-level binary sequence tagging (optimized as Equation 6) and as an indispensable component to produce changes information. Then, the fixed decoder generates a high-level representation $\mathbf{z_f}$, $\mathbf{z_f} \in \mathbb{R}^{m \times H}$ is used to generate a repaired code $\hat{F}$, and produce changes information with $\mathbf{z_b}$. After that, the commit decoder that combines with the changes-aware dynamic attention mechanism generates an output representation $\mathbf{z_c}$, $\mathbf{z_c} \in \mathbb{R}^{l \times H}$ is used to attend over each representation of $\mathbf{z_b}$ and $\mathbf{z_f}$, then get a final output distribution to generate messages. In the following part, we will introduce our proposed changes-aware dynamic attention mechanism, as well as the method to jointly train our model.

Changes-aware Dynamic Attention During decoding the message, the output $\mathbf{z_c}$ generated by the message decoder respectively attends over $\mathbf{z_b}$ and $\mathbf{z_f}$ to obtain the context representation $\mathbf{c_b}$ and $\mathbf{c_f}$ by dot-product attention. which is formulated by

$$\mathbf{c}_\phi = softmax\left(\frac{\mathbf{z}_c \mathbf{z}_\phi^T}{\sqrt{H}}\right)\mathbf{z}_\phi \tag{9}$$

where $\phi \in \{\mathbf{b}, \mathbf{f}\}$. Similar as Vaswani et al. (2017), we use the scaling factor $\sqrt{H}$ to make gradient more stable during optimization. Intuitively, the context vector could provide much information to dynamically indicate the alignments of changes over the attended features during decoding the commit messages.

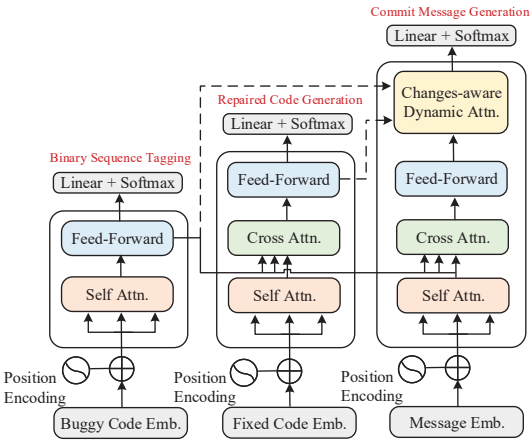


Figure 2: The architecture of the proposed joint model, where residual connection and layer normalization are omitted for simplification.

We subtract $\mathbf{c_b}$ from $\mathbf{c_f}$ in order to represent the semantic changes that took place from buggy code to its fixed version, and plus $\mathbf{c_b}$ with $\mathbf{c_f}$ to denote the semantic summarization of them, which is defined by

$$\begin{aligned} \delta &= \mathbf{c_b} - \mathbf{c_f} \\ \zeta &= \mathbf{c_b} + \mathbf{c_f} \end{aligned} \tag{10}$$

here, the δ and ζ are $\mathbb{R}^{l \times H}$ matrices, which represent the changes context and summarization context, respectively. Intuitively, the δ is necessary to generate informative commit messages since it describes the changes from buggy code to its fixed version. Nevertheless, the summarization ζ is also indispensable during decoding, since it always contain vital information to generate the meaningful tokens (e.g., function name or class name), which may not be included in δ.

We develop the control gates to balance the contribution between δ and ζ during decoding the message. The control gates control the degree that attending over each feature of δ and ζ, which is defined as

$$\begin{bmatrix} g_\delta \\ g_\zeta \end{bmatrix} = \sigma\left(\mathbf{W}_g\left[\delta; \zeta\right]^T + \mathbf{b}_g\right) \tag{11}$$

where $[a; b]$ denotes the concatenation between a and b. $\mathbf{W}_g \in \mathbb{R}^{2H \times 2H}$ and $\mathbf{b}_g \in \mathbb{R}^{2H \times l}$ are learnable parameters. The gates $g_\delta \in \mathbb{R}^{H \times l}$ and $g_\zeta \in \mathbb{R}^{H \times l}$ are used to control δ and ζ, respectively.

