2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)

Generating Commit Messages from Diffs using Pointer-Generator Network

Qin Liu[†] [‡], Zihe Liu[†], Hongming Zhu[†] *, Hongfei Fan[†], Bowen Du[§] *, Yu Qian[†]
[†]School of Software Engineering, Tongji University, Shanghai, China
[‡]Tsingtao Advanced Research Institute, Tongji University, Tsingtao, China
[§]Department of Computer Science, The University of Warwick, Coventry, United Kingdom
{qin.liu, 1015ziheliu, zhu_hongming, fanhongfei, 100price100}@tongji.edu.cn, B.Du@warwick.ac.uk

***Abstract*—The commit messages in source code repositories are valuable but not easy to be generated manually in time for tracking issues, reporting bugs, and understanding codes. Recently published works indicated that the deep neural machine translation approaches have drawn considerable attentions on automatic generation of commit messages. However, they could not deal with out-of-vocabulary (OOV) words, which are essential context-specific identifiers such as class names and method names in code diffs. In this paper, we propose PtrGNCMsg, a novel approach which is based on an improved sequence-to-sequence model with the pointer-generator network to translate code diffs into commit messages. By searching the smallest identifier set with the highest probability, PtrGNCMsg outperforms recent approaches based on neural machine translation, and first enables the prediction of OOV words. The experimental results based on the corpus of diffs and manual commit messages from the top 2,000 Java projects in GitHub show that PtrGNCMsg outperforms the state-of-the-art approach with improved BLEU by 2.01, ROUGE-1 by 4.47 and ROUGE-L by 4.23, respectively.**

***Index Terms*—automatic commit message generation; sequence-to-sequence model; pointer-generator network; code change pattern recognition;**

I. Introduction

Version control systems (VCS) are widely used to organize multiple versions during the lifecycle of a software project. One of the valuable assets in VCS is the commit message based on code diffs between two versions of a project. The code diff refers to the difference of codes between two versions, including code lines added or deleted in each file. A commit message is a short natural statement that summarizes the corresponding code diffs and describes the purpose of the changes. High-quality commit messages are necessary, because they make it easier for developers to understand code changes and version evolutions from a high level without reading and analyzing detailed code changes. They are also used to track project actions including issues, feature additions, and bug reports [1, 2, 3]. However, the quality of commit messages in real projects is unstable due to the lack of experience and the negligence under market pressures [3, 4], which has been studied to affect many other types of documentations [5, 6, 7]. In summary, commit messages are necessary but difficult to be written in high quality.

* Hongming Zhu and Bowen Du are corresponding authors.

Approaches to automatically generating commit messages have been proposed to improve their quality. Some researchers adopted predefined templates and rules to generate static commit messages. Buse *et al.* [2] extracted path predicates of changed statements and generated commit messages with pseudocode. Linares-Vásquez *et al.* [8] compared two abstract syntax trees and adopted predefined rules to generate commit messages. These approaches use long sentences based on static templates to describe where and what changes, but a high-quality manual commit message is usually a short sentence focusing on not only locations of code changes, but also the motivation of these code changes. To generate commit messages using a short sentence and describing the purpose of code changes, Jiang *et al.* [3] and Loyola *et al.* [9] adapted a deep neural machine translation model, the RNN Encoder-Decoder model [10] with attention mechanism [11], to learn the semantics of code changes and summarization patterns. This approach takes code changes as the source language into the deep learning models, and outputs predicted commit messages with short natural language sentences as the target language.

However, these deep neural machine translation models are difficult to deal with out-of-vocabulary (OOV) words, because they can only generate words from a pre-defined vocabulary [12]. To maintain the generalization of models, the pre-defined vocabulary contains only high-frequency words. The context-specific identifiers (file names, class names, method names and variable names) are specific for one project, so they are usually not included in the vocabulary. In this case, the commit messages automatically generated would always miss these important context-specific identifiers. For example, as for a high-quality manual commit message "fixed a bug in ReplicatedChronicleMap.value()", it would be too vague to understand if the context-specific identifier "ReplicatedChronicleMap" is deleted.

To address this issue, we propose *PtrGNCMsg*, a novel approach which is based on the pointer-generator network and translates code diffs into commit messages. The pointer-generator network is an adapted version of the attention RNN Encoder-Decoder models, which can either copy words from the source sentence or generate words from the pre-defined vocabulary. At each decoder prediction time step, the

2574-3864/19/$31.00 ©2019 IEEE
DOI 10.1109/MSR.2019.00056

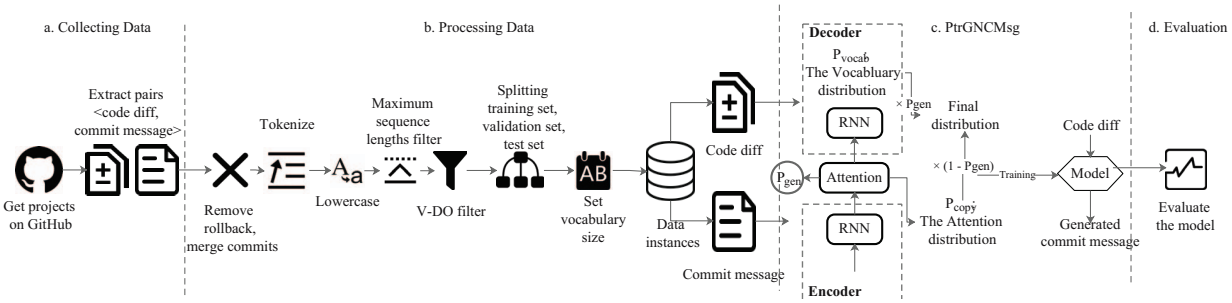

Fig. 1: The overall workflow of our approach.

RNN Decoder calculates the probability p_{copy} of copying words from the source sentence according to the attention distribution, and the probability p_{vocab} of selecting words in the vocabulary. These probabilities are integrated to gain the final probability of each word in the pre-defined vocabulary and source sentence.

By adopting this deep neural network approach, we convert the commit message generation problem into a sequence-to-sequence problem, taking tokenized code changes as the source language and commits messages as the target language. As a result, it generates more accurate commit messages with OOV identifiers by copying them from code changes to the generated commit message.

Fig. 1 shows the overall workflow of our approach, mainly including four steps.

Step 1. We collect and extract pairs <code changes, commit message> from top Java projects in descending order of stars in GitHub. We also use the pairs from Java projects collected by Jiang *et al.* [3] as raw datasets.

Step 2. We tokenize each sentence with punctuations and whitespaces, lowercase the raw datasets and filter them with some patterns. We respectively split each dataset into the training set, validation set, and test set. To make the vocabulary contain the most generic words, we search the smallest word candidates set to cover the most parts of sentences [13].

Step 3. We use the proposed PtrGNCMsg model to generate commit messages.

Step 4. We measure the accuracy of the model using BLEU [14] and ROUGE [15], which are widely used to evaluate machine summarization and translation.

Experimental results show that our approach achieves higher average BLEU and ROUGE scores than the standard attention RNN Encoder-Decoder models on the two datasets above.

The main contributions of our work are as follows:

- We proposed PtrGNCMsg, a novel approach which is based on an improved sequence-to-sequence model with the pointer-generator network to translate code diffs into commit messages, and first enables the prediction of indispensable OOV context-specific identifiers.
- We collected more data instances on code diffs and corresponding commit messages from GitHub, which can be used for further research. Our dataset is accessible at https://zenodo.org/record/2529947#.XCyPk88zbBJ.

