Context-Aware Retrieval-based Deep Commit Message Generation

HAOYE WANG, Zhejiang University, China
XIN XIA, Monash University, Australia
DAVID LO, Singapore Management University, Singapore
QIANG HE, Swinburne University of Technology, Australia
XINYU WANG, Zhejiang University, China
JOHN GRUNDY, Monash University, Australia

Commit messages recorded in version control systems contain valuable information for software development, maintenance, and comprehension. Unfortunately, developers often commit code with empty or poor quality commit messages. To address this issue, several studies have proposed approaches to generate commit messages from commit *diffs*. Recent studies make use of neural machine translation algorithms to try and translate git *diffs* into commit messages and have achieved some promising results. However, these learning-based methods tend to generate high-frequency words but ignore low-frequency ones. In addition, they suffer from exposure bias issues, which leads to a gap between training phase and testing phase.

In this paper, we propose CoRec to address the above two limitations. Specifically, we first train a context-aware encoder-decoder model which randomly selects the previous output of the decoder or the embedding vector of a ground truth word as context to make the model gradually aware of previous alignment choices. Given a *diff* for testing, the trained model is reused to retrieve the most similar *diff* from the training set. Finally, we use the retrieval *diff* to guide the probability distribution for the final generated vocabulary. Our method combines the advantages of both information retrieval and neural machine translation. We evaluate CoRec on a dataset from Liu et al. and a large-scale dataset crawled from 10k popular Java repositories in Github. Our experimental results show that CoRec significantly outperforms the state-of-the-art method NNGen by 19% on average in terms of BLEU.

ACM Reference Format:
Haoye Wang, Xin Xia, David Lo, Qiang He, Xinyu Wang, and John Grundy. 2021. Context-Aware Retrieval-based Deep Commit Message Generation. *ACM Trans. Softw. Eng. Methodol.* 1, 1 (May 2021), 29 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Version control systems (VCS) are widely used to manage changes between versions. When submitting a code change, developers can attach a *commit message* to describe the change and/or explain why the change was made [47][10]. VCS stores commit messages along with *diffs* which represent the differences between the current and previous version of the repository. Used well,

Authors' addresses: Haoye Wang, Zhejiang University, China, why_@zju.edu.cn; Xin Xia, Monash University, Melbourne, VIC, 3168, Australia, xin.xia@monash.edu; David Lo, Singapore Management University, Singapore, Singapore, davidlo@smu.edu.sg; Qiang He, Swinburne University of Technology, Australia, qhe@swin.edu.au; Xinyu Wang, Zhejiang University, China, wangxinyu@zju.edu.cn; John Grundy, Monash University, Melbourne, VIC, 3168, Australia, john.grundy@monash.edu.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2021 Copyright held by the owner/author(s). Publication rights licensed to ACM.
1049-331X/2021/5-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

these commit messages can greatly reduce the time that software developers take to understand high-level reasons behind code changes. In addition, commit messages can also provide information to better understand the evolution of software, including issues, feature additions and bug reports [28][10][14]. Consequently, high quality commit messages are essential for software development, maintenances and comprehension.

However, developers often do not accurately describe the changes in the repository with natural language. As the complexity of the code structure increases, writing high-quality commit messages is time-consuming and difficult for developers, and they often neglect to even write commit messages at all[15][44]. According to Dyer et al. [15], around 14% of the commit messages in more than 23K open-source projects hosted in SourceForge were completely empty. Thus, it is necessary and helpful to automatically generate commit messages for code changes.

Several approaches have been proposed to generate commit messages automatically. These methods can be roughly divided into three categories (see Section 8 for more detail):

Rule-based: DeltaDoc [10] uses the template "do Y Instead of Z" to generate commit messages based on the control flow of the program between code versions. ChangeScribe [36] and an approach proposed by Shen et al. [55] summarize code changes such as method additions based on specific pre-defined rules or templates. However, these approaches sometimes can not cover all the cases or explain the reasons or purposes behind code changes.

Retrieval-based: Liu et al. [39] and Huang et al. [24] utilize Information Retrieval (IR) to reuse the commit messages of similar code changes. Although Liu et al.'s method has achieved good results in terms of effectiveness and efficiency, retrieval based approaches cannot output accurate messages when there are no similar *diffs* in the corpus.

Learning-based: learning-based approaches have the ability to learn the semantic correspondence between source code and commit message and have achieved promising results. They have become a trend to solve the commit message generation problem. Jiang et al. [25], Loyola et al. [42] and Liu et al. [37] adopt learning based techniques for the commit message generation task. Liu et al. [37] made use of neural machine translation (NMT) model with the pointer-generator network to translate *diffs* into commit messages.

Retrieval-based methods reuse the commit messages of similar *diffs*, while learning-based methods can capture the semantic connection between *diff* and commit message. Although learning-based methods have achieved promising results, they prefer high-frequency words in the corpus and their generated messages are relatively less readable [6]. If a ground-truth word is rare in the training corpus, learning-based methods may generate a wrong result. Since retrieval-based methods take advantage of tokens in similar commits, the commit message produced may contain low-frequency words related to the given *diff*.

In learning-based methods, an additional problem exists, which we call *exposure bias* [52]. Most NMT models are typically trained with teacher forcing [64], in which the model generates each word conditioned on the ground truth words of the target sequence. Specifically, at training time the ground truth word is passed as the next input to the decoder while at testing time the gold target sequence is not available and hence the previous word predicted is fed as context. That is, the NMT model is never trained to learn how to predict under the conditions it may meet at testing and may not be robust for them.

