Commit Message Generation for Source Code Changes

Shengbin Xu[1] , **Yuan Yao**[1] , **Feng Xu**[1] , **Tianxiao Gu**[2] , **Hanghang Tong**[3] and **Jian Lu**[1]
[1]State Key Laboratory for Novel Software Technology, Nanjing University, China
[2]Alibaba Group, USA
[3]Arizona State University, USA
kingxu@smail.nju.edu.cn, {y.yao, xf, lj}@nju.edu.cn, tianxiao.gu@gmail.com, hanghang.tong@asu.edu

Abstract

Commit messages, which summarize the source code changes in natural language, are essential for program comprehension and software evolution understanding. Unfortunately, due to the lack of direct motivation, commit messages are sometimes neglected by developers, making it necessary to automatically generate such messages. State-of-the-art adopts learning based approaches such as neural machine translation models for the commit message generation problem. However, they tend to ignore the code structure information and suffer from the out-of-vocabulary issue. In this paper, we propose CODISUM to address the above two limitations. In particular, we first extract both code structure and code semantics from the source code changes, and then jointly model these two sources of information so as to better learn the representations of the code changes. Moreover, we augment the model with copying mechanism to further mitigate the out-of-vocabulary issue. Experimental evaluations on real data demonstrate that the proposed approach significantly outperforms the state-of-the-art in terms of accurately generating the commit messages.

1 Introduction

In the practice of software development, a version control system is usually adopted which requires the developers to write a *commit message* to summarize each submitted code change. The commit message is in the form of natural language and thus can help software developers understand the high-level intuition behind the code change without the need to read the low-level implementation details. Consequently, commit messages become an indispensable part of software development, and play an essential role in software comprehension and maintenance.

Unfortunately, it is non-trivial for developers to write a meaningful summary to precisely capture the changes made to the source code written in certain programming languages. Furthermore, writing messages for each commit even becomes a burden to developers since the code base may evolve and grow incrementally with a number of fine, small commits rather than a huge change. Due to the lack of direct motivation, commit messages are sometimes overly neglected by developers. For example, it is reported that around 14% of the commit messages in more than 23,000 open source Java projects hosted in SourceForge[1] are completely empty [Dyer *et al.*, 2013]. Therefore, automatically summarizing code changes becomes an important task. We refer to this task as *commit message generation* in this paper.

To date, several commit message generation approaches have been proposed in the literature. These approaches are developed in three stages (see Section 4 for more discussions). First, early work translates code changes into natural language descriptions based on pre-defined rules and templates [Buse and Weimer, 2010; Cortés-Coy *et al.*, 2014; Shen *et al.*, 2016]. Nevertheless, these descriptions sometimes fail to explain the reasons or purposes of code changes and thus become tedious and useless in practice. Later, some researchers utilize information retrieval techniques to generate commit messages. Their basic idea is to re-use the commit messages of similar code changes [Huang *et al.*, 2017; Liu *et al.*, 2018]. The main limitation of these approaches is that they cannot output accurate summaries if there are no similar code changes. In recent years, there is a tendency to adopt learning based techniques for the commit message generation problem [Jiang *et al.*, 2017; Loyola *et al.*, 2017; Loyola *et al.*, 2018]. Typically, these approaches translate the source code changes into commit messages using *neural machine translation (NMT)* models.

Although learning based approaches have obtained promising results in comparison with earlier work, they still suffer from the following two limitations. First, NMT usually can only maintain a limited vocabulary of the most frequent words, making it difficult to generate the *out-of-vocabulary (OOV)* words such as the self-defined class names or variable names. Second, when learning the representations of the code changes, existing approaches tend to follow the "naturalness" hypothesis of software [Hindle *et al.*, 2012], and directly feed the code changes into the model. However, such treatment only encodes the code semantics while ignores the code structure.