The final output of changes-aware dynamic attention module is the linear combination between

the output state of commit message decoder and the gated fusion of context representations, which can be calculated as:

$$\mathbf{o_c} = \mathbf{z_c} + \mathbf{W_o}\left(g_\delta \odot \delta^T + g_\zeta \odot \zeta^T\right) \quad (12)$$

where $\mathbf{W_o} \in \mathbb{R}^{H\times H}$ is the learnable weights. $\odot$ denotes the element-wise product.

Joint Training We jointly train our model in an end-to-end manner, the overall loss is defined as

$$\mathcal{L}_\mathcal{J}(\theta) = \mathcal{L}_\mathcal{R}(\theta) + \mathcal{L}_\mathcal{C}(\theta) + \mathcal{L}_\mathcal{T}(\theta) \quad (13)$$

where $\mathcal{L}_\mathcal{R}(\theta)$, $\mathcal{L}_\mathcal{C}(\theta)$and $\mathcal{L}_\mathcal{T}(\theta)$ are used to optimize the repaired code generation, commit message generation, and binary sequence classification, respectively. When training multilingual model of fixing code and predicting commit message, following multilingual neural machine translation (Johnson et al., 2017), we mix the training corpus and add a special token (e.g., <java>) at the beginning of each input sequence to distinguish from different programming languages.

4 Data

In this section, we describe the creation of the dataset in detail. We first describe how we collect the data in the wild. Then, we introduce the preparation process of the data to make it suitable for our tasks.

Data Collection We collected data from GitHub Archive[3] using the GH Archive API. We first download the event data in each public event from 2018 to 2020, and then filter them using the Github API[4] to obtain meta information of each commit. Specifically, we maintain the commits that consist of edits to the files with multiple programming languages. (i.e., Java, Javascript, Python, C sharp, Cpp). Moreover, to ensure that the prior file and post file are repairing code, we follow (Fischer et al., 2003), where the commits without the specific patterns (i.e., "fix" or "solve") in its commit messages are filter out. After we obtain the meta information of the filtered commit, we begin downloading the buggy file (i.e., the file prior to the commit) and fixed file (i.e., the file following the commit) in pair. Apart from the above multilingual dataset, we also build a Java-only monolingual triple dataset from the corpus (buggy-fixed pair) released by Tufano et al. (2018)[5].

	Languages	Train	Valid	Test	Total
Multi.	Python	36682	4585	4586	45853
	Java	11129	1391	1392	13912
	Javascript	21446	2680	2681	26807
	C-sharp	5424	678	678	6780
	Cpp	8510	1063	1064	10637
Mono.	Java	47775	3000	3000	53775

Table 1: Data statistic of the multilingual and the monolingual dataset.

Data Preparation The commit messages are filtered by (i) removing the messages whose length shorter than 3 words or longer than 100 words; (ii) filtering the `url` in commit messages; (iii) removing the messages that appear more than 3 times in the dataset. The rationale behind the latter decision was to remove the data with meaningless commit messages (e.g., "fix bug.", "fix an error.", etc.). For the processing of file-level buggy code and fixed code, we follow (Tufano et al., 2018), where both the buggy code and fixed code are separated into method-level fragments since the file-level granularity is too large to learn patterns of transformation. After preparation, we obtain the clean triples consist of buggy code, fixed code, and commit message. The statistics of the dataset used in this paper are summarized in Table 1. More processing details and statistics can be found in Appendix A and Appendix B. We release the datasets at https://github.com/jqbemnlp/BFCsData.

5 Experiments

5.1 Experimental Settings

Evaluation Metrics We conduct evaluations on both code repair and commit message generation. For the code repair, we use exact match accuracy (Chen et al., 2018) to measure the percentage of the predicted fixed code that are exactly matching the truth fixed code. In addition, we also introduce the BLEU-4 score (Papineni et al., 2002) as a supplementary metric to evaluate their partial match. For the commit message generation, we use BLEU-4 and Rouge-L (Lin, 2004) to evaluate our model.