In this paper, we present a novel approach named **Co**ntext-Aware **Re**trieval-based Deep **C**ommit Message Generation (CoRec), addressing these two limitations (the low-frequency word problem and *exposure bias* issue). In the training phase, we adapt a previously introduced decay sampling mechanism [71][9] that randomly selects the previous output of decoder or the ground truth vector as context to make the model gradually aware of previous alignment choices. At the beginning

of the training phase, the model selects the embedding vectors of ground truth words as context at a greater probability. As the model converges gradually, the previous outputs of the decoder, which contain information of previous alignment choices, are more frequently chosen as context. With this mechanism, CoRec can bridge the gap between training phase and testing phase. Given a *diff* for testing, we reuse the trained model to represent the given *diff* as a semantic vector and retrieve the most similar *diff* in the training set. While generating each word, CoRec decodes each retrieval *diff* and uses the retrieval information to adjust the final probability distribution over all the words in the target vocabulary. In this way, CoRec is able to incorporate the advantage of the retrieval-based methods and thereby improve the quality of the final generated commit message.

Furthermore, we have collected and built a new dataset whose scale is 10 times larger than the previous benchmark dataset [25]. We conducted a user study and experiments on both our new dataset and previous benchmark dataset. We use the state-of-the-art NNGen [39] and PtrGNCMsg [37] as baselines. The results show that our CoRec approach can significantly outperform NNGen and PtrGNCMsg by 19% and 64% on average, in terms of BLEU (an accuracy measure that is widely used to evaluate machine translation systems).

The main contributions of our work include:

- We propose a novel commit message generation approach which is the first work that combines the advantages of retrieval-based and learning-based methods for commit message generation.
- The proposed approach is capable of mitigating *exposure bias*. We are also the first one to address this problem in commit message generation.
- We provide a new, large-scale and cleaned benchmark dataset extracted from 10,000 Github repositories and make it publicly available. As far as we know, it is the largest dataset for end-to-end commit message generation.
- We carry out extensive experiments including a human evaluation, demonstrating that the proposed approach outperforms the state-of-the-art.

The rest of this paper is organized as follows. Section 2 introduces some basics of commit message and neural networks. Section 3 describes the details of proposed approach. Section 4 describes data preprocessing and the experiments. Section 5 presents our human evaluation process and its results. Section 6 shows the feedback from developers. Section 7 is the discussion about CoRec and Section 8 reviews the related work. Finally, Section 9 concludes the paper.

2 BACKGROUND

2.1 Commit, *diff*, Commit Messages

Git [1] is one of the most widely used version control systems in the world. The commits in Jiang et al.'s dataset [25] and our dataset are both extracted from Git repositories. Whenever developers submit a new version, Git will create a commit to record this change and require the developers to enter a textual message. A commit message is the summary entered by the developers to describe this version change where the change here is represented by *diff*.

Diff characterizes the difference between two versions and can be generated using the *git diff* command in Git. Git offers four *diff* algorithms, namely, Myers, Minimal, Patience, and Histogram [49]. All previous studies on the commit message generation did not consider different *diff* algorithms, and the default algorithm Myers is widely used. Therefore, the *git diff* algorithm we use in this work is the default algorithm Myers. We will explore the impact of different *diff* algorithms on the quality of commit message generation in the future. As shown in Figure 1, a commit mentioned in this work refers to the pair of a code *diff* and its corresponding commit message. Given a commit, we refer

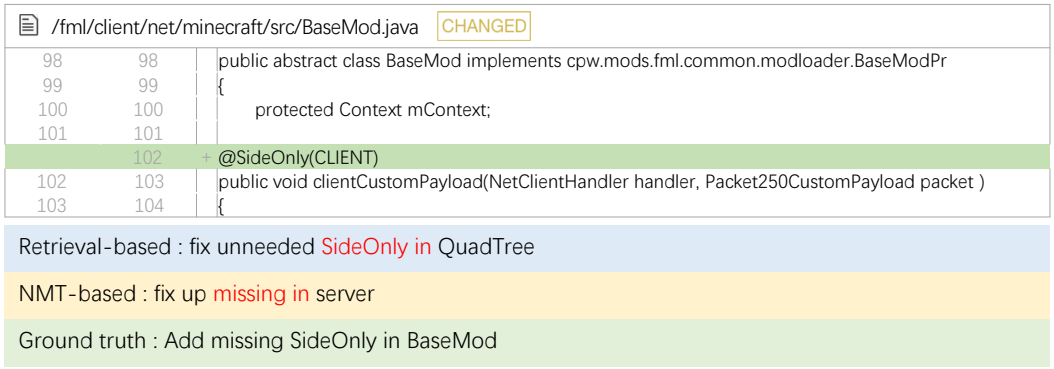

Fig. 1. Example of commit message generation showing *low-frequency word* problem