For better understanding the above two limitations, we present an illustrative example in Figure 1, which shows a

[1]https://sourceforge.net/

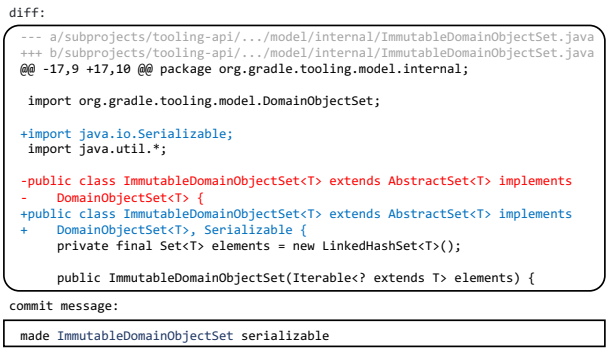


Figure 1: An example of a code change and its commit message.

diff	public class ImmutableDomainObjectSet<T>
Substitution	ImmutableDomainObjectSet ↔ c0, T ↔ n0
Structure	public class c0<n0>
Semantics	c0:"immutable domain object set", n0:"t"

Table 1: The extracted code structure and code semantics for the example in Figure 1.

piece of code change (or diff) as well as its commit message. Typically, a code change contains various program tokens such as keywords, operators, and identifiers (i.e., names for classes, methods, and variables). In the example, the generated commit message contains the class name "ImmutableDomainObjectSet". However, this word is usually treated as an OOV word as it may appear only a few times or even once in all the diffs, making it difficult for existing NMT models to generate the word. Moreover, the meaning of a change is the same for any other class that makes the same change (e.g., adding "Serializable" into the class declaration), although such a class may have a different class name. Therefore, in addition to the semantics contained in the self-defined identifiers, the code skeleton/structure[2] should be jointly modeled to better learn the representations of the code changes.

In this paper, we propose a novel approach CODISUM for the commit message generation problem, addressing the above two limitations. In particular, we first extract both the code structure and the code semantics from the code changes. For the code structure, we identify all the class/method-/variable names and replace them with the corresponding placeholders. For the code semantics, we segment each class/method/variable name into several single words. Here, the word segmentation is based on the well-known *naming conventions* widely adopted by developers. For example, part of the extracted code structure and code semantics for Figure 1 are shown in Table 1, where we use the class name "ImmutableDomainObjectSet" as an example to illustrate our idea. First, we identify the class name and replace it with a placeholder "c0". Then, we segment it into a sequence of words, i.e., "immutable domain object set".

Next, we model both code structure and code semantics with a *recurrent neural network (RNN)*, and further align the two RNNs at the position of each placeholder. With this step, we can combine the learned representations of "immutable domain object set" (semantic representation) and "c0" (structural representation) as the overall representation for the class name "ImmutableDomainObjectSet". Finally, to allow directly copying the OOV words from the input code changes to the output commit messages, we incorporate the copying mechanism [See *et al.*, 2017] in our model. For example, the model may directly copy "c0" into the generated message, and we substitute "c0" with "ImmutableDomainObjectSet" as the final output.

In summary, the main contributions of this paper include:

- We propose a commit message generation approach CODISUM that jointly learns the representations of both code structure and code semantics. The model can also mitigate the OOV problem.
- We conduct experimental evaluations on real data, demonstrating that the proposed approach significantly outperforms the state-of-the-art in terms of the accuracy of generating commit messages.

The rest of the paper is organized as follows. Section 2 describes the proposed approach. Section 3 shows the experimental results. Section 4 reviews the related work, and Section 5 concludes.

2 The Proposed Approach

In this section, we present the proposed approach, which is named as CODISUM (Code Diff Summarization).

2.1 Overview

The overview of the proposed CODISUM is shown in Figure 2, which contains an encoder part and a decoder part.

The input of the model is a piece of source code change. Then, in the encoder part, we first extract the code structure from the input by recognizing the identifiers (i.e., the names of classes, methods, and variables). We replace all the identifiers with the corresponding placeholders[3] and then learn the representations of the resulting code structure. For each identifier, we also learn its semantic representation, and combine the learned semantics with the corresponding placeholder as the overall representation of this identifier. The overall representation of each input word is then fed into an attention layer to obtain the final representation of the input code change.

For the decoder part, we use a multi-layer unidirectional GRU [Cho *et al.*, 2014] to generate a sequence of words as the initial commit message. Meanwhile, we incorporate the copying mechanism [See *et al.*, 2017] to allow directly copying the OOV words from the code changes. Note that such copying applies on the code structure, which directly copies the placeholders instead of the original input words. Finally, we substitute the placeholders with the corresponding input words as the final generated commit message.

[2]In this work, we define the code structure as the code skeleton by substituting identifiers with placeholders.

[3]We use different placeholders for the identifiers in each code change. For different code changes, we re-use the placeholders.