The rest of this paper is organized as follows. Section II presents the related work. Section III describes the detailed design of the proposed approach including dataset preparation, the model, and the training procedure. Section IV shows the evaluation results. Lastly, we conclude the paper in Section V.

II. Related Work

A. Mining API and Code Changes

Data mining techniques have been adapted to code changes to detect programming usage patterns. These works can be categorized into two groups based on the levels of programming changes.

The first group mines API-level programming changes to find API (Application Programming Interface) usage patterns. LIBSYNC [16] guides developers in migrating API usage from the old version of a library to the new version by mining other projects which have migrated to the new version library. It extracts API usage graphs from projects used this library of two versions and adapts frequent item-set mining to detect migration patterns. The approach of Dagenais *et al.* [17] recommends adaptive changes in a framework by analyzing how the framework is adapted to its changes. Uddin *et al.* [18] proposed a clustering technique to detect API usage patterns from the change history of a client program.

Different from mining API-level programming changes, the second group mines code changes to detect code usage patterns. Negara *et al.* [19] mined fine-grained code changes to detect unknown change patterns. Their technique is based on the CHARM algorithm [20] for closed frequent item-set mining on continuous sequences of code changes which are ordered by their timestamps. To find groups of similar code changes from VCS, Kreutzer *et al.* [21] evaluated two syntactical similarity metrics: one based on agglomerative hierarchical clustering and the other one based on DBSCAN clustering algorithm. Molderez *et al.* [22] adapted standard frequent item-set mining to detect unknown systematic edits from fine-grained code changes based on VCS information.

These approaches use data mining techniques to detect programming usage patterns from programming changes. How-

ever, they cannot summarize the code changes with a phrase in natural language.

B. Traditional Commit Message Generation Techniques

Some traditional techniques have been proposed for automatically generating commit messages to summarize the code changes. They mainly use predefined templates and rules to generate static commit messages. DeltaDoc [2] extracts path predicates of changed statements and generates commit messages with pseudocode for each changed method according to predefined templates. However, it does not consider the relationship between multiple changed methods in a commit of code changes. ChangeScribe [8] compares two Abstract Syntax Trees (AST) and uses predefined rules to generate commit messages. It mainly focuses on where code changes occur from different levels including package, class, and method.

These approaches only generate fairly long messages with static rules and templates, and the generated messages are usually about the location and content of code changes [2]. However, high-quality manual commit messages typically summarize the purpose of code changes with short natural sentences, which has to integrate patterns and semantics of code changes.

C. Deep Learning on Source Code

The RNN Encoder-Decoder model [10] with the attention mechanism [11] is a sequence-to-sequence model [10], whose purpose is to generate a target sequence from a source sequence. It is used widely in deep learning and neural machine translation. Some studies show that software has naturalness [23], so many researchers attempted to adapt deep learning techniques to source codes. The approach DeepAPI [7] recommends API usage sequences according to given natural language queries. DeepAPI considers the natural language query and the API usage sequence as two different languages and adapts the RNN Encoder-Decoder model. As a result, it can better recognize the deep semantics of natural language queries than the traditional techniques with bags-of-words. Iyer *et al.* [24] tokenized source codes with punctuations and whitespaces to obtain the code sequence, and inputted this code sequence into the attention RNN Encoder-Decoder model to output the natural language summarization of the corresponding code snippet. To consider the code structure and semantics information, Hu *et al.* [25] parsed code snippet into an AST and transformed the AST to a sequence. They inputted the AST sequence instead of simply tokenized code snippets into the attention RNN Encoder-Decoder model.

These high-performance approaches present that source code has naturalness and many source code problems can be transformed into sequence-to-sequence problems, so some researches adapted this sequence-to-sequence model to the generation of commit messages from code changes to detect patterns and semantics of these code changes. Loyola *et al.* [9] and Jiang *et al.* [3] used the attention RNN Encoder-Decoder model for understanding the evolutionary history of code changes and generating summaries of code changes. The high performances of these approaches indicate that deep learning can learn semantics and code changes patterns. However, standard RNN Encoder-Decoder models can only generate words from a static vocabulary. Because the context-specific identifiers are not generic but specific for one project, they should be excluded from the vocabulary. As a result, these approaches cannot output these identifiers in the commit messages.

To output OOV identifiers at code completion, Li *et al.* [26] proposed a novel RNN [27] model with experiences of attention mechanism [11] and the pointer mechanism [28, 29]. To output the next word of an input code sequence, this approach mixes RNN, outputting the next word from a fixed vocabulary, and the pointer mechanism, outputting the next word from the input sequence. Meanwhile, See *et al.* proposed the pointer-generator network [29], a mixture of the attention RNN Encoder-Decoder model and the pointer mechanism, to summarize an article with a paragraph, which contains some OOV words. The high performances of these approaches inspire us to adapt the mixed sequence-to-sequence model based on the pointer-generator network to generate commit messages and enable the prediction of OOV context-specific identifiers.

III. Approach

In this section, we describe the process of generating a commit message sequence from the relevant code diffs. The first three parts of Fig. 1 show the overall workflow for our approach. It contains collecting data, processing data, and the structure of the PtrGNCMsg model, which are explained detailedly below.

A. Dataset Preparation

Fig. 2 shows the workflow for data processing with a specific example of a real commit in GitHub.

1) Collecting Raw Dataset: The raw dataset is collected from top 2,081 Java projects sorted in descending order of stars in GitHub using GitHub Developer REST API [30]. Jiang *et al.* [3] collect the top 1,000 projects, and we collect the top 1,001-2,081 projects. When developers use Git to manage projects, they change some codes (code diffs) for each commit and then write short sentences (commit message) to summarize where and why the changes occur. The code diff includes which files are changed and which lines of each file are added and removed. Each raw dataset instance contains code diffs and the corresponding commit message for a commit.

2) Preprocessing Raw Dataset: Firstly, we remove the rollback and merged commits. Rollback commits are used to revert the project to a previous commit version, and merged commits is designed to integrate changes from multiple commits into one, so these commits do not generate new information but only take another commit node in the commit tree as the latest commit. Secondly, according to commit message guidelines [31], we extract the first sentence from a commit message as the subject by breaking it into

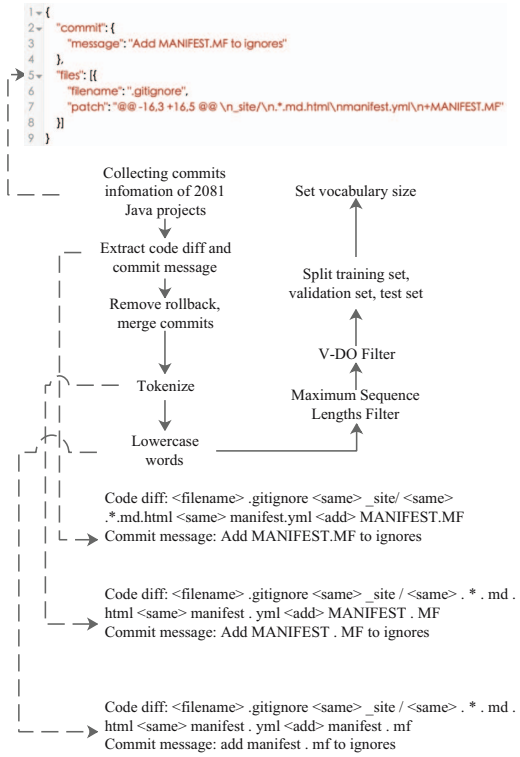

Fig. 2: A sample of the data processing workflow.

sentences with a trailing punctuation (;:,.!?) followed by a whitespace. Similarly, Gu *et al.* [7] consider the first sentence of JavaDoc as the natural summary of the API call sequence in the corresponding function. Thirdly, a code diff contains multiple lines each of which starts with +, - or other characters, representing the addition, deletion, or the same of a line compared to the previous version. Because both + and - are also identifiers in Java, we replace them with particular words <add> or <delete>, or insert <same> at the beginning of the line. Fourthly, we use punctuations and whitespaces to tokenize the sentences in code diffs and commit messages. Finally, we lowercase all words in tokenized code diffs and commit messages, which is used widely in natural language processing and deep sequence to sequence models [32, 33, 34].