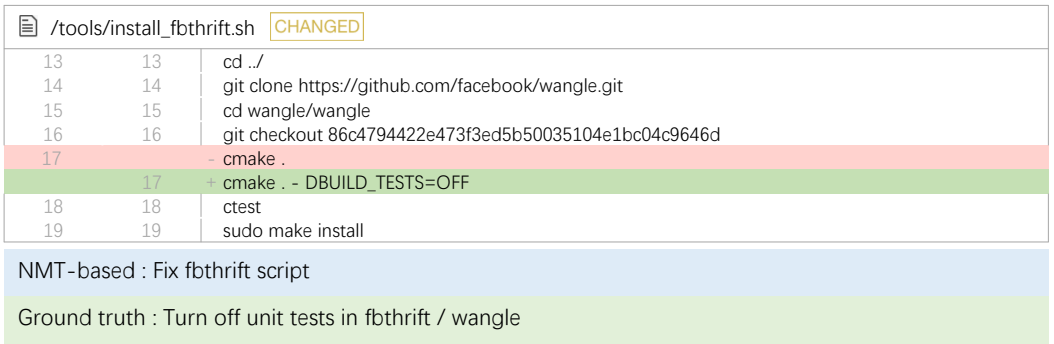

Fig. 2. Example of commit message generation showing *exposure bias* problem

to its original commit message as the *reference message*. The commit produced by retrieval-based methods is called a *retrieval commit*.

2.2 Motivating Examples

In order to explain the motivation for our work, we present two examples corresponding to the low-frequency word problem and *exposure bias* issue respectively.

Consider the code change in Figure 1. The commit message generated by a typical retrieval-based method contains the low-frequency word "SideOnly" (which is an annotation used in the Java code). The NMT-based method ignores this ground truth word but tends to generate high-frequency words ("missing" and "in"). If we use retrieved similar commits to guide the learning-based commit message generation, the generated sentences will better take into account both high-frequency and low-frequency words.

As shown in Figure 2, at training time, even if the first word actually generated is "Fix", the model still uses a ground truth word "Turn" as the context when predicting the second word. In this case, the model is trained to predict "off" conditioned on the previous word "Turn". While at testing time, given a similar *diff*, once the model predicts a wrong word "Fix", the wrong word has to be fed as context because there is no guide from the gold target sequence. Hence the errors accumulate among the sequence and cause the quality of generated sentence to decrease.

2.3 Neural Machine Translation

In Neural Machine Translation, an encoder-decoder is a widely-used structure [57][12][66]. The entire process can be simply described as: learning a model from the training data to find a target sentence $\mathbf{y}^* = \left\{y_1^*, \cdots, y_{|\mathbf{y}^*|}^*\right\}$ with the highest conditional probability according to the input source sentence $\mathbf{x} = \left\{x_1, \cdots, x_{|\mathbf{x}|}\right\}$. The encoder module of the model tries to encode the variable-length input source sequence into a fixed-length hidden state vector. Then the hidden state vector is decoded through the decoder module and finally generates the target sentence.

In early research, RNNs [12] were often used to constitute encoders and decoders due to their recurrent structure. Although RNNs are capable of learning information from the last time step, RNNs still suffer from long-term dependencies. As the length of sentence grows, information further back will become unavailable. LSTM [21] that uses a memory cell to remember important information was introduced to address this problem. The gating mechanism in the memory cell enables LSTM to use less space to remember important information and forget information that is unneeded. Sutskever et al. [57] proposed an encoder-decoder model where both the encoder and decoder consist of multi-layered LSTM. A variant of LSTM, Gated Recurrent Units (GRU) [12] has the advantage that it is computationally cheaper than LSTM.

An attention mechanism is proposed to solve the problem when input sentences are too long to compress information into hidden states. The attention can make use of all the hidden states of the encoder through an alignment function, not just the final hidden state. At each time step during decoding, the model searches the alignment relationship to extract source information (called source context vector). The source context vectors and the previous generated tokens are then used for prediction of the target sequence. The alignment weights can be calculated in different ways, e.g., dotprod or general (Luong et al. [43]) or Multi-Layer Perception (Bahdanau et al. [7]). The attention mechanism allows modeling of dependencies without regard to their distance in the input or output sequences. Attention mechanisms have become an integral part of compelling sequence modeling and transduction models for various tasks [27][59].

3 OUR APPROACH

We propose a new approach, named CoRec, to translate *git diffs* that include both code changes and non-code changes into commit messages. The overall framework is illustrated in Figure 3. The details of its model training phase and online testing phase are described in Section 3.1 and Section 3.2, respectively.

3.1 Context-Aware Encoder-Decoder Model Training

In the training phase, most NMT models use a ground truth sequence as input to the decoder. In the testing phase the commit message is generated according to the previous words generated by the models [71]. This gap between training and testing – called *exposure bias* [52] – means that the model has never been trained on its own errors if the model predicts a previous word differently from the ground truth.

To address this problem, we employ a decay sampling mechanism to randomly select the output of decoder or the ground truth content vector during the training phase. In this way, the model is trained to predict words with the previous alignment information which is contained in the output of decoder. Our model is based on the attentional Encoder-Decoder model proposed by Bahdanau et al. [7]. We also use attention to attend to the important parts of the *git diff* sequence during the generation of commit messages. On this basis, we further apply the decay sampling mechanism to the decoder part, which gives our model the ability to mitigate the *exposure bias* issue. The structure of our deep learning architecture is shown as Figure 4.