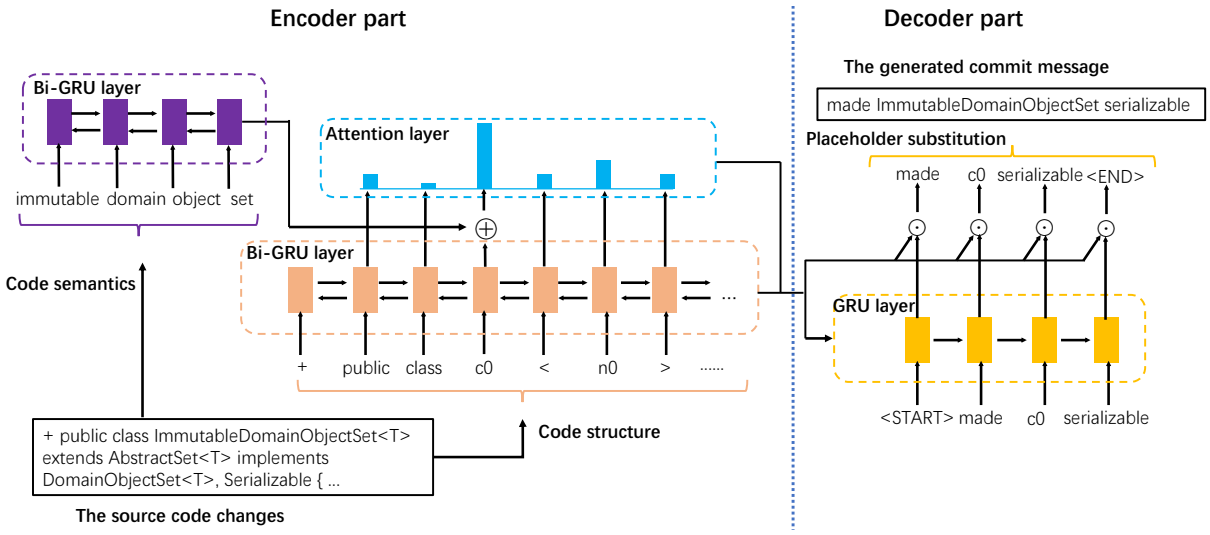


Figure 2: The overview of the proposed approach.

2.2 Modeling Code Structure

In the following, we describe some details of the CODISUM model. With code structure extracted from the input `diff`, we denote the embeddings of the words/placeholders in the code structure as $[x_1, x_2, \ldots, x_N]$, where $x_i \in \mathbb{R}^{d_x}$ is the embedding of the i-th word and N is the maximum length of the structure sequence. Note that the embeddings of all the words including those in code changes and commit messages are maintained in the same lookup table.

To model the code structure, we use the multi-layer bidirectional GRUs[4]. Each neuron takes the word embedding x_i and the output of the previous neuron h_{i-1} as input, and outputs h_i for the current word,

$$h_i = \text{Bi-GRU}\left(x_i, h_{i-1}\right), \tag{1}$$

where $h_i \in \mathbb{R}^{d_h}$ is the hidden state of the GRU neuron.

Note that the `diff` usually contains a sign (i.e., "+" or "-") that allows us to distinguish if the lines are added or removed. In practice, we also maintain a lookup table for these signs and add the corresponding embedding before each word, i.e., $x_i = [y_i; x_i]$, where y_i is the embedding of "+" or "-".

2.3 Modeling Code Semantics

If the word x_i in the code structure is a placeholder (i.e., the original word in the code changes is an identifier[5]), we also use multi-layer bi-directional GRUs to encode the semantics of the identifier. By splitting the identifier into separate words based on the naming convention, and denoting the resulting word sequence as $[z_{i,1}, z_{i,2}, \ldots, z_{i,N'}]$ where $z_{i,j} \in \mathbb{R}^{d_z}$ is the embedding of the j-th separate word in x_i and N' ($N' \ll N$) is the maximum length of the separate word sequences, we have

$$h_{i,j} = \text{Bi-GRU}\left(z_{i,j}, h_{i,j-1}\right), \tag{2}$$

where $h_{i,j} \in \mathbb{R}^{d_h}$ in the hidden state of the j-th separate word. Here, for the separate words, we also maintain them in the same lookup table with the other words in code changes and commit messages (i.e., $d_x = d_z$).