3) Maximum Sequence Lengths Limitation and V-DO Filter: The lengths of the source and target sequences in sequence-to-sequence models are often set between 50 to 100 [11]. However, the majority of the manual commit messages are short summarization sentences and usually contain 20 to 30 words [3], so we set the maximum length of the source and target sequences to 100 and 30, respectively. All the data instances outside this range are discarded from the datasets.

Some manual commit messages in GitHub are poorly written, which will affect the performances of models. The study of Jiang *et al.* [35] shows that 47% commit messages start with a verb or a direct-object, so we discard all the sentences which do not begin with a verb or a direct-object (V-DO Filter). After these two filter steps, we have 32,663 pairs <code diff, commit message>.

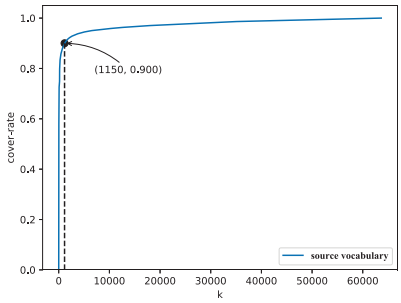

Fig. 3: The plot between k and $cover{-}rate$ on the top1,000-lowercase dataset.

4) Training Set, Validation Set, and Test set: For the final 32,663 pairs in top 1,001-2,081 projects, we randomly select around 73% pairs as the training set, 15% as the validation set and 12% pairs as the test set. For the final 32,208 pairs in top 1,000 projects, the dataset providers [3] randomly select 3,000 pairs as the validation set, 3,000 pairs as the test set, and the rest pairs as the training set.

5) Setting Vocabulary Size: Both the attention RNN Encoder-Decoder model and PtrGNCMsg require a fixed source vocabulary and target vocabulary, which are collected from the training set and the validation set. For the dataset of top 1,000 Java projects, the numbers of distinct words in code diffs and commit messages are 63,653 and 15,135. For the dataset of top 1,001-2,081 projects, there are 84,792 distinct words in code diffs, and 17,268 distinct words in commit messages. To generate all the possible commit messages, standard RNN Encoder-Decoder models have to include all distinct words in code diffs (63,653 for top1,000, 84,792 for top 1,001-2,081), which requires a lot of time and memory to train the models.

Since most of the context-specific identifiers only occur in one specific project and PtrGNCMsg can copy words from the source sentence to the target sentence, we can exclude these low-frequency words from the vocabulary for better performance and better model generalization ability. To search the smallest word candidates set to cover the most parts of sentences, we propose the cumulative function $cover{-}rate(k)$, which is the sum of the frequencies of top k words sorted in descending order of frequency divided by the sum of the frequency of all words.

$$cover{-}rate(k) = \frac{\sum_{i=1}^{k} freq(i)}{\sum_{i=1}^{N} freq(i)} \quad (1)$$

where $freq(i)$ is the frequency of the i-th word and N is the total number of words. For example, Fig. 3, the plot of k and $cover{-}rate$ on the top1,000-lowercase dataset, shows that $cover{-}rate$ rockets up from 0 to 0.9 when the value of k increases from 0 to 1,150, while $cover{-}rate$ only rises from 0.9 to 1.0, when k increases from 1,150 to 63,653.

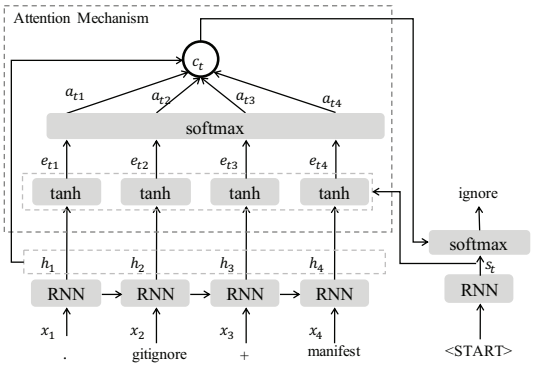

Fig. 4: The structure of the attention RNN Encoder-Decoder model.

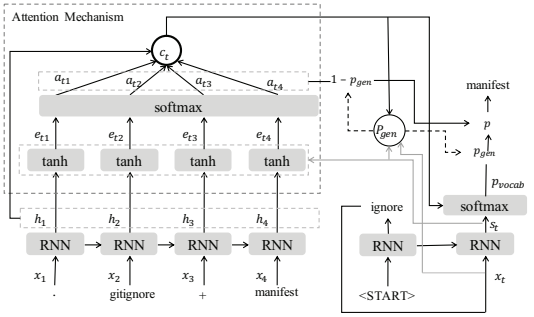

Fig. 5: The structure of PtrGNCMsg, an improvement from the standard attention RNN Encoder-Decoder model in Fig. 4.

After above steps, we obtain two different datasets: the dataset from top 1,000 Java projects with the lowercase conversion (top1,000-lowercase) and without this conversion (top1,000), and the dataset from top 1,001-2,081 Java projects with the lowercase conversion (top2,000-lowercase) and without this conversion (top2,000). The data instance is the pair <code diff, commit message> described above. The comparative experiments are conducted between the models based on these two datasets. The performance results are discussed detailedly in Section IV.

B. The Model of PtrGNCMsg

The proposed *PtrGNCMsg* is based on an improved sequence-to-sequence model using the pointer-generator network. When predicting the next word in the target sequence, besides calculating the probabilities p_{vocab} of each word in the fixed vocabulary, PtrGNCMsg also calculates the probabilities p_{copy} of copying each word from source sentence and the soft switch probability p_{gen} which decides whether generating the next word from vocabulary or copying words from source sentence. The details of these three probabilities are as follows.

1) Probabilities of Generating Words from Vocabulary: For a sentence of inputs $x = (x_1, ..., x_T)$, RNN [27] can learn the semantics of sentence and predict the next word according to previous words. RNN contains three layers, including an input layer which maps each input word x_i to a vector $\boldsymbol{x_i}$, a recurrent hidden layer which calculates current hidden state $\boldsymbol{h_i}$ after reading each word, and an output layer which calculates the probabilities of the next word from the vocabulary.

At each step i, it firstly calculates $\boldsymbol{h_i}$ according to $\boldsymbol{x_i}$ and $\boldsymbol{h_{i-1}}$ in the hidden layer:

$$\boldsymbol{h_i} = \text{sigmoid}(\boldsymbol{w^{hh}h_{i-1}} + \boldsymbol{w^{hx}x_i} + \boldsymbol{b^h}) \tag{2}$$

where $\boldsymbol{w^{hh}}$, $\boldsymbol{w^{hx}}$ and $\boldsymbol{b^h}$ are the learnable parameters.