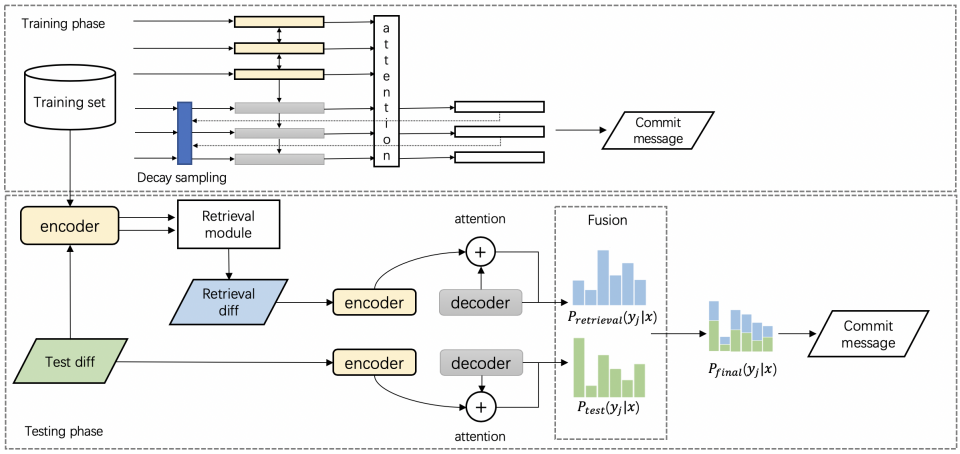

Fig. 3. The overall framework of our approach.

Assume the *diff* sequence and the ground truth commit message are $\mathbf{x} = \{x_1, \cdots, x_{|\mathbf{x}|}\}$ and $\mathbf{y} = \{y_1, \cdots, y_{|\mathbf{y}|}\}$, referring to the *diff* and reference commit message in Figure 1.

Encoder. The input of Encoder is a variable length source sequence $\mathbf{x} = \{x_1, \cdots, x_{|\mathbf{x}|}\}$, which is the *diff* part in Figure 1 and Figure 2. First, there is an embedding layer used to represent each word of input as a fixed-length vector. The Encoder takes one vector at a time until it reaches the end of sequence. We use Bidirectional LSTM (Bi-LSTM) [53] to capture the information in front and behind of the current position. At time step i, the hidden state of Bi-LSTM is computed as:

$$h_i = \left[\overrightarrow{h}_i; \overleftarrow{h}_i\right] \tag{1}$$

$$\overrightarrow{h}_i = LSTM(e_{x_i}, \overrightarrow{h}_{i-1}) \tag{2}$$

$$\overleftarrow{h}_i = LSTM(e_{x_i}, \overleftarrow{h}_{i+1}) \tag{3}$$

where e_{x_i} is the embedding vector of the word x_i.

Attention. The attention is designed to selectively focus on parts of the source sentence during translation. We use global attention proposed by Bahdanau et al. [7] in this model to extract source context vector. The source context vector is computed over the sequence of hidden states $h_1, \cdots, h_j, \cdots, h_n$ according to the formula below.

$$c_j = \sum_{i=1}^{|x|} a_{ij} h_i \tag{4}$$

where a_{ij} is the attention weights of hidden state h_i. The attention mechanism will give more weight to the hidden state vectors of important tokens. For example, the hidden state representations of "@SideOnly(CLIENT)" in Figure 1 and "DBUILD_TESTS=OFF" in Figure 2. In this way, our model can better find the changed parts of the *diff* and pay attention to the contents with similar characteristics.

At the decoding time step j, the relevance between the i-th source word and the target word y_j^* is evaluated and normalized over the source sequence.

$$a_{ij} = \frac{exp(r_{ij})}{\sum_{i'=1}^{|x|} exp(r_{i'j})} \tag{5}$$

$$r_{ij} = v_a^T tanh(W_a s_{j-1} + U_a h_i) \tag{6}$$

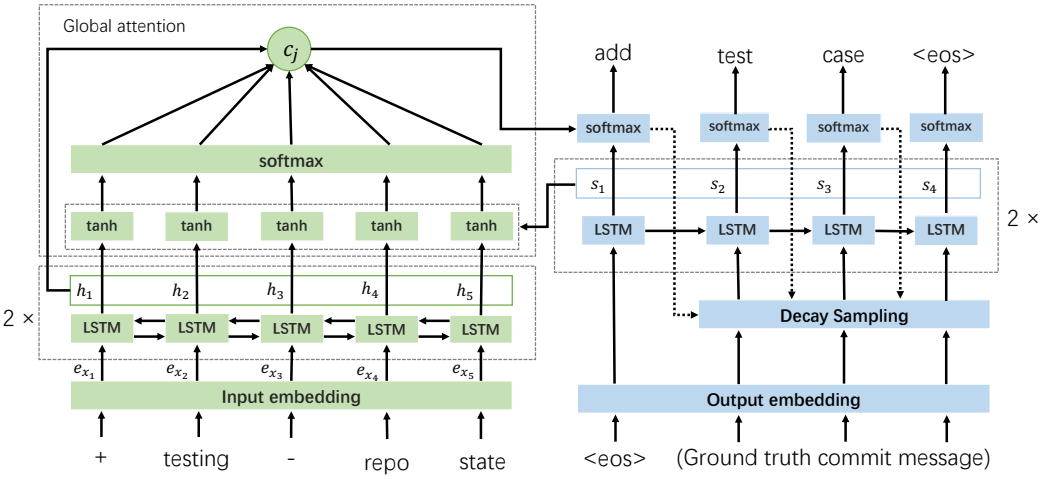

Fig. 4. The structure of Context-Aware Encoder-Decoder Model. "2×" refers to the number of encoder layers and decoder layers.

where s_{j-1} is the previous hidden state of decoder, W_a and U_a are the model parameters which will be learned during training.