Finally, since we use a multi-layer bi-directional structure of RNNs, we concatenate the hidden states of the two last neurons of the top-layer in both directions as the semantic representation of the input sequence. That is, we have the semantic representation of word x_i as

$$z_i = \left[\overrightarrow{h}_{i,N'}; \overleftarrow{h}_{i,N'}\right], \tag{3}$$

where $\overrightarrow{h}_{i,N'}$ and $\overleftarrow{h}_{i,N'}$ are the hidden states from the forward direction and the backward direction, respectively.

2.4 Combining Code Structure and Semantics

Next, we describe how we jointly model code structure and code semantics. Although we can simply learn the two representations and directly concatenate them as the encoder results, such a method would miss the correspondence between code structure and code semantics, and thus leading to inferior generation accuracy. This is also confirmed by our experiments as we will later show.

In CODISUM, we align code structure and code semantics. Specially, for each input `diff`, we maintain N sequences corresponding to the N input words. For example, if the i-th word x_i is an identifier, we learn its semantic presentation z_i via Eq. (3) and concatenate it with the hidden state h_i (i.e., structural representation) from Eq. (1) as the overall representation of x_i, i.e.,

$$h'_i = [h_i; z_i]. \tag{4}$$

Otherwise, if the i-th word is not an identifier, we do zero-padding after h_i to generate the combined representation h'_i for word x_i, i.e., $h'_i = [h_i; 0s]$.

Attention Layer. For the decoder part, we assume that decoder output at step t as s_t. The attention distribution is then

[4]We omit the multi-layer structure in Figure 2 for brevity.

[5]Without ambiguity, we directly use x_i to indicate the word with embedding x_i in this paper.

calculated as follows,

$$\begin{aligned} e_i^t &= v^T \tanh\left(W_h\left[h_i; z_i\right] + W_s s_t\right) \\ \alpha_i^t &= \text{softmax}\left(e_i^t\right), \end{aligned} \tag{5}$$

where v, W_h, and W_s are learnable parameters. The normalized α_i^t determines how much attention should be given to $h'_i = [h_i; z_i]$ when generating the t-th word in the commit message. Then, we can produce a weighted sum of h'_i, which is also known as the context vector, as the representation of the input code change,

$$h_t^* = \sum_{i=1}^{N} \alpha_i^t h'_i. \tag{6}$$

Copying Mechanism. Finally, we incorporate the copying mechanism [See *et al.*, 2017]. This mechanism directly copies some words (e.g., class names) from the input code changes to the generated commit messages.

Without copying mechanism, we can generate the t-th word based on the following distribution,

$$P' = \text{softmax}\left(W\left[h_t^*; s_t\right] + b\right), \tag{7}$$

where W and b are learnable parameters. Here, P' contains the *generation distribution* over all the words in the vocabulary. To allow copying, we further compute a *copying distribution* over the words in the structure sequence $[x_1, x_2, \ldots, x_N]$ as,

$$P'' = D^T \alpha^t, \tag{8}$$

where $\alpha^t = [\alpha_1^t; \alpha_2^t; \ldots; \alpha_N^t]$ is the attention vector, $D \in \mathbb{R}^{N \times V}$ is the one-hot coding of the structure sequence, and V is the vocabulary size. As indicated by the above equation, we tend to copy the words with higher attention.

Then, we only need to determine the probability of choosing a word from the *generation distribution* or the *copying distribution*. Specially, we use the following equation to compute this probability,

$$p = \sigma\left(w'\left[h_t^*; s_t\right] + b'\right), \tag{9}$$

where vector w' and scalar b' are learnable parameters, and σ is the sigmoid function. We obtain the final probability distribution of generating a word w in the commit message as

$$P(w) = (1-p)\, P'(w) + p \cdot P''(w). \tag{10}$$

3 Experimental Evaluations

3.1 Experimental Setup

(A) Dataset. We use the commonly-used dataset in this area, which was collected by Jiang and McMillan [2017] from the top 1,000 popular Java projects in Github and contains 509k `diff` files and corresponding commit messages. We make the following pre-processing steps on the dataset. First, we remove the `diff` files that contain file changes other than *.java* files. Next, we apply standard NLP processing on the data. That is, we keep only the source code in the `diff` files, remove the punctuations and special symbols in the commit messages, tokenize `diffs` and commit messages, and remove the data that contains less than three words. Finally, we delete the duplicated `diffs` as there are some successive commits in a short period fixing the same bug with the same commit message. After pre-processing, we obtain $90,661$ pairs of $\langle$*diff, commit message*$\rangle$ and randomly choose $75,000$ for training, $8,000$ for validation, and $7,661$ for testing.