[3] https://www.githubarchive.org

[4] https://docs.github.com/en/free-pro-team@latest/rest

[5] https://sites.google.com/view/learning-fixes/data

Models	Automated Code Repair		Commit Message Generation	
	BLEU-4	xMatch	BLEU-4	ROUGE-L
Naive Method	87.45	0.00	8.40	7.98
Oracle Method	-	-	12.64	11.59
Cascaded Model	85.07	3.21	9.69	9.41
+ Teacher-student	**88.23**	6.16	10.58	10.19
+ Multitask	87.94	**8.33**	10.36	10.1
+ Back-translation	87.73	5.26	10.19	9.84
Joint Model	87.61	8.01	**11.48***	**10.62***

Table 2: Results of the cascaded model and the proposed joint model on the monolingual dataset for code repair and commit message generation tasks. The bold face indicates the best result under the corresponding metric. Significant improvements over the best baseline results are marked with * (t-test, p<0.05).

Implementation Details All models are implemented using Pytorch framework[6], trained on four GPUs of NVIDIA Tesla V100. We use Byte Pair Encoding (BPE)[7] (Sennrich et al., 2016b) to encode input using a shared vocabulary with 50K symbols. The Transformer structure and the hyperparameters are following the default setting in the open-source implementation of XLM[8] (Lample and Conneau, 2019), apart from the embedding dimension and maximum sequence length, which are set as 256 and 512, respectively. More training details can be found in Appendix C.

5.2 Results on Monolingual Dataset

We first compare the performance of our proposed cascaded model and joint model for code repair and commit message generation tasks on the monolingual dataset, as listed in Table 2.

Automated Code Repair The teacher-student model achieves the highest score on the metric BLEU-4, which indicates that the commit message could provide effective guidance for the model to repair code. Moreover, it also indicates that the teacher-student method successfully distills the knowledge from the teacher model to the student model, without much loss of accuracy[9]. The multitask learning model outperforms other experimental models on metric exact match, and get the comparable performance of the teacher-student model on BLEU-4. The intuition behind that is the model has learned the location information of buggy line from the supervision of line-level binary sequence classification task, which could provide potential guidance for the model to correctly repair code. It is worth noting that the Naive method, which directly uses the buggy code to compare with its repaired version, also gets the distinct score on metric BLEU-4, which indicates the high overlap ratio between the buggy-fixed pairs.

Commit Message Generation Both the joint model and cascaded model are superior to generate meaningful commit message than the naive method, which directly uses buggy code to realize message generation. The joint model outperforms cascaded models over all evaluation metrics of commit message generation. Specifically, it achieves about 10.8% and 5.1% improvements respectively on BLEU-4 and ROUGE-L compared to the multitask learning model, which is one of the most competitive models on code repair. It is highly likely that the joint model effectively captures the inter-dependencies among global features to represent the changing details of code repair during commit message generation, thereof mitigates the error propagation of the two-stage model.

5.3 Results on Multilingual Dataset

We further analyze the results of the joint model on the multilingual dataset for both code repair and commit message generation. Table 3 shows the results on five program languages.

With regard to code repair, the result shows that the multilingual model achieves significant improvements compared to the monolingual model in terms of java, cpp and c-sharp dataset, and obtains comparable performance on python and javascript dataset, whether using BLEU-4 or exact match as evaluation metric. The intuition behind that is

[6] An open-source deep learning platform (https://pytorch.org/)

[7] The BPE codes are learned by fastBPE (https://github.com/glample/fastBPE).

[8] https://github.com/facebookresearch/XLM

[9] We have evaluated the performance of both the teacher model and student model, the results show that the student model obtains the comparable performance with teacher model.

Langs.	Automated Code Repair						Commit Message Generation					
	BLEU-4			xMatch			BLEU-4			ROUGE-L		
	mono.	multi.	Δ	mono.	multi.	Δ	mono.	multi.	Δ	mono.	multi.	Δ
python	95.21	94.99	-0.22	8.32	8.01	-0.31	13.29	14.01	0.72	12.83	13.46	0.63
javascript	94.89	95.21	0.32	6.78	7.42	0.64	11.03	11.63	0.60	10.79	11.30	0.51
java	95.72	96.74	1.02	6.33	7.82	1.49	12.26	13.79	1.53	11.72	12.73	1.01
cpp	94.10	95.45	1.35	5.63	7.34	1.71	9.71	11.04	1.33	8.63	9.84	1.21
c-sharp	93.26	95.34	2.08	3.98	6.92	2.94	8.13	10.98	2.85	7.19	9.93	2.74

Table 3: Results on the multilingual dataset for both code repair and commit message generation.