Context-Aware Decoder. During training, the Context-Aware decoder is to feed as context either (i) the embedding vector of ground truth word with a probability of p or (ii) the previous output of decoder with a probability $1 - p$. At the beginning of training, as the model is not well trained, we select the embedding vector of the ground truth word as context more often to make the model converge quickly and avoid local optimum. When the training process comes to an end, the value of p becomes very small in order to make the model fully exposed to the conditions that it may confront in the testing phase. In this way the probability p of using the ground truth word vectors as context needs to decrease as the training process progresses. Similar to [71][9], we apply a mini-batch based schedule to decrease p as a function of the mini-batch index k. However, in our approach, the input of the decay sampling mechanism is different from [71][9]. We use the output of the decoder directly instead of using the generated words to embed again, thus reducing the training time. Our mechanism is also able to adapt to datasets of different sizes for the reason that the model needs more training steps to learn information in a larger dataset. Hence, the speed at which the value of p decreases needs to be adjusted according to different datasets. The probability p is computed as:

$$p = \frac{\mu}{\mu + exp(\frac{kB}{4\mu N})} \tag{7}$$

where B is the batch size of training, N is the number of commit pairs in the training data; μ is a hyper-parameter that controls the descent rate of p and the default value of 12 can cover the training process very well. This setting makes our training process gradually change from fully teacher forcing to a less guided scheme. In other words, the context-aware decoder first represents the input (e.g. the ground truth messages in Figure 1 and 2) as an embedding vector with fixed length. The decay sampling module decides whether to use the embedding vector or the previous output of decoder as the commit message information.

Our context-Aware Decoder uses LSTM to unroll the commit message information. At time step j, the hidden state of decoder is updated by:

$$s_j = LSTM(e_{y_{j-1}}, s_{j-1}) \tag{8}$$

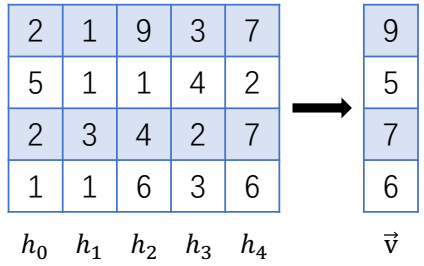

Fig. 5. Example of max pooling.

where $e_{y_{j-1}}$ is the output of the decay sampling mechanism. As shown in the Figure 4, once getting the hidden state of decoder, the probability distribution P_j for predicting next word is computed as:

$$P_j = softmax(t_j) \tag{9}$$

$$t_j = g(e_{y_{j-1}}, c_j, s_j) \tag{10}$$

where g is a linear transformation to make each target word has one corresponding dimension. The probability distribution P_j represents the probabilities that each word in the vocabulary is generated as the next word.

The goal of the network is to maximize the probability of the ground truth sequence based on maximum likelihood estimation (MLE) [5]. So the loss function below is minimized:

$$\mathcal{L}(\theta) = -\sum_{n=1}^{N}\sum_{j=1}^{|y^n|} logP_j^n\left[y_j^n\right] \tag{11}$$

where θ is the trainable parameters of the model, N is the total size of training data, $|\mathbf{y}^n|$ refers to the length of the n-th ground truth commit message, j indicates the time step, $P_j^n\left[y_j^n\right]$ is the probability of predicting ground truth word for the *diff* with the reference target word y_j^n.

3.2 Retrieval-based Commit Message Inference

After training our Context-Aware Encoder-Decoder Model, we use the *Diff* Retrieval module to retrieve relevant commits from the training set given a *diff* from validation or test set. In order to improve the performance for predicting low-frequency words, we incorporate in the knowledge of similar commits from the training set to guide the commit message generation. Considering the advantages of the neural network in capturing semantic information [46], CoRec reuses the encoder part of the model trained using the procedure described in Section 3.1 to retrieve similar commits.

Given a *git diff*, we first run forward the encoder for the trained model and get the hidden state vectors of the last layer of the Bi-LSTM. Global max pooling [29] is then applied to compress the hidden state vectors $[h_1, \cdots, h_n] \in R^{n\times 2k}$ into vector $v \in R^{1\times 2k}$ where n is the length of *git diff* and k is the hidden dimension size. Figure 5 shows an example of the max pooling operation on a sequence of hidden state vectors.

The cosine similarity is calculated between the given *git diff* with each *diff* in the training set as follows:

$$cosine(\overrightarrow{v}_{test}, \overrightarrow{v}_i) = \frac{\overrightarrow{v}_{test} \cdot \overrightarrow{v}_i}{\|\overrightarrow{v}_{test}\|\|\overrightarrow{v}_i\|}, i \in [1, N] \tag{12}$$

$$\overrightarrow{v} = maxpooling(\left[h_j^1, \cdots, h_j^{2k}\right]), j \in [1, n] \tag{13}$$

Table 1. Keywords and patterns

keywords: changelog, gitignore, readme, release, version
ignore update [*]
modify dockerfile/makefile
update submodule(s)

"[]" means optional, "/" refers to "or"

where N is the number of commit pairs in the training data. The most similar *diff* is selected for each given *git diff* according to similarity score. Due to the GPU memory limitation, it is impossible to calculate the similarities at once if the dataset is too large. Hence, we shard the test dataset and then search for the most similar *diff*s offline.