Different from basic RNN, the purpose of PtrGNCMsg is to transfer source language sentence $x = (x_1, ..., x_{T_x})$ to target language sentence $y = (y_1, ..., y_{T_y})$, whose length T_y can be different from the length of target sentence T_x. It is composed of RNN Encoder and RNN Decoder. RNN Encoder receives the source sentence $(x_1, ..., x_{T_x})$ as input, and calculates fixed dimensional representation $\boldsymbol{c}$ of this source sentence. When predicting the word at each position in the target sentence, the importance of each word at each position in the source sentence is different. As a result, we use a different context vector $\boldsymbol{c_t}$ instead of the same context vector $\boldsymbol{c}$ in predicting the next target word for each step t according to the attention mechanism. Fig. 4 shows the overall structure of attention mechanism. To obtain the context vector $\boldsymbol{c_t}$, the model calculates the attention distribution $\boldsymbol{a_t}$ firstly:

$$\boldsymbol{e_{ti}} = \boldsymbol{v^e}\tanh(\boldsymbol{w^h h_i} + \boldsymbol{w^s s_t} + \boldsymbol{b^e}) \tag{3}$$

$$\boldsymbol{a_t} = \text{softmax}(\boldsymbol{e_t}) \tag{4}$$

where $\boldsymbol{h_i}$ is the hidden state calculated sequentially by RNN Encoder, $\boldsymbol{s_t}$ is the decoder state at each predicting step t, and $\boldsymbol{v^e}$, $\boldsymbol{w^g}$, $\boldsymbol{w^s}$ and $\boldsymbol{b^e}$ are learnable parameters. The attention distribution can be seen as the probability distribution of source words, telling the decoder the importance of each word in the source sentence in predicting the next word. Next, the context vector $\boldsymbol{c_t}$ can be calculated according to $\boldsymbol{a_t}$ and $\boldsymbol{h_i}$:

$$\boldsymbol{c_t} = \sum_{i=1}^{T_x} \boldsymbol{a_{ti} h_i} \tag{5}$$

Then the target vocabulary probability distribution to predict word y_t for source sentence x at predicting time step t can be calculated by $\boldsymbol{s_t}$ and $\boldsymbol{c_t}$:

$$p_{vocab}(y_t|x) = softmax(\boldsymbol{v_2^{p_v}}(\boldsymbol{v_1^{p_v}}[\boldsymbol{s_t}, \boldsymbol{c_t}] + \boldsymbol{b_1^{p_v}}) + \boldsymbol{b_2^{p_v}}) \tag{6}$$

where $\boldsymbol{v_1^{p_v}}$, $\boldsymbol{v_2^{p_v}}$, $\boldsymbol{b_1^{p_v}}$ and $\boldsymbol{b_2^{p_v}}$ are learnable parameters.

2) Probabilities of Copying Words from Source Sentence: When predicting the t-th word in the target sentence, the attention distribution $\boldsymbol{a_{ti}}$ presents the weight of the importance of each word x_i in the source sentence in predicting the next word. Therefore, the probability distribution for words in the source sentence at predicting step t is as follows:

$$p_{copy}(y_t|x) = \sum_{i:x_i=y_t}^{T_x} \boldsymbol{a_{ti}} \tag{7}$$

The probability of copying the word y_t in source sentence to target sentence is the sum of each a_{ti} where the word x_i is the same as the word y_t.

3) The Probability of Soft Switch: The probability of this soft switch presents the different importance of selecting a word from the vocabulary or copying a word from the source sentence as the next word. It should be determined by the source sentence, the already predicted portion of the target sentence and the next input word of the RNN Decoder, so the generation probability $\boldsymbol{p_{gen}}$ for step t is a linear combination of the context vector $\boldsymbol{c_t}$, the decoder state $\boldsymbol{s_t}$ and the decoder input $\boldsymbol{x_t}$ as shown in Fig. 5 and the equation below:

$$p_{gen} = \text{sigmoid}(\boldsymbol{w^{p_g c}c_t} + \boldsymbol{w^{p_g s}s_t} + \boldsymbol{w^{p_g x}x_t} + \boldsymbol{b^{p_g}}) \quad (8)$$

where $\boldsymbol{w^{p_g c}}$, $\boldsymbol{w^{p_g s}}$, $\boldsymbol{w^{p_g x}}$ and $\boldsymbol{b^{p_g}}$ are learnable parameters. Next, $\boldsymbol{p_{gen}}$ is used as a soft switch between the generator mode $\boldsymbol{p_{gen}}$ or the copying mode $\boldsymbol{p_{copy}}$.

For a source sentence and the corresponding target sentence, the extended vocabulary contains words in the target vocabulary and the source sentence. The extended vocabulary for each pair <code diff, commit message> is different because an extended vocabulary contains words in the corresponding code diff. After calculating p_{gen}, the final probabilities of each word in the extended vocabulary are calculated by the following equation:

$$p(y_t|x) = p_{gen}p_{vocab}(y_t|x) + (1 - p_{gen})p_{copy}(y_t|x) \quad (9)$$

If the word y_t only occurs in the pre-defined vocabulary, then the value $(1-p_{gen})$ will be 0. In contrast, if the word y_t only occurs in the source sentence, then the value p_{gen} will be 0. Otherwise, the value p_{gen} is between 0 and 1.0. The accuracy of p_{gen} determines whether the model can accurately select the mode of producing words from the source sentence or the fixed vocabulary, so that it will affect the accuracy of the final probabilities $p(y_t|x)$.

4) The Loss Function: After calculating the final distribution of the extended vocabulary, the PtrGNCMsg can be trained to minimize the negative conditional log-likelihood which is represented by the following function:

$$L = \frac{1}{N}\sum_{i=1}^{N}\sum_{t=1}^{T_y} cost_{it} \quad (10)$$

where N is the number of training instances and $cost_{it}$ is the cost function in predicting the t-th target word for the source sentence x_i with the reference target word y_t as follows:

$$cost_{it} = -log(p(y_t|x_i)) \quad (11)$$

C. Training for PtrGNCMsg

The library TensorFlow-gpu 1.9 is used to implement PtrGNCMsg.

We use different hyper-parameters to train the models and select the final hyper-parameters used by the best models. The final hyper-parameters are as follows. The embedding sizes for the source sentence and the target sentence are both 512; the number of GRU [36] layers in encoder and decoder is 4, and each layer has 512 GRU cells; the batch size is 16; we use clip-norm, and the value of *clip_norm* is 5; we randomly drop out 30% neural cells in every GRU layer when optimizing parameters of the model; we use Adam [37] optimizer algorithm with 0.001 initial learning rate; the maximum number of epochs is 1,000.

At each epoch, the training dataset is shuffled randomly and used to optimize the parameters of the model. Then the model is used to calculate the average BLEU score on the validation dataset. After each epoch, the model is saved only when the average validation BLEU score is the highest by far. If the highest BLEU score by far does not appear in the latest consecutive 10 epochs, then the training is early stopped.

We train the PtrGNCMsg in a computer with one GTX 1080 Ti GPU with 12G memory. The training lasts approximately 220 minutes and stops at approximately 45 epochs.

IV. Evaluation

In this section, we evaluate the performance of our approach PtrGNCMsg on two datasets as described in Section III-A, by measuring the accuracy of generated commit messages. There are two research questions concerned:

- RQ1: How accurate is the PtrGNCMsg model for generating commit messages compared to the standard attention sequence-to-sequence model?
- RQ2: How accurate is the PtrGNCMsg model under different *max-cover-rate*s compared to the standard attention sequence-to-sequence model?

A. Accuracy Measurements: BLEU and ROUGE

We use BLEU [14] and ROUGE [15] to measure the accuracy of generated commit messages. They have been used widely for machine translation and summarization generation. BLEU measures precision by calculating how many n-grams in the machine-generated summaries appear in the human reference summaries, while ROUGE measures recall by calculating how many n-grams in the reference summaries appear in the generated summaries.