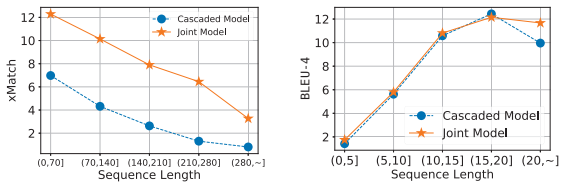
the corpus mixed with the multiple programming languages is helpful to make up for the lack of monolingual data during repairing code. In other words, the model could learn the potential bug-fixing patterns from multiple languages, and apply them to the limited monolingual data to handle the deficiency of data-limitation problem. A similar observation can also be found during generating commit messages. As shown in Table 3, for commit message generation task, the multilingual model outperforms monolingual model over all evaluation metrics and languages. We believe that the alignment of embedding spaces across multiple programming languages, shares either the same alphabet or anchor tokens such as variable, digits, string, method name, etc., which allows the model to learn these alignments simultaneously during generating commit messages.

5.4 Discussion

Ablation Study To further study the effects brought by different techniques, we show in Table 4 the result of different joint model variants on the monolingual dataset. First, we remove the line-level sequence classification task from our joint model. Therefore, the model is optimized without the loss function $\mathcal{L}_{\mathcal{T}}(\theta)$ that is mentioned in Equation 13. We observe that the results of both code repair and commit message generation decrease distinctly, which shows that locating the buggy lines is important for repairing code and generating commit messages. Then, we remove the proposed changes-aware dynamic attention module. It can be seen that this modification doesn't impact too much for the code repair, but affect the performance of commit message generation by a large margin. The main reason is that the changes-aware dynamic attention module could effectively model the changes from buggy code to its fixed version, thereby improves the performance of commit message generation.

Models	Code Repair		Message Generation	
	BLEU-4	xMatch	BLEU-4	ROUGE-L
Joint Model	87.61	**8.01**	**11.48**	**10.62**
- Binary Tagging	85.64	3.98	10.10	9.25
- Changes-aware Attn	**87.88**	7.94	9.03	8.67

Table 4: Ablation Study for joint model on monolingual dataset. "-" means remove the corresponding part separately.



(a) Code Repair (b) Message Generation

Figure 3: Length studies for both code repair and commit message generation.

Lengths Analysis We further analyze the model performance on different sequence length, and conduct a comprehensive study for both code repair and commit message generation on the monolingual dataset. Figure 3(a) and 3(b) present the results of code repair and commit message generation, respectively. Figure 3(a) demonstrates the challenge of this task, especially when the repaired code with a long sequence length. It can be seen that even the exact match score of the two models declined with the growing length of repaired code, the joint model still outperforms the cascaded model over all length ranges, which demonstrates the stronger capability of the joint model on modeling the code repair. Figure 3(b) presents the comparative results of the cascaded model and joint model on generating commit message. We observe that the joint model outperforms the cascaded model when the sequence length exceeds 20, which demonstrates the superiority of the joint model to excavate the underlying semantic relationships between code changes and their corresponding commit message during handling with the long message generation.

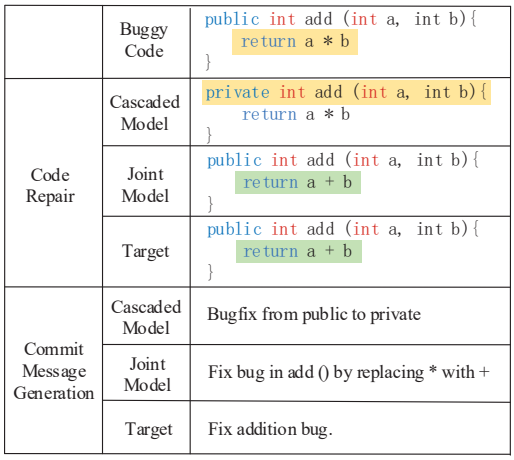


Figure 4: Examples on the monolingual dataset. The repaired lines are highlighted with different colors, where yellow means the buggy code is wrongly repaired, while green means it was correctly repaired.