Our original intention was to use the retrieval *diff* to boost the model's performance for generating commit messages. However, there are two problems: initially the retrieval *diff* may include noisy information. Furthermore, the retrieval module may have negative effects if the similarity between retrieval *diff* and test *diff* is low. The first problem can be solved if we generate commit messages with the conditional probability of each word because the noisy words usually have low probability. So we need to combine the conditional probability of test *diff* and retrieved *diff* to generate the final commit message. To address the second problem, BLEU [51] is employed as the metric to measure the similarity. BLEU score is a popular and automated metric for evaluating the quality of machine translation. It can also be used to measure the similarity between two sentences [39]. When the retrieval *diff* and the test *diff* are not similar at all, which means there is no *diff* similar to the test *diff* in the training set, we should eliminate the effect of retrieval *diff* as much as possible.

As shown in Figure 3, given a test *diff*, CoRec first parallelly encodes the test *diff* and the retrieval *diff* by the model trained using the procedure described in Section 3.1. The hidden states of the last layer encoder are represented by H_{test} and $H_{retrieval}$ respectively. At each time step j, the context vector is calculated by Equation 4. Thus, the conditional probabilities for predicting the next word is computed according to Equations 9 and 10, denoted by $P_{test}(y_j|x)$ and $P_{retrieval}(y_j|x)$ respectively. The final probability distribution is computed as follow:

$$\begin{aligned} P_{final}(y_j|x) =& P_{test}(y_j|x)+ \\ & \lambda BLEU(\mathbf{x}_{retrieval}, \mathbf{x}_{test}) P_{retrieval}(y_j|x) \end{aligned} \tag{14}$$

where λ is a hyper-parameter which is tuned based on a validation set.

Beam search algorithm [58] is also adopted in the testing phase to generate commit message with higher quality. At each time step j, we keep top-k candidate sequences with the highest conditional probability among all combinations where k is the beam size. Until a terminator is encountered or the sentence reaches the maximum length, the top-1 sentence is output as the final target commit message.

4 EVALUATION

We evaluate the performance of our CoRec approach on the dataset described in Section 4.1 and the dataset from Liu et al. [39]. Our experiments aim to answer the following research questions:

RQ1: How does CoRec perform compared to other baselines?

RQ2: What impact does each of our main components have in our model?

RQ3: Can CoRec mitigate the low-frequency word problem?

RQ4: What is the effect of our model on *exposure bias*?

4.1 Data Preparation

Jiang et al. [25] published a dataset for commit message generation. However, they only collected top 1,000 Java projects from Github and their dataset represents only 1.75% out of 2 million commits in GitHub [39]. In order to evaluate the performance of CoRec on a large-scale dataset, we collected the top 10,000 repositories (ordered by the number of stars) which were created between January 2012 and December 2018 in Github. The commit messages and *diffs* are processed as follows.

Commit Messages. First, non-English commits are removed and the commit message is lowercased. Commit messages are parsed with the NLTK Punkt sentence tokenizer [2]. The first sentences are extracted from commit messages as the target because the first sentences are usually the summaries of the entire commit message. Second, brackets are removed from the commit messages. Github issue IDs and commit IDs are respectively replaced by "<issue_id>" and "<commit_id>" to ensure semantic integrity. Third, merge and rollback commits are removed by checking whether the commit messages begun with "merge" or "rollback". We perform this step as the *diffs* of these commits are often thousands of lines and not a suitable input for Neural Machine Translation.

Liu et al. [39] reported that 16% of Jiang et al.'s commits [25] are noisy commits. The results from Liu et al. [39] showed that these noisy commits are easy to generate and will greatly improve the evaluation metrics of generated commit messages. We also explored the effects of these noisy data (see Section 7.1 for more details) and the results also show that the noise can improve the performance of our approach. In order to ensure the quality of dataset, we used the keywords and patterns (shown in Table 1) to filter out commit messages. They are extensions of those used in Liu et al.'s study [39]. We needed to extend them as Liu et al.'s patterns cannot cover some noisy commits in our dataset.

Existing commit messages have different writing styles and some of the messages are poorly written. We want to learn to generate high quality commit messages. Thus, we apply the Verb-Direct Object (V-DO) analysis to filter out a set of relatively good-quality commit messages. Jiang et al. [26] showed that 47% of commit messages follow this pattern. The commits which do not begin with a "dobj" dependency in the Stanford CoreNLP library [45] are removed. Please note that the approaches from [25][39][37] also works on this type of commits.

Finally, we tokenize the extracted sentences with white spaces and punctuation. Extracted sentences with length of no more than 30 are kept because most hand-written commit messages are shorter than 30 tokens [25].

Diffs. We convert *diffs* into lowercase and delete the *diff* header by using regular expressions. Commit IDs in *diff* are replaced by "<commit_id>" to ensure semantic integrity. Similar to Jiang et al. [25], commits with a *diff* larger than 1MB are removed (about 24k commits in our preprocessing) to ensure that the long commits such as merged and rollback are removed. Finally, *diffs* are tokenized by white spaces and punctuation and we keep only commits with a *diff* length of no more than 100 tokens. This is because the lengths of source and target sequences in sequence-to-sequence models are often set between 50 to 100 [7][25].

Duplicated commits are removed to make the training and testing sets disjoint. We collected 10,000 repositories in Github. We used keywords in combination with patterns to filter out as much noise as possible. After preprocessing and filtering, about 63% of the commits were filtered out and we finally get 107.4k quality commits while Jiang's dataset has only 22.0k left after filtering with our script. We randomly select 90% of these commits as the training set. There are 96,704, 5,372 and 5,372 commit pairs in the training set, validation set and test set respectively. To the best of our knowledge, this is by far the largest dataset for end-to-end commit message generation. We refer to this dataset as the *top-10000 dataset* in this paper.