1) BLEU Score: In general, BLEU is proposed to fix the precision of n-gram by calculating clip count as follows:

$$Cnt_{clip}(ngram) = \min(Cnt_{pred}(ngram), Cnt_{ref}(ngram)) \quad (12)$$

where Cnt_{pred} and Cnt_{ref} are the numbers of occurrences of an n-gram respectively in the predicted sentence and the reference sentence. The precision of BLEU is calculated as follows:

$$p_n = \frac{\sum_{(pred,ref)\in test}\sum_{ngram\in pred} Cnt_{clip}(ngram)}{\sum_{(pred,ref)\in test}\sum_{ngram\in pred} Cnt_{pred}(ngram)} \quad (13)$$

where $test$ is the set of pairs of predicted and reference target sentences. For each pair $(pred, ref)$, $ngram$ is the distinct n-gram phrase in the predicted sentence $pred$. The precision of the n-gram p_n becomes higher as the length of the sentence becomes shorter. To avoid this bias, BLEU introduces a length brevity penalty factor in the final score:

$$BP = \begin{cases} 1 & \text{if } len_{pred} > len_{ref} \\ e^{1-len_{ref}/len_{pred}} & \text{if } len_{pred} <= len_{ref} \end{cases} \quad (14)$$

TABLE I: BLEU and ROUGE scores of PtrGNCMsg and NMT with the source vocabulary size of 50K and target *max-cover-rate* of 1.0.

Dataset	NMT / PtrGNCMsg			
	BLEU	ROUGE-1	ROUGE-2	ROUGE-L
top1,000	37.08 / 35.37	35.34 / 35.55	26.73 / 24.65	35.09 / 35.19
top1,000-lowercase	38.18 / 37.95	36.50 / 37.88	27.37 / 26.80	36.27 / 37.43
top2,000	38.80 / 41.92	36.26 / 38.46	25.84 / 27.17	36.01 / 38.07
top2,000-lowercase	37.68 / 42.35	35.75 / 38.91	25.12 / 27.35	35.52 / 38.51

TABLE II: BLEU and ROUGE scores of PtrGNCMsg and NMT with *max-cover-rate* 0.9.

Dataset	NMT / PtrGNCMsg			
	BLEU	ROUGE-1	ROUGE-2	ROUGE-L
top1,000	33.05 / 34.21	35.54 / 34.94	25.75 / 23.55	35.38 / 34.67
top1,000-lowercase	37.00 / **39.01**	36.35 / **40.82**	26.05 / **28.91**	36.16 / **40.39**
top2,000	37.72 / 37.97	35.90 / 39.64	24.55 / 26.29	35.71 / 39.24
top2,000-lowercase	37.78 / **40.79**	35.73 / **40.50**	24.47 / **26.89**	35.48 / **40.04**

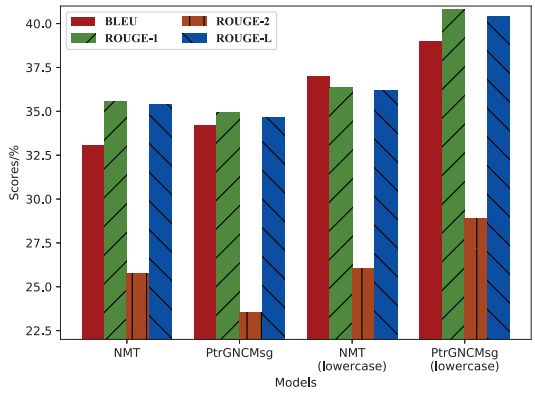

Fig. 6: BLEU and ROUGE scores of PtrGNCMsg and NMT on the top1,000 dataset with *max-cover-rate* 0.9.

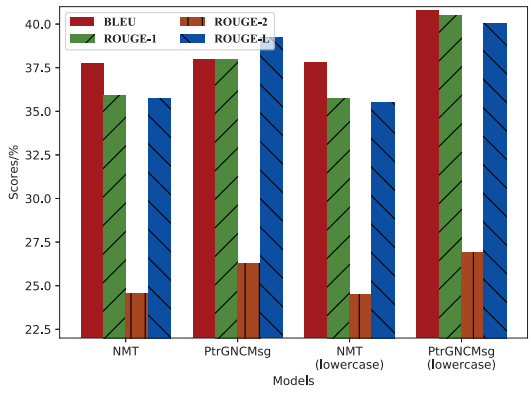

Fig. 7: BLEU and ROUGE scores of PtrGNCMsg and NMT on the top2,000 dataset with *max-cover-rate* 0.9.

where len_{pred} and len_{ref} are the lengths of the predicted and reference sentences respectively. To balance n-grams with different n, BLEU is calculated by p_n of n-grams:

$$BLEU = BP * \exp(\sum_{n=1}^{N} w_n \log p_n) \tag{15}$$

The weight of each n-gram w_n is set to $\frac{1}{N}$, and the number of n-grams N is set to 4, which is also widely used to evaluate other sequence-to-sequence problems [7, 3, 24].

2) ROUGE Score: ROUGE does not try to evaluate how fluent the summary is, but only tries to evaluate the adequacy by counting how many n-grams in the reference summaries appear in the generated summaries. The recall of ROUGE is calculated as follows:

$$ROUGE - N = \frac{\sum_{(pred,ref) \in test} \sum_{ngram \in pred} Cnt_{pred}(ngram)}{\sum_{(pred,ref) \in test} \sum_{ngram \in ref} Cnt_{ref}(ngram)} \tag{16}$$

where $ngram$ is the distinct ngram phrase in the reference sentence ref, and Cnt_{pred} and Cnt_{ref} are the numbers of occurrences of an n-gram in the predicted sentence and the reference sentence respectively. The number of n-grams N is set to 1 and 2, which is also widely used in the evaluation of other summarization problems [15, 29]. Besides, ROUGE-L is calculated by using the longest common subsequence and F-measure [38].

BLEU and ROUGE are usually expressed as a percentage value between 0 and 100. The more words and phrases in the predicted sentence appear in the reference sentence, the higher the ROUGE is. The more words and phrases in the reference sentence appear in the predicted sentence, the higher the BLEU is.

B. RQ1: Accuracy Compared to Attention Sequence-to-sequence Models

The the state-of-the-art approach adopts the attention RNN Encoder-Decoder model [3], named Neural Machine Translation (NMT). To measure the accuracy compared to it, we also train NMT as the baseline model.

We train PtrGNCMsg and NMT on two different datasets with the lowercase conversion and without this conversion,

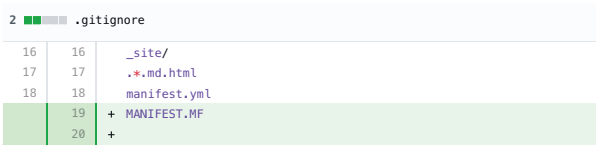

Fig. 8: This is the first test example. The figure shows the code diffs in a commit, and the corresponding reference message is "Add MANIFEST . MF to ignores". The messages generated by PtrGNCMsg and NMT are "ignore manifest . mf" and "ignore a couple of <UNK> - template link to gitignore .", respectively.