Case Study We conduct case study on both monolingual and multilingual dataset. Figure 4 presents the results on monolingual datasets. With regard to the code repair part, the cascaded model wrongly modify the code by replacing `public` with `private` in the first line of buggy code, which indicates that it is hard for the model to locate and repair the buggy code without giving any prior information. It is worth noting that the joint model correctly repairs code, we believe that the line-level binary sequence classification task assists the model in locating the buggy lines, thereby improving the model's performance during repair the code. As for commit message generation, the joint model successfully captures the changes from buggy code to its fixed version and generates an appropriate message, while the cascaded model fails may due to the error propagation. More examples on multilingual dataset are shown in Appendix D.

6 Related Work

Our work is enlightened from two research lines of studies, which are automated code repair and commit message generation. We discuss these topics in the following.

Automated Code Repair Conventional approaches mainly focused on a relatively limited and manually-craft set of fixing patterns, which can only fix bugs in a given language or a specific application domain (Saha et al., 2017; Jin et al., 2011; Nguyen et al., 2013). Very recently, deep learning based approaches are proposed to automatically repair code by learning from massive open-source projects with numerous buggy-fixes pairs (Tufano et al., 2018; Chen et al., 2019; Vasic et al., 2019; Yasunaga and Liang, 2020). Tufano et al. (2018) first proposed using end-to-end neural machine translation model for learning bug-fixing patches. Besides, Guo et al. (2020) demonstrated that appropriately incorporating the natural language descriptions into the pre-train model could further improve the performance of code repair.

Commit Message Generation Early work on automatic commit message generation translates source code changes (such as feature additions and bug repairs) into natural language based on predefined rules and templates (Buse and Weimer, 2010; Cortés-Coy et al., 2014). To overcome the limitation of high complexity and difficult extensibility, some researchers employ information retrieval methods to generate commit messages, which attempts to re-use the commit messages of similar code changes (Huang et al., 2017). Recent work has focused on adopting machine learning based techniques for the commit message generation problem, which usually train a sequence-to-sequence model to translate the source changes into commit messages (Jiang et al., 2017; Loyola et al., 2017; Xu et al., 2019).

Although automated code repair and commit message generation have achieved rapid development in recent years, existing work usually regards them as two separate tasks and ignores the potential relationship between them. Different from previous work, we attempt to bridge the two tasks since commit message can be used to record the process of code repair. Specifically, we propose a novel task to repair code and generate commit message simultaneously with the proposed cascaded and joint methods, based on our collected *buggy-fixed-commit* dataset.

7 Conclusion

In this paper, we propose a novel task to jointly repair code and generate commit message. We provide several competitive architectures, including cascaded model and joint model. To train and evaluate our models, we collect a multilingual *buggy-fixed-commit* dataset from Github. The empirical study is conducted to demonstrate the effectiveness of our proposed methods. For future work, we plan to incorporate the tree structure of code into the task and employ more indicative metrics (Ren et al., 2020) to evaluate the model performance.

8 Acknowledgement

This work was supported in part by the National Natural Science Foundation of China (Grant Nos.U1636211, 61672081,61370126), the 2020 Tencent Wechat Rhino-Bird Focused Research Program, and the Fund of the State Key Laboratory of Software Development Environment (Grant No. SKLSDE-2021ZX-18).

References

Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. 2014. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*.

Raymond PL Buse and Westley R Weimer. 2010. Automatically documenting program changes. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, pages 33–42.

Xinyun Chen, Chang Liu, and Dawn Song. 2018. Tree-to-tree neural networks for program translation. *Advances in neural information processing systems*, 31:2547–2557.

Yun Chen, Yang Liu, Yong Cheng, and Victor OK Li. 2017. A teacher-student framework for zero-resource neural machine translation. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1925–1935.

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*.

Colin B. Clement, Dawn Drain, Jonathan Timcheck, Alexey Svyatkovskiy, and Neel Sundaresan. 2020. Pymt5: multi-mode translation of natural language and python code with transformers. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing, EMNLP 2020, Online, November 16-20, 2020*, pages 9052–9065. Association for Computational Linguistics.