Liu et al. [39] created a benchmark dataset after deleting the noisy commits from Jiang et al.'s dataset [25]. The dataset from Liu et al. contains a training set, validation set and test set, which contains 22,112, 2,511 and 2,521 commit pairs respectively. We refer to this dataset as *top-1000 dataset* in this paper.

Our CoRec approach code used in our experiments and our new *top-10000 dataset* are available at http://tiny.cc/o4j4oz.

4.2 Training Details

Our CoRec approach was developed using the OpenNMT-py [31]. While training the model, we set the embedding size and the dimensions of hidden states in LSTM to be 512. The number of encoder layer and decoder layer are both set to be 2. The training batch size is set to be 32 and dropout is applied to avoid overfitting with dropout rate being 0.1. We use Adam [30] optimizer algorithm with 0.001 initial learning rate; The maximum number of training step is set to be 100k and 400k for *top-1000 dataset* and *top-10000 dataset* respectively. Correspondingly, the λ in Equation 14 is set to be 0.8 and 0.5 based on the validation set. The maximum length and minimum length for the target sentence are set to be 30 and 2 respectively. Beam size is set to be 5 by default and the maximum length of input *diff* is set to be 100. We carry out all our experiments on a Ubuntu 16.04 server with one GeForce GTX 1080 GPU 8G memory, 12 cores 3.6GHz CPU and 64GB memory.

4.3 Evaluation Metrics

We evaluate the performance of the commit message generation task with respect to three widely-used metrics: BLEU [51], METEOR [8] and ROUGE-L [35] . These metrics all measure the quality of generated comments.

BLEU is widely used to assess the quality of machine translation systems [7][12][22][50]. It is defined as the geometric mean of n-gram (for BLEU, n=1,2,3,4) matching precision scores multiplied by a length brevity penalty factor (BP). For each pair $(pred, ref)$, the length brevity penalty factor is introduced to prevent the precision bias brought by short generated sentences. The BLEU score is computed as $BLEU = BP * exp(\sum_{n=1}^{N} w_n log p_n)$, and

$$BP = \begin{cases} 1, & if \quad len_{pred} > len_{ref} \\ e^{1-len_{ref}/len_{pred}}, & if \quad len_{pred} \leq len_{ref} \end{cases}$$

where len_{pred} are the lengths of the predicted sentences , len_{ref} are the lengths of reference sentences, N = 4 and the weight of each n-gram w_n is set to $\frac{1}{N}$.

METEOR combines unigram matching precision and recall scores using harmonic mean. METEOR further employs synonym matching and is calculated as:$METEOR = \frac{10PR(1-c)}{R+9P}$, where P is the unigram precision, R is the unigram recall and c is the Penalty factor for fragmentary matches which is computed as: $c = 0.5 * (\frac{|\textbf{chunks}|}{|\textbf{unigrams}|})^3$, where |**chunks**| is the number of matched chunk and |**unigrams**| is the number of matched unigram.

ROUGE-L computes the length of the longest common subsequence between generated sentence and reference sentence. It does not try to evaluate how fluent the summary is but focuses on the recall scores. We compute these metrics using the scripts provided by Sharma et al. [54].

4.4 Baselines

Jiang et al. [25] and Liu et al. [39] use the *git diff* which contains both non-code and code changes as input to generate commit messages. Existing commit message generation approaches PtrGNCMsg [37], based on NMT, and NNGen [39], both report better performance than Jiang et al. [25]. Hence we compare our model with the following two baseline systems:

PtrGNCMsg: PtrGNCMsg is an approach based on an improved attentional ecoder-decoder model with the pointer-generator network to translate code *diffs* into commit messages. By searching the smallest identifier set with the highest probability, PtrGNCMsg enables the prediction of out-of-vocabulary (OOV) words. In the field of commit message generation, PtrGNCMsg has proven to be superior to approaches based on neural machine translation (NMT). In our experiment, we set the *max-cover-rate* (a hyper-parameter in their approach) to be 0.9 according to their paper [37].

NNGen: NNGen is the state-of-the-art approach which uses the nearest neighbor algorithm to retrieve the top-k commits from training set. The top-k commits are then compared to derive the one as target commit message with the least distance by leveraging BLEU scores to identify the nearest neighbor. On both the original data of Jiang et al. [25] and the data set with noisy commits removed [39], NNGen has far surpassed NMT in both performance and speed.

4.5 Experimental Results

Table 2. Comparison results with baseline models on different datasets

Dataset	Methods	BLEU(%)	p_1(%)	p_2(%)	p_3(%)	p_4(%)	METEOR(%)	ROUGE-L(%)
top-1000	PtrGNCMsg	12.31	27.8	15.3	11.8	10.5	11.94	26.76
	NNGen	16.41	27.6	16.8	13.4	11.7	14.04	28.60
	CoReC	**19.80**	**33.1**	**22.2**	**19.0**	**17.9**	**15.69**	**31.13**
top-10000	PtrGNCMsg	24.67	42.3	27.6	24.4	23.5	18.64	37.24
	NNGen	35.28	46.1	36.0	33.1	31.6	23.62	42.16
	CoReC	**41.26**	**55.1**	**44.9**	**44.3**	**45.4**	**26.87**	**47.20**

p_n (n = 1,2,3,4) refers to the modified n-gram precision.