(B) Evaluation Metrics. We first use two metrics that are widely used in natural language processing, i.e., BLEU score [Papineni *et al.*, 2002] and METEOR [Denkowski and Lavie, 2014], to measure how close are the generated sentences compared to the real sentences. Moreover, since we aim to generate some OOV words, we also calculate the Recall metric of the correctly generated identifiers in the commit messages.

(C) Compared Approaches. We compare the following approaches in our experiments.

- *NMT* [Jiang *et al.*, 2017; Loyola *et al.*, 2017]. NMT uses attention-based RNNs to translate `diffs` into commit messages. It treats `diffs` and commit messages as two different languages.
- *NNGen* [Liu *et al.*, 2018]. NNGen is an information retrieval approach. It represents `diffs` as "bags-of-words" vectors, and re-uses the commit message from the most similar `diff` calculated with cosine similarity.
- *CopyNet* [See *et al.*, 2017]. CopyNet is a text summarization method, which incorporates copying mechanism into the decoder to allow copying words from source to target.
- CODISUM. CODISUM is the proposed approach built upon the NMT model[6].

(D) Parameters and Initializations. For the encoder part, we set the maximum length of the structure sequence and the semantics sequence to 200 (i.e., $N = 200$) and 5 (i.e., $N' = 5$), respectively. For the decoder part, the maximum length of commit message is set to 20. All the word embedding dimensionality is set to 150 (i.e., $d_x = d_z = 150$) with random initialization. For all multi-layer structure of RNNs, we set the layer number to 3. In the Bi-GRUs, the hidden state dimensionality of one direction is set to 128, and overall hidden state dimensionality is 256 (i.e., $d_h = 256$). We also set the hidden state dimensionality of the decoder GRUs as 256. All the compared methods are set with the same parameters if applicable (e.g., the hidden state of NMT and CopyNet is set to 256). When training, we adopt the categorical cross-entropy loss function and RMSProp optimizer with batch size 100 and dropout rate 0.1. We stop the training process when the loss is no longer decreasing.

3.2 Experimental Results

Effectiveness Comparisons

For effectiveness, we compare the proposed CODISUM with the existing competitors, and show the results in Table 2.

We can first observe from the table that the proposed CODISUM outperforms the compared methods on all the

[6]The code is publicly available at https://github.com/SoftWiser-group/CoDiSum.

Model	BLEU (%)	METEOR (%)	Recall (%)
NMT	0.87	4.81	2.98
NNGen	2.16	4.34	10.60
CopyNet	1.16	5.52	14.48
CODISUM	**2.19**	**7.46**	**33.96**

Table 2: Effectiveness comparison results. The proposed CODISUM outperforms the existing methods on all the three evaluation metrics.

Model	BLEU (%)	METEOR (%)	Recall (%)
CODISUM$_1$	2.06	7.04	30.21
CODISUM$_2$	1.97	6.86	29.63
CODISUM$_3$	2.05	7.14	33.87
CODISUM	2.19	7.46	33.96

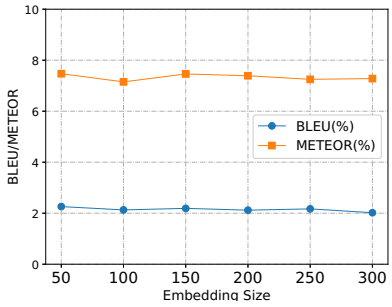
Table 3: Performance gain analysis of CODISUM. The results indicate that the joint modeling of code structure and code semantics, the copying mechanism, and the identifier substitution are all helpful for commit message generation.

three metrics. In the competitors, CopyNet significantly outperforms NMT. Since CopyNet incorporates the copying mechanism into NMT, this result indicates the importance of applying copying mechanism in commit message generation. Compared to CopyNet, the relative improvements of CODISUM are 88.8%, 35.1%, and 134.5% w.r.t. BLEU, METEOR, and Recall, respectively. There are two major differences between CopyNet and CODISUM. First, we share all the word embeddings from code changes and commit messages. Second, we jointly model the code structure and the code semantics. The improvements indicate the importance of these two differences. Compared to NNGen, CODISUM improves it especially on the METEOR and Recall metrics. The improvement on the BLEU metric is relatively minor. One possible reason is that the BLEU metric favors much fluent sentences as those re-used by NNGen.