Fig. 9: This is the second test example. The figure shows the code diffs in a commit, and the reference message is "fixed a bug in ReplicatedChronicleMap.value()". The messages generated by PtrGNCMsg and NMT are "fixed a bug in replicatedchroniclemap entry" and "<UNK> <UNK> to prevent more <UNK> by <UNK> <UNK>", respectively.

with *max-cover-rate* of 0.9 on both source and target vocabularies. Then the average BLEU, ROUGE-1, ROUGE-2 and ROUGE-L are calculated on the corresponding test datasets for each model as shown in Table II, Fig. 6 and Fig. 7. The table and figures illustrates that the PtrGNCMsg gets higher BLEU and ROUGE scores than NMT on both two datasets.

ROUGE-1 and ROUGE-L are higher by 4.47 and 4.23 on the top1,000-lowercase dataset, and 4.77 and 4.56 on the top2,000-lowercase dataset. The much higher ROUGE scores indicate that the commit messages generated by PtrGNCMsg contains more words from references than NMT, which are the context-specific OOV identifiers. Fig. 8 and Fig. 9 illustrate two examples. The key words "mf" and "replicatedchroniclemap" both belong to OOV words. PtrGNCMsg successfully predicts those OOV words, while the NMT model fails to predict the words and outputs several <UNK>s.

Meanwhile, the BLEU scores of PtrGNCMsg are slightly higher than NMT (2.01 higher on the top1,000-lowercase dataset, and 3.01 higher on the top2,000-lowercase dataset). The length of commit messages generated by PtrGNCMsg is shorter than the length of commit messages generated by NMT and the length of the references. The brevity penalty of BLEU slightly reduces the BLEU scores of PtrGNCMsg. For example, the average lengths of commit messages from reference, PtrGNCMsg and NMT on the top1,000-lowercase dataset are 7.6, 6.76, and 7.2 respectively, which leads to the brevity penalty 0.88 and 0.94 for PtrGNCMsg and NMT, respectively.

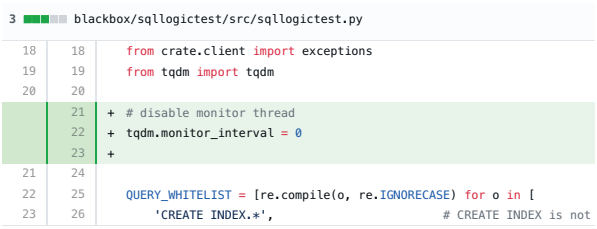

Fig. 10: This is the test example with the inaccurate prediction. The figure shows the code diffs in a commit, and the reference message is "disable tqdm monitor thread which leaked". The messages generated by PtrGNCMsg and NMT are "add disable monitor thread" and "add to <UNK>", respectively.

Secondly, the results on the lowercase datasets for each model are better. For example, the results on top1,000-lowercase are higher than top1,000 by 4.80 BLEU, 5.88 ROUGE-1, and 5.72 ROUGE-L. After lowercasing words, datasets can use the less vocabulary size to achieve the same max-cover-rate, and that models can better distinguish between same and different words, and more easily learn generic features. For example, at a *max-cover-rate* of 0.9, the source and target vocabulary sizes on the top1,000-lowercase dataset are 20% and 8% less than on the top1,000 dataset, respectively.

However, some predicted sentences perform not very well. An example is shown in Fig. 10. Because these commit messages are usually related to bug reports, issues and documents, code diffs do not contain enough information. As a result, the model can only predict "add disable monitor thread", but fails to know which thread to disable. In the future, we may let the model learn from more information resources such as project structures, code semantics, project documents, and project issues.

To compare the generalization capabilities of PtrGNCMsg and NMT, we crossly test them on the top1,000-lowercase dataset and top 2,000-lowercase dataset. We use the models trained on the top1,000-lowercase dataset with *max-cover-rate* of 0.9 to generate predicted commit messages given the code diffs on the test dataset of top2,000-lowercase, and vice versa. Table V shows some examples, where bold words indicate the OOV words. These examples show that when the words in references belong to OOV words, PtrGNCMsg generates more accurate commit messages with OOV words, while NMT usually outputs the particular word <UNK>.

To make it more comparable, we also followed the configurations used by NMT, setting the source vocabulary size to 50K and *max-cover-rate* of target vocabulary to 1.0. As shown in Table I, the BLEU and ROUGE scores of PtrGNCMsg and NMT are higher, and the scores of PtrGNCMsg are higher than NMT in most cases. However, the vocabulary with these configs includes almost all the words in the datasets, and it also contains all non-generic context-specific identifiers which are specific for one project such as "native_window_transform_hint" and "gl_invalid_conversion". It makes PtrGNCMsg and NMT difficult to learn how to generate

TABLE III: BLEU and ROUGE scores of models under different *max-cover-rate* values on top1,000-lowercase dataset.

max-cover-rate	NMT / PtrGNCMsg				Vocabulary Size	
	BLEU	ROUGE-1	ROUGE-2	ROUGE-L	Source	Target
0.80	30.49 / 36.75	32.21 / 38.45	21.72 / 26.68	32.05 / 38.08	178	574
0.85	34.47 / 37.79	34.44 / 39.28	24.20 / 27.18	34.26 / 38.89	401	972
0.90	37.00 / **39.01**	36.35 / **40.82**	26.05 / **28.91**	36.16 / **40.39**	1,149	1,886
0.95	37.55 / 38.26	36.32 / 40.44	26.77 / 27.86	36.12 / 40.02	6,517	4,958
1.00	38.33 / 37.57	37.35 / 38.18	27.89 / 27.00	37.12 / 37.75	63,653	15,135

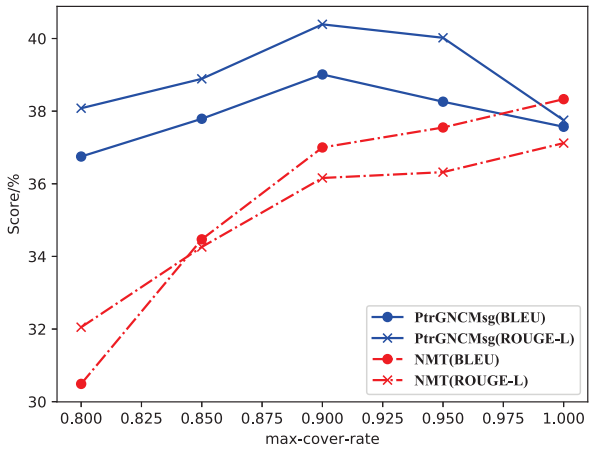

Fig. 11: BLEU and ROUGE scores of PtrGNCMsg and NMT under different *max-cover-rate* values on the top1,000-lowercase dataset.

generic words from the vocabulary that these non-generic identifiers usually do not appear in real predictions. And PtrGNCMsg has no chance to learn how to copy non-generic words from the vocabulary because the vocabulary contains all the non-generic identifiers appearing in the dataset. For example, Table IV shows that the results of cross test on the two datasets for PtrGNCMsg and NMT are not as accurate as the datasets with *max-cover-rate* of 0.9.

Besides, the training for PtrGNCMsg on the top1,000-lowercase dataset with *max-cover-rate* of 0.9 lasts approximately 220 minutes and stops at around 45 epochs, while the training for NMT on the same dataset and computer lasts approximately 20 hours and stops at around 700 epochs. The main reason is that NMT checks the validation dataset per 10,000 iterations (around 33 epochs) for early stopping while PtrGNCMsg checks per one epoch.

In conclusion, with the better capability of predictions of generic words and OOV context-specific identifiers, PtrGNCMsg gets higher BLEU and ROUGE scores than NMT on two datasets. Meanwhile, the results on the lowercase datasets are much better.