Luis Fernando Cortés-Coy, Mario Linares-Vásquez, Jairo Aponte, and Denys Poshyvanyk. 2014. On automatically generating commit messages via summarization of source code changes. In *2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation*, pages 275–284. IEEE.

Elizabeth Dinella, Hanjun Dai, Ziyang Li, Mayur Naik, Le Song, and Ke Wang. 2020. Hoppity: Learning graph transformations to detect and fix bugs in programs. In *International Conference on Learning Representations (ICLR)*.

Tobias Domhan and Felix Hieber. 2017. Using target-side monolingual data for neural machine translation through multi-task learning. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*, pages 1500–1505.

Michael Fischer, Martin Pinzger, and Harald Gall. 2003. Populating a release history database from version control and bug tracking systems. In *International Conference on Software Maintenance, 2003. ICSM 2003. Proceedings.*, pages 23–32. IEEE.

Tao Ge, Furu Wei, and Ming Zhou. 2018. Reaching human-level performance in automatic grammatical error correction: An empirical study. *arXiv preprint arXiv:1807.01270*.

Xiaodong Gu, Hongyu Zhang, and Sunghun Kim. 2018. Deep code search. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pages 933–944. IEEE.

Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Jian Yin, Daxin Jiang, et al. 2020. Graphcodebert: Pre-training code representations with data flow. *arXiv preprint arXiv:2009.08366*.

Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. 2015. Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531*.

Yuan Huang, Qiaoyang Zheng, Xiangping Chen, Yingfei Xiong, Zhiyong Liu, and Xiaonan Luo. 2017. Mining version control system for automatically generating commit comment. In *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 414–423. IEEE.

Siyuan Jiang, Ameer Armaly, and Collin McMillan. 2017. Automatically generating commit messages from diffs using neural machine translation. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 135–146. IEEE.

Guoliang Jin, Linhai Song, Wei Zhang, Shan Lu, and Ben Liblit. 2011. Automated atomicity-violation fixing. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, pages 389–400.

M. Johnson, Mike Schuster, Quoc V. Le, M. Krikun, Y. Wu, Z. Chen, Nikhil Thorat, F. Viégas, M. Wattenberg, G. S. Corrado, Macduff Hughes, and J. Dean. 2017. Google's multilingual neural machine translation system: Enabling zero-shot translation. *Transactions of the Association for Computational Linguistics*, 5:339–351.

S Kullback and R Leibler. 2006. On information and sufficiency. the annals of mathematical statistics. *The Annals of Mathematical Statistics*, pages 79–86.

Marie-Anne Lachaux, Baptiste Roziere, Lowik Chanussot, and Guillaume Lample. 2020. Unsupervised translation of programming languages. *arXiv preprint arXiv:2006.03511*.

Guillaume Lample and Alexis Conneau. 2019. Cross-lingual language model pretraining. *arXiv preprint arXiv:1901.07291*.

Chin-Yew Lin. 2004. Rouge: A package for automatic evaluation of summaries. In *Text summarization branches out*, pages 74–81.

Shangqing Liu, Cuiyun Gao, Sen Chen, Nie Lun Yiu, and Yang Liu. 2020. Atom: Commit message generation based on abstract syntax tree and hybrid ranking. *IEEE Transactions on Software Engineering*.

Pablo Loyola, Edison Marrese-Taylor, and Yutaka Matsuo. 2017. A neural architecture for generating natural language descriptions from source code changes. *arXiv preprint arXiv:1704.04856*.

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.

Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. 2013. Semfix: Program repair via semantic analysis. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 772–781. IEEE.

Lun Yiu Nie, Cuiyun Gao, Zhicong Zhong, Wai Lam, Yang Liu, and Zenglin Xu. 2020. Contextualized code representation learning for commit message generation. *arXiv preprint arXiv:2007.06934*.

Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, pages 311–318.

Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. 2020. Codebleu: a method for automatic evaluation of code synthesis. *arXiv preprint arXiv:2009.10297*.

Ripon K Saha, Yingjun Lyu, Hiroaki Yoshida, and Mukul R Prasad. 2017. Elixir: Effective object-oriented program repair. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 648–659. IEEE.