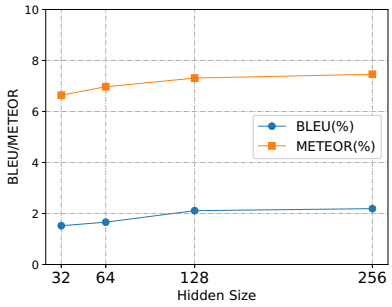
Additionally, we can observe that the improvement of the Recall metric is especially larger. The reason is as follows. CODISUM can either generate the identifiers with the generation distribution (we can easily put all the placeholders in the vocabulary) or copy the placeholders with the copying distribution. In contrast, CopyNet can only generate such words with the copying distribution. Therefore, although the generated messages are of close length, the generated messages from CODISUM contain many more identifiers.

Performance Gain Analysis

Next, we analyze the performance gain of the proposed approach. The results are shown in Table 3. In the table, CODISUM$_1$ is the variant that deletes the copying mechanism in CODISUM. Compared to NMT (whose results are in Table 2), CODISUM$_1$ jointly models code structure and code semantics. As we can see, CODISUM$_1$ significantly improves NMT indicating the usefulness of our joint modeling. Meanwhile, as indicated by the Recall metric, by allowing the generation of placeholders and the followed identifier substitution, CODISUM$_1$ can already mitigate the OOV issue.



(a) The effect of the word embedding size d_x



(b) The effect of the hidden state size d_h

Figure 3: The parameter sensitivity study. The performance of CODISUM stays relatively stable w.r.t. the two parameters. We fix $d_x = 150$ and $d_h = 256$ in this work.

Also, CODISUM is still better than CODISUM$_1$, indicating the usefulness of the copying mechanism.

The second variant CODISUM$_2$ deletes the code semantics part in CODISUM. Compared to CopyNet (whose results are in Table 2), this variant replaces the identifiers with placeholders to reduce vocabulary size. The results show that CODISUM$_2$ is much better than CopyNet, indicating the importance of identifier substitution.

The third variant CODISUM$_3$ directly concatenates the semantic representation and the structural representation without alignment. As we can see, CODISUM improves CODISUM$_3$ which, again, verifies the usefulness of the proposed joint modeling method.

Parameter Sensitivity

Next, we study the effects of two parameters in the proposed method, including word embedding size d_x and Bi-GRU hidden state size d_h. The results are shown in Figure 3, where we report the BLEU and METEOR scores. Figure 3(a) shows that the BLEU and METEOR scores are stable when the word embedding size varies from 50 to 300. We fix the embedding size as $d_x = 150$ in this work. Figure 3(b) shows that the BLEU and METEOR scores increase when the hidden state size increases. However, when d_h grows from 128 to 256, only slight performance gain is observed. Considering the time cost, we set the hidden state size d_h as 256.

Case Studies

Finally, we present an example of the generated messages from the compared methods. The results

Real Message	Put `ErrorReporter checkReportsOnApplicationStart` back in the public api
NMT	Fix quality flaw
NNGen	Allow clean application exception handling to report exceptions
CopyNet	Fix a typo in `ErrorReporter`
CODISUM	Make `checkReportsOnApplicationStart` public

Table 4: An example of the generated commit messages. CODISUM generates more accurate messages and successfully copies the method name.

are shown in Table 4 where the real commit message written by human is "Put `ErrorReporter checkReportsOnApplicationStart` back in the public api". Although the meaning of the generated messages by the compared methods is close to that of the real message, the generated message by CODISUM is more accurate. Moreover, CODISUM can successfully generate/copy the right method name `checkReportsOnApplicationStart`. In contrast, NMT and NNGen do not copy the method name, and CopyNet copies the class name `ErrorReporter` instead of the more proper method name.

4 Related Work

In this section, we briefly review the major related work and divide them into three parts: (1) commit message generation or source code change summarization, (2) code summarization, and (3) text summarization.

Commit Message Generation

Existing commit message generation methods in literature can be categorized into three classes. In the first class, researchers extract information from code changes and translate it into natural language based on pre-defined rules or templates [Buse and Weimer, 2010; Cortés-Coy *et al.*, 2014; Linares-Vásquez *et al.*, 2015; Shen *et al.*, 2016]. For example, Buse *et al.* [2010] analyze the control flow the program and use the template "do Y Instead of Z" to generate the commit messages; Cortés-Coy *et al.* [2014] focus on several types of commit intents (e.g., creating and destroying objects, modifying the attribute getting and setting of an object, etc.), and accompany these intents with pre-defined rules. The limitations of these approaches lie in two aspects: 1) they can only deal with the pre-defined types of code changes, and 2) the generated commit messages do not explain the intuition behind the code changes, which requires human insight.