C. RQ2: Accuracy with Different Max-Cover-Rate Values

Many context-specific identifiers occur only in a specific project. If these words are in the pre-defined vocabulary, the trained model will be overfitting. Therefore, we do not select all words occurring in training and validation datasets, but set the value of *max-cover-rate* as less than 1.0. As shown in Fig. 3, $cover{-}rate$ rockets up from 0 to 0.9 when the value of k increases from 0 to 1,150, while $cover{-}rate$ only rises from 0.9 to 1.0 when k increases from 1,150 to 63,653. Therefore, we choose the different max-cover-rates from 0.8 (0.9 - 0.1) to 1.0 (0.9 + 0.1) on the top1,000-lowercase dataset to train PtrGNCMsg and NMT.

Table III and Fig. 11 show the results. As *max-cover-rate* increases from 0.8 to 1.0, BLEU and ROUGE scores of PtrGNCMsg increase until *max-cover-rate* is 0.9 and decrease since then, while the scores of NMT increase throughout the whole region. When the *max-cover-rate* is smaller (such as 0.8), there are only 178 and 574 words in the source and target vocabularies, respectively. At *max-cover-rate* 0.8, BLEU and ROUGE scores of PtrGNCMsg are slightly lower, but the scores of NMT are much lower than those of PtrGNCMsg by around 6.0 points.

When *max-cover-rate* is too large, some non-generic words will be contained in the pre-defined vocabulary, which makes PtrGNCMsg difficult to learn how to copy non-generic words from source sentences. On the contrary, some generic words will be excluded from the vocabulary when the *max-cover-rate* is too small, impeding our model learning how to generate generic words from the vocabulary. As a result, BLEU and ROUGE scores increase first and then decrease for PtrGNCMsg, and the maximum values of BLEU and ROUGE-L are 39.01 and 40.39 when *max-cover-rate* is around 0.9.

The trend of BLEU and ROUGE scores increasing all the time shows that the higher the value of *max-cover-rate* is, the easier NMT learns when and where to generate words from the vocabulary. However, the vocabulary sizes rocket up when *max-cover-rate* increases. For example, the vocabulary size at *max-cover-rate* 1.0 is much larger than that at *max-cover-rate* 0.9. Meanwhile, too large vocabulary will include non-generic words and affect the ability of generalization of models.

Results of NMT are the best when *max-cover-rate* is 1.0 with source and target vocabulary sizes of 63,653 and 15,135, accessing 38.33 BLEU and 37.12 ROUGE-L, while results of PtrGNCMsg are the best when *max-cover-rate* is 0.90 with much smaller source and target vocabulary sizes of 1,149 and 1,886, accessing higher BLEU and ROUGE-L scores (39.01 and 40.39).

In conclusion, as *max-cover-rate* is from 0.8 to 1.0, the BLEU and ROUGE scores of NMT are increasing, while the scores of PtrGNCMsg rise until *max-cover-rate* 0.9 and fall

TABLE IV: Examples of cross test on the top1,000-lowercase and top2,000-lowercase datasets with the source vocabulary size of 50K and target *max-cover-rate* of 1.0.

Reference	PtrGNCMsg	NMT
Commit messages generated from the code diffs on the top2,000-lowercase test dataset by the models trained on the top1,000-lowercase dataset		
release ' 0 . 84 . 0 '	bump version to 0 . 1	build new pre release 0 . 13 . 0
add dave yarwood to authors	add dave yarwood to the dave	add dave yarwood to authors
upgrade to gradle - pitest - plugin 1 . 1 . 10	fix test	upgrade gradle build tools
adding a missing @ override annotation	override methods	fix max stacksize on potions .
fixed typo in **codetrans** config	add link to config	fix for linux , fixes the generated osgi modules
Commit messages generated from the code diffs on the top1,000-lowercase test dataset by the models trained on the top2,000-lowercase dataset		
setting version to 1 . 0 . 133 - snapshot	23 updated actor sbt snapshot	upgraded spring security metadata to 0 . 1 . 0 - rc2
add emacs backup files to .	added eclipse emacs files to gitignore	added application files to . gitignore
add storm - 1835 to changelog . md	renamed build boot plugin to spring all plugin	updates changelog
bump up druid version to 0 . 4 . 0	upgrade to new version	changed project . version to 4 . 4 . 0 - snapshot
prepare for next development iteration	pom 7 for java version	increamented new version

TABLE V: Examples of cross test on the top1,000-lowercase and top2,000-lowercase datasets with *max-cover-rate* of 0.9.

Reference	PtrGNCMsg	NMT
Commit messages generated from the code diffs on the top2,000-lowercase test dataset by the models trained on the top1,000-lowercase dataset		
release ' 0 . **84** . 0 '	bump version to 0 . **84** . 0	set version for release .
add **dave yarwood** to authors	add **dave yarwood** to authors	add textlessUNKtextgreater to contributors
upgrade to gradle - **pitest** - plugin 1 . 1 . 10	update gradle - **pitest** - plugin to 1 . 1 . 10	revert the <UNK> plugin due to <UNK> bug . 1 . 2
adding a missing @ override annotation	add missing @ override annotation	add suppresswarnings annotation
fixed typo in **codetrans** config	fixed typo in **codetrans** - config .	fixing build comments
Commit messages generated from the code diffs on the top1,000-lowercase test dataset by the models trained on the top2,000-lowercase dataset		
setting version to 1 . 0 . **133** - snapshot	updated version to 1 . 0 . **133** - snapshot	re - mysql to next development version : 1 . 0 . 0 - rc2 - snapshot
add **emacs** backup files to .	added **emacs** backup files to . gitignore	ignoring <UNK> temp logs
add storm - **1835** to changelog . md	add storm - **1835** dump in changelog	updated changelog
bump up **druid** version to 0 . 4 . 0	updated to **druid** version 0 . 4 . 0 - snapshot	[artifactory - release] next development version
prepare for next development iteration	[maven - release - plugin] prepare for next -development iteration	[maven - release - plugin] prepare for next -development iteration

since then. With *max-cover-rate*, we eventually find the smallest word candidate set to cover the most parts of sentences. Using these lower vocabulary sizes according to *max-cover-rate*, PtrGNCMsg gets much higher BLEU and ROUGE scores than NMT.

V. Conclusion

Developers use version control systems to organize projects by committing code changes (code diffs) with a short sentence summary for each version. Commit messages are necessary for presenting the purposes and locations of code changes, and for tracking issues, features and bug reports. However, it is usually difficult for developers to write high-quality commit messages due to work pressures and the lack of time.

In this paper, we apply a deep neural language model to generate commit messages from the code diffs in code changes. Some approaches use the attention RNN Encoder-Decoder model to generate commit messages, but the generated commit messages cannot contain OOV context-specific identifiers, which commonly appear in code diffs and manual commit messages. To enable the prediction of OOV context-specific identifiers in commit message generation, we propose PtrGNCMsg, an improvement from the attention RNN Encoder-Decoder model, which learns to either copy an OOV word from the source sentence or generate a word from the fixed vocabulary. We collect data instances from the top 2,000 Java projects in GitHub and use *max-cover-rate* to find the smallest word candidate set for covering the most parts of sentences. Our experimental studies showed that our approach is more effective in commit message generation, which can generate more accurate OOV context-specific identifiers in commit messages with lower vocabulary size according to *max-cover-rate*. The code for our approach is available at https://zenodo.org/record/2542706#.XECK8C277BJ.

Some manual commit messages are generated by comprehensively considering other factors besides diffs information, such as project issues and bug reports, and it is imperfect to generate these messages only from code diffs information. In the future, we plan to take project issues and bug reports as the extra programming context into the models to further improve the work.