Rico Sennrich, Barry Haddow, and Alexandra Birch. 2016a. Edinburgh neural machine translation systems for wmt 16. *arXiv preprint arXiv:1606.02891*.

Rico Sennrich, Barry Haddow, and Alexandra Birch. 2016b. 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.

Alexey Svyatkovskiy, Shao Kun Deng, Shengyu Fu, and Neel Sundaresan. 2020. Intellicode compose: Code generation using transformer. *arXiv preprint arXiv:2005.08025*.

Yu Tang, Long Zhou, Ambrosio Blanco, Shujie Liu, Furu Wei, Ming Zhou, and Muyun Yang. 2021. Grammar-based patches generation for automated program repair. In *Findings of the Association for Computational Linguistics: ACL/IJCNLP 2021, Online Event, August 1-6, 2021*, pages 1300–1305. Association for Computational Linguistics.

Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. 2018. An empirical investigation into learning bug-fixing patches in the wild via neural machine translation. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 832–837.

Marko Vasic, Aditya Kanade, Petros Maniatis, David Bieber, and Rishabh Singh. 2019. Neural program repair by jointly learning to localize and repair. *arXiv preprint arXiv:1904.01720*.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008.

Ming Wen, Junjie Chen, Rongxin Wu, Dan Hao, and Shing-Chi Cheung. 2018. Context-aware patch generation for better automated program repair. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pages 1–11. IEEE.

Shengbin Xu, Yuan Yao, Feng Xu, Tianxiao Gu, Hanghang Tong, and Jian Lu. 2019. Commit message generation for source code changes. In *IJCAI*, pages 3975–3981.

Michihiro Yasunaga and Percy Liang. 2020. Graph-based, self-supervised program repair from diagnostic feedback. *arXiv preprint arXiv:2005.10636*.

Meishan Zhang, Yue Zhang, and Guohong Fu. 2017. End-to-end neural relation extraction with global optimization. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*, pages 1730–1740.

A Data Processing Details

To ensure the commit message describes the changes information that took place from buggy method to its fixed version, we only consider the changes that are inside of a single method in the file. Changes that involve multiple methods are not considered in our work, since it is implicit to indicate which method does the commit message describe to. Besides, we also develop a heuristics method in which the lexical overlap is employed to filter the commits that the commit message doesn't describe the changes from the buggy method to its fixed version. Specifically, we first tokenize the commit message and the code by `nltk`[10] and `pygments`[11], respectively. Then, we only maintain the commit in which at least one of the tokens in the commit message matches a code token belonging to the buggy code or repaired code[12]. In order to check whether the message describes the changes from buggy code to its fixed version, we randomly selected 100 samples and employ two well-educated annotators for independently analyzing the identified commits. After solving 4 cases of disagreement, they concluded that 97% of the identified commits were true positive.

B Data Statistics

Data Statistics	Multi.					Mono.
	py	js	java	cpp	c-sharp	
avg. # tokens per buggy	144.7	149.8	135.0	153.5	140.7	166.9
avg. # LOC per buggy	16.9	19.7	15.7	17.8	18.5	12.6
avg. # tokens per commit	12.0	9.5	12.6	15.6	10.6	13.8

Table 5: Overview of the multilingual and monolingual datasets. "LOC" denotes the physical lines of code.

We take the further analysis for the monolingual and multilingual datasets. Table 5 summarizes the average number of tokens per buggy, the average line of code per buggy, and the average number of tokens per commit. We observe that the average number of lines and tokens for buggy code are considerable, which indicates the difficulty of this task. Figure 5 and Figure 6 present the distribution of the amount of token for buggy code and commit message, respectively. We observe that the density

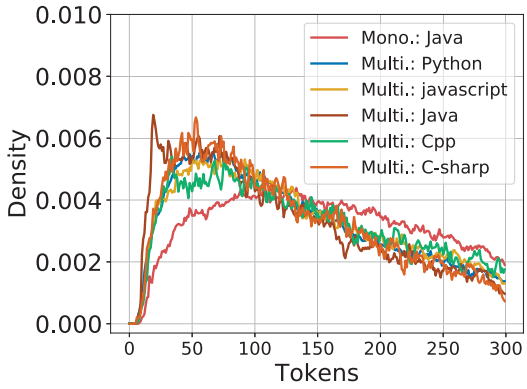


Figure 5: The smoothed distribution of buggy code in terms of their size. The x-axis denotes the number of tokens for the buggy code. The y-axis indicates the density of the buggy code with the corresponding amount of tokens.