In the second class, in order to generate commit messages that are easily understandable by humans, Huang *et al.* [2017] and Liu *et al.* [2018] propose to search similar code changes and reuse their commit messages. However, these approaches heavily rely on whether similar code changes can be retrieved, and how similar the code changes are.

In the third class, deep learning models such as neural machine translation have been used to translate the code changes into commit messages [Loyola *et al.*, 2017; Jiang *et al.*, 2017; Loyola *et al.*, 2018]. For example, Jiang *et al.* [2017] directly apply NMT models to make the translation; Loyola *et al.* [2018] further incorporate the context of the code changes into the model. Although the results of these approaches are promising, they still suffer from two challenges, i.e., how to incorporate code structure, and how to generate OOV words. In this work, we build our model upon the NMT model and aim to address the two challenges via jointly modeling the code structure and code semantics.

Code Summarization

Our work is also related to code summarization, which aims at generating descriptions of code snippets. In literature, Sridhara *et al.* [2010] propose to summarize Java methods by identifying important statements in the Java methods and then transforming these statements into natural language descriptions. Movshovitz-Attias and Cohen [2013] generate comments from code using *n*-gram models and topic models. Oda *et al.* [2015] translate Python code to pseudo-code (in natural language) using machine translation techniques, aiming at improving the code readability. Iyer *et al.* [2016] design a neural attention model that summarizes code as text. Hu *et al.* [2018a; 2018b] combine the neural machine translation model with the structural information (e.g., abstract syntax tree, AST) and the API knowledge to generate the summaries.

Different from the code summarization methods, we aim at summarizing code changes instead of code snippets. Existing code summarization methods mainly extract the AST structure of code. However, such methods cannot be directly applied to code changes as we usually cannot extract the AST structure from code changes, which are short and incomplete compared to code snippets.

Text Summarization

Finally, our work is related to the general text summarization methods [Reiter and Dale, 1997; Rush *et al.*, 2015; Chopra *et al.*, 2016; Nallapati *et al.*, 2016; See *et al.*, 2017]. As a building block of the proposed approach, we also use the copying mechanism inspired by the pointer-generator networks [See *et al.*, 2017].

5 Conclusions

In this paper, we propose a learning based approach for the commit message generation problem. In particular, built upon the machine translation model, we jointly model the code structure and code semantics of the code changes to better learn their representations, and further augment the model with copying mechanism to mitigate the out-of-vocabulary issue. Experimental results demonstrate that the proposed approach significantly outperforms the state-of-the-art in terms of accurately generating the commit messages.

Future directions include incorporating additional context information and code structure information for the commit message generation problem.

Acknowledgments

This work is supported by the National Natural Science Foundation of China (No. 61690204, 61672274, 61702252) and the Collaborative Innovation Center of Novel Software Technology and Industrialization. Hanghang Tong is partially supported by NSF (IIS-1651203, IIS-1715385 and IIS-1743040). Yuan Yao is the corresponding author.

References

[Buse and Weimer, 2010] Raymond PL Buse and Westley R Weimer. Automatically documenting program changes. In *ASE*, pages 33–42, 2010.

[Cho *et al.*, 2014] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv*, 2014.

[Chopra *et al.*, 2016] Sumit Chopra, Michael Auli, and Alexander M Rush. Abstractive sentence summarization with attentive recurrent neural networks. In *NAACL*, pages 93–98, 2016.

[Cortés-Coy *et al.*, 2014] Luis Fernando Cortés-Coy, Mario Linares-Vásquez, Jairo Aponte, and Denys Poshyvanyk. On automatically generating commit messages via summarization of source code changes. In *SCAM*, pages 275–284, 2014.

[Denkowski and Lavie, 2014] Michael Denkowski and Alon Lavie. Meteor universal: Language specific translation evaluation for any target language. In *Proceedings of the ninth workshop on statistical machine translation*, pages 376–380, 2014.