4.5.1 RQ1: How does CoRec *perform compared to other baselines?* Table 2 presents the results of different methods applied to the two datasets. We can see that **retrieval-based method NNGen outperforms the NMT-based method PtrGNCMsg** on both *top-1000 dataset* and *top-10000 dataset*. The authors of PtrGNCMsg [37] conducted their experiments and demonstrated good results on the dataset from Jiang et al. [25] without removing noisy messages. However, Liu et al. [39] have reported that if we remove these noisy messages which have little information or can be automatically produced by some rule-based tools, the performences of NMT will plummet. Our experimental results confirm this result once again when these noisy messages are removed from our datasets. Although PtrGNCMsg can deal with out-of-vocabulary (OOV) words, NNGen still outperforms PtrGNCMsg by 33.3%, 17.6% and 6.9% on *top-1000 dataset* (43.0%, 26.7% and 13.2% on *top-10000 dataset*) in terms of BLEU, METEOR and ROUGE-L respectively. This phenomenon demonstrates the effectiveness of the retrieval-based method on commit message generation task.

Compared to the state-of-the-art method NNGen [39] on *top-1000 dataset*, the relative improvements of CoRec are 20.7%, 11.8% and 8.8% w.r.t. BLEU, METEOR, and ROUGE-L, respectively. As for the experiments based on *top-10000 dataset*, CoRec outperforms NNGen by 17.0%, 13.8% and 12.0% in terms of BLEU, METEOR and ROUGE-L, respectively. On average, CoRec outperforms NNGen by 19%, 13% and 10% in terms of BLEU, METEOR, and ROUGE-L respectively on the two datasets.

We can observe that **our approach achieves the best performance for all evaluation metrics**. One reason is that we retrieve the most similar commits as additional contexts to our attentional encoder-decoder model. The retrieved commits bring additional information which helps to improve performance. Also, from the table, we can find that the improvements of p_3 and p_4 (as shown

in Table 2) are especially larger. This phenomenon means that CoRec significantly improves the matching accuracy of the 3-grams and 4-grams between messages generated and gold references. CoRec introduces a decay sampling mechanism, described in Section 3.1, to eliminate the *exposure bias* between training phase and testing phase. Using this mechanism, the model is capable of learning how to predict next word even if the previous prediction is wrong. Hence, the accumulation of errors is reduced in the testing phase. Although we use the decay sampling mechanism in our approach, which introduces some randomness, the large amount of training steps weakens the influence from the randomness. We have run the model training process 5 times and the fluctuation on evaluation metrics BLEU is ±0.2%. Therefore, we believe that the performance of our method is relatively stable.

Furthermore, we find that **all the methods perform better on *top-10000 dataset***. This shows the advantages of having a larger-scale dataset. We have made our *top-10000 dataset* publicly available. Future research on the commit messages generation task will benefit from using our more comprehensive *top-10000 dataset*.

In summary, CoRec **gets higher BLEU, METEOR and ROUGE-L scores than all the baselines on the two test datasets**.

Table 3. Effectiveness of each modules in CoRec

Dataset	Methods	BLEU(%)	p_1(%)	p_2(%)	p_3(%)	p_4(%)	METEOR(%)	ROUGE-L(%)
top-1000	Retrieval	18.01	29.1	19.0	15.6	13.9	14.62	29.14
	Basic model	18.01	29.2	18.9	16.1	15.1	14.15	28.50
	CoRec$^{-Retrieval}$	19.08	**33.8**	**22.9**	**20.2**	**19.9**	14.87	29.75
	CoRec$^{-DecaySampling}$	19.23	29.9	19.5	16.4	15.0	15.19	30.22
	CoReC	**19.80**	33.1	22.2	19.0	17.9	**15.69**	**31.13**
top-10000	Retrieval	36.24	46.1	36.2	33.6	32.3	24.11	42.84
	Basic model	37.27	52.0	41.7	41.3	43.0	24.18	43.14
	CoRec$^{-Retrieval}$	40.25	54.8	44.8	**44.6**	**46.2**	26.09	45.81
	CoRec$^{-DecaySampling}$	40.14	52.1	42.0	41.0	41.8	26.12	45.66
	CoReC	**41.26**	**55.1**	**44.9**	44.3	45.4	**26.87**	**47.20**

p_n (n = 1,2,3,4) refers to the modified n-gram precision.

4.5.2 RQ2: What impact does each of our main components have in our model? We analyze the performance gain achieved due to various components of our approach by performing an ablation study. Table 3 shows these results. We first do not use our context-aware retrieval-based model, but only regard the output of the retrieval module (from Section 3.2) as the generated commits. We refer to this reduced approach as *Retrieval* in Table 3. Then we use the attentional encoder-decoder model with different modules we proposed to determine its effectiveness. In the table, the Basic model is the original model without decay sampling mechanism or retrieval module, with other settings the same as our CoRec method ; CoRec$^{-Retrieval}$ is the second variant that does not include the retrieval module from CoRec; the third variant CoRec$^{-DecaySampling}$ that does not include the decay sampling mechanism from CoRec.

Table 3 shows that **our retrieval module performance is better than NNGen on both datasets (results in Table 2)**. The reason is that our model captures the sequential information of *diff* and embeds the semantics into hidden state vectors. We can also see that CoRec$^{-Retrieval}$

Table 4. Statistics results for word frequency on different datasets

Dataset