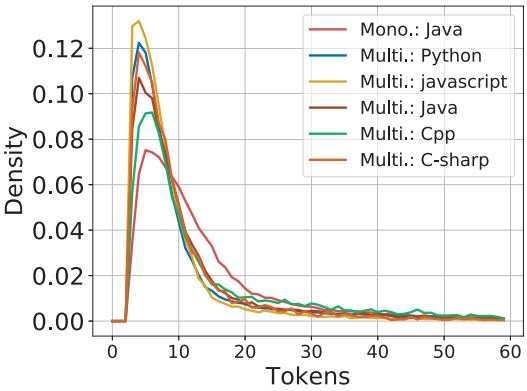


Figure 6: The distribution of commit message based on their size.

for the buggy code has a long tail that extends over 300 tokens, while the density for the commit message has a peak before 10 tokens. Figure 7 shows the distribution of the line number for the first appeared buggy line. It can be seen that the density for the line number also has a peak before the fifth line of buggy code.

C Hyperparameter Settings

We train our models[13] using Adam optimizer, the initial learning rate is 3×10^{-4}. The mini-batch size and the dropout rate are 16 and 0.1, respectively. We train our models for a maximum of 50 epochs[14]. To avoid overfitting, we implement the early stop if the validation performance does not increase for 10 consecutive iterations.

[10] https://www.nltk.org/

[11] https://pygments.org/

[12] During filtering commits, we have removed meaningless tokens in a commit message, such as punctuation, stop words, url, changes ID, etc., which avoids meaningless tokens affect the quality of filtered results

[13] The base Transformer model has about 40M parameters and the joint model introduce about 0.8M parameters over it.

[14] It takes about 30 minutes to train an epoch.

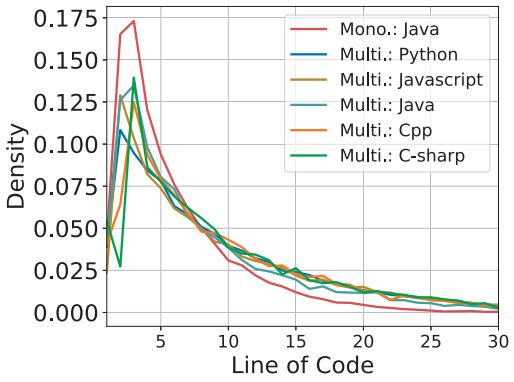


Figure 7: The distribution of the line of code that is first appeared in the buggy.

	Buggy Code	`bool HitRecord :: isValid () { return isNull; }`
Code Repair	Cascaded Model†	`bool HitRecord :: FindAll () { return }`
	Joint Model†	`bool HitRecord :: isValid () { return isNull; }`
	Joint Model‡	`bool HitRecord :: isValid () { return ! isNull; }`
	Target	`bool HitRecord :: isValid () { return ! isNull; }`
Commit Message Generation	Cascaded Model†	Color class fix.
	Joint Model†	Fix bug in KalmanFilter.
	Joint Model‡	Fix bug in HitRecord
	Target	Fix incorrect HitRecord.

Figure 8: Examples on our crawled Cpp dataset. † and ‡ mean the model is running under the monolingual setting and multilingual settings, respectively.

D Case Study

To further analyze our model under a low-resource setting, we present an example collected from the Cpp dataset. As shown in Figure 8, both the pipeline-based model and joint model, which are under the monolingual setting, fail to correctly repair code and appropriately generate commit messages, the most likely reason is that the lacking amount of data doesn't allow the model to successfully capture the useful patterns which are adapted to the specific task. Expectantly, The joint model under a multilingual setting successfully solves the returned value bug. Besides, it captures the key words "**HitRecord**" in generated commit message, which makes the message more relative to the code context. The example demonstrates that the model performs better under a multilingual setting compared to which under the monolingual setting, especially on the condition that the amount of monolingual data is limited.