[Dyer *et al.*, 2013] Robert Dyer, Hoan Anh Nguyen, Hridesh Rajan, and Tien N Nguyen. Boa: A language and infrastructure for analyzing ultra-large-scale software repositories. In *ICSE*, pages 422–431, 2013.

[Hindle *et al.*, 2012] Abram Hindle, Earl T Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. On the naturalness of software. In *ICSE*, pages 837–847, 2012.

[Hu *et al.*, 2018a] Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. Deep code comment generation. In *ICPC*, pages 200–210, 2018.

[Hu *et al.*, 2018b] Xing Hu, Ge Li, Xin Xia, David Lo, Shuai Lu, and Zhi Jin. Summarizing source code with transferred api knowledge. In *IJCAI*, pages 2269–2275, 2018.

[Huang *et al.*, 2017] Yuan Huang, Qiaoyang Zheng, Xiangping Chen, Yingfei Xiong, Zhiyong Liu, and Xiaonan Luo. Mining version control system for automatically generating commit comment. In *ESEM*, pages 414–423, 2017.

[Iyer *et al.*, 2016] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. Summarizing source code using a neural attention model. In *ACL*, pages 2073–2083, 2016.

[Jiang and McMillan, 2017] Siyuan Jiang and Collin McMillan. Towards automatic generation of short summaries of commits. In *ICPC*, pages 320–323, 2017.

[Jiang *et al.*, 2017] Siyuan Jiang, Ameer Armaly, and Collin McMillan. Automatically generating commit messages from diffs using neural machine translation. In *ASE*, pages 135–146, 2017.

[Linares-Vásquez *et al.*, 2015] Mario Linares-Vásquez, Luis Fernando Cortés-Coy, Jairo Aponte, and Denys Poshyvanyk. Changescribe: A tool for automatically generating commit messages. In *ICSE*, pages 709–712, 2015.

[Liu *et al.*, 2018] Zhongxin Liu, Xin Xia, Ahmed E Hassan, David Lo, Zhenchang Xing, and Xinyu Wang. Neural-machine-translation-based commit message generation: how far are we? In *ASE*, pages 373–384, 2018.

[Loyola *et al.*, 2017] Pablo Loyola, Edison Marrese-Taylor, and Yutaka Matsuo. A neural architecture for generating natural language descriptions from source code changes. In *ACL*, pages 287–292, 2017.

[Loyola *et al.*, 2018] Pablo Loyola, Edison Marrese-Taylor, Jorge Balazs, Yutaka Matsuo, and Fumiko Satoh. Content aware source code change description generation. In *INLG*, pages 119–128, 2018.

[Movshovitz-Attias and Cohen, 2013] Dana Movshovitz-Attias and William W Cohen. Natural language models for predicting programming comments. In *ACL*, pages 35–40, 2013.

[Nallapati *et al.*, 2016] Ramesh Nallapati, Bowen Zhou, Cicero dos Santos, Caglar Gulcehre, and Bing Xiang. Abstractive text summarization using sequence-to-sequence rnns and beyond. In *CoNLL*, pages 280–290, 2016.

[Oda *et al.*, 2015] Yusuke Oda, Hiroyuki Fudaba, Graham Neubig, Hideaki Hata, Sakriani Sakti, Tomoki Toda, and Satoshi Nakamura. Learning to generate pseudo-code from source code using statistical machine translation (t). In *ASE*, pages 574–584, 2015.

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

[Reiter and Dale, 1997] Ehud Reiter and Robert Dale. Building applied natural language generation systems. *Natural Language Engineering*, 3(1):57–87, 1997.

[Rush *et al.*, 2015] Alexander M Rush, Sumit Chopra, and Jason Weston. A neural attention model for abstractive sentence summarization. In *EMNLP*, pages 379–389, 2015.

[See *et al.*, 2017] Abigail See, Peter J Liu, and Christopher D Manning. Get to the point: Summarization with pointer-generator networks. In *ACL*, pages 1073–1083, 2017.

[Shen *et al.*, 2016] Jinfeng Shen, Xiaobing Sun, Bin Li, Hui Yang, and Jiajun Hu. On automatic summarization of what and why information in source code changes. In *COMPSAC*, volume 1, pages 103–112, 2016.

[Sridhara *et al.*, 2010] Giriprasad Sridhara, Emily Hill, Divya Muppaneni, Lori Pollock, and K Vijay-Shanker. Towards automatically generating summary comments for java methods. In *ASE*, pages 43–52, 2010.