Acknowledgment

This research has been supported by the National Key R&D Program of China (No. 2018YFB0505000), the Shanghai Committee of Science and Technology (No. 17511107303, No. 17511110202), the National Natural Science Foundation of China (No. 61702374), the Shanghai Sailing Program (No. 17YF1420500), and the Fundamental Research Funds for the Central Universities.

REFERENCES

[1] B. De Alwis and J. Sillito, "Why are software projects moving from centralized to decentralized version control systems?" in *Proceedings of the 2009 ICSE Workshop on Cooperative and Human Aspects on Software Engineering*. IEEE Computer Society, 2009, pp. 36–39.

[2] R. P. Buse and W. R. Weimer, "Automatically documenting program changes," in *Proceedings of the IEEE/ACM international conference on Automated software engineering*. ACM, 2010, pp. 33–42.

[3] S. Jiang, A. Armaly, and C. McMillan, "Automatically generating commit messages from diffs using neural machine translation," in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*. IEEE Press, 2017, pp. 135–146.

[4] W. Maalej and H.-J. Happel, "Can development work describe itself?" in *Mining Software Repositories (MSR), 2010 7th IEEE Working Conference on*. IEEE, 2010, pp. 191–200.

[5] T. Roehm, R. Tiarks, R. Koschke, and W. Maalej, "How do professional developers comprehend software?" in *Proceedings of the 34th International Conference on Software Engineering*. IEEE Press, 2012, pp. 255–265.

[6] J. Wang, Y. Dang, H. Zhang, K. Chen, T. Xie, and D. Zhang, "Mining succinct and high-coverage api usage patterns from source code," in *Proceedings of the 10th Working Conference on Mining Software Repositories*. IEEE Press, 2013, pp. 319–328.

[7] X. Gu, H. Zhang, D. Zhang, and S. Kim, "Deep api learning," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2016, pp. 631–642.

[8] M. Linares-Vásquez, L. F. Cortés-Coy, J. Aponte, and D. Poshyvanyk, "Changescribe: A tool for automatically generating commit messages," in *Proceedings of the 37th International Conference on Software Engineering-Volume 2*. IEEE Press, 2015, pp. 709–712.

[9] P. Loyola, E. Marrese-Taylor, and Y. Matsuo, "A neural architecture for generating natural language descriptions from source code changes," *arXiv preprint arXiv:1704.04856*, 2017.

[10] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," in *Advances in neural information processing systems*, 2014, pp. 3104–3112.

[11] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," *arXiv preprint arXiv:1409.0473*, 2014.

[12] V. J. Hellendoorn and P. Devanbu, "Are deep neural networks the best choice for modeling source code?" in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, 2017, pp. 763–773.

[13] G. K. Zipf, "Human behavior and the principle of least effort: An introduction to human eoclogy," 1949.

[14] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, "Bleu: a method for automatic evaluation of machine translation," in *Proceedings of the 40th annual meeting on association for computational linguistics*. Association for Computational Linguistics, 2002, pp. 311–318.

[15] C.-Y. Lin, "Rouge: A package for automatic evaluation of summaries," *Text Summarization Branches Out*, 2004.

[16] H. A. Nguyen, T. T. Nguyen, G. Wilson Jr, A. T. Nguyen, M. Kim, and T. N. Nguyen, "A graph-based approach to api usage adaptation," in *ACM Sigplan Notices*, vol. 45, no. 10. ACM, 2010, pp. 302–321.

[17] B. Dagenais and M. P. Robillard, "Recommending adaptive changes for framework evolution," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 20, no. 4, p. 19, 2011.

[18] G. Uddin, B. Dagenais, and M. P. Robillard, "Temporal analysis of api usage concepts," in *Proceedings of the 34th International Conference on Software Engineering*. IEEE Press, 2012, pp. 804–814.

[19] S. Negara, M. Codoban, D. Dig, and R. E. Johnson, "Mining fine-grained code changes to detect unknown change patterns," in *Proceedings of the 36th International Conference on Software Engineering*. ACM, 2014, pp. 803–813.

[20] M. J. Zaki and C.-J. Hsiao, "Charm: An efficient algorithm for closed itemset mining," in *Proceedings of the 2002 SIAM international conference on data mining*. SIAM, 2002, pp. 457–473.

[21] P. Kreutzer, G. Dotzler, M. Ring, B. M. Eskofier, and M. Philippsen, "Automatic clustering of code changes," in *Proceedings of the 13th International Conference on Mining Software Repositories*. ACM, 2016, pp. 61–72.

[22] T. Molderez, R. Stevens, and C. De Roover, "Mining change histories for unknown systematic edits," in *Mining Software Repositories (MSR), 2017 IEEE/ACM 14th International Conference on*. IEEE, 2017, pp. 248–256.

[23] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu, "On the naturalness of software," in *Software Engineering (ICSE), 2012 34th International Conference on*. IEEE, 2012, pp. 837–847.

[24] S. Iyer, I. Konstas, A. Cheung, and L. Zettlemoyer, "Summarizing source code using a neural attention model," in *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, vol. 1, 2016, pp. 2073–2083.

[25] X. Hu, G. Li, X. Xia, D. Lo, and Z. Jin, "Deep code comment generation," in *Proceedings of the 26th Conference on Program Comprehension*. ACM, 2018, pp. 200–210.

[26] J. Li, Y. Wang, M. R. Lyu, and I. King, "Code completion with neural attention and pointer networks," *arXiv preprint arXiv:1711.09573*, 2017.

[27] T. Mikolov, M. Karafiát, L. Burget, J. Černockỳ, and S. Khudanpur, "Recurrent neural network based language model," in *Eleventh Annual Conference of the International Speech Communication Association*, 2010.

[28] O. Vinyals, M. Fortunato, and N. Jaitly, "Pointer networks," in *Advances in Neural Information Processing Systems*, 2015, pp. 2692–2700.

[29] A. See, P. J. Liu, and C. D. Manning, "Get to the point: Summarization with pointer-generator networks," *arXiv preprint arXiv:1704.04368*, 2017.

[30] "Github api v3 — github developer guide," https://developer.github.com/v3/, accessed: 20 December, 2018.

[31] "Gerrit / commit message guidelines," https://www.mediawiki.org/wiki/Gerrit/Commit_message_guidelines, accessed: 20 December, 2018.

[32] S. Venugopalan, M. Rohrbach, J. Donahue, R. Mooney, T. Darrell, and K. Saenko, "Sequence to sequence-video to text," in *Proceedings of the IEEE international conference on computer vision*, 2015, pp. 4534–4542.

[33] M. Rabinovich, M. Stern, and D. Klein, "Abstract syntax networks for code generation and semantic parsing," *arXiv preprint arXiv:1704.07535*, 2017.

[34] Z. Yang, Y. Yuan, Y. Wu, W. W. Cohen, and R. R. Salakhutdinov, "Review networks for caption generation," in *Advances in Neural Information Processing Systems*, 2016, pp. 2361–2369.

[35] S. Jiang and C. McMillan, "Towards automatic generation of short summaries of commits," in *Proceedings of the 25th International Conference on Program Comprehension*. IEEE Press, 2017, pp. 320–323.

[36] J. Chung, C. Gulcehre, K. Cho, and Y. Bengio, "Empirical evaluation of gated recurrent neural networks on sequence modeling," *arXiv preprint arXiv:1412.3555*, 2014.

[37] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[38] N. Chinchor, "Muc-4 evaluation metrics," in *Proceedings of the 4th conference on Message understanding*. Association for Computational Linguistics, 1992, pp. 22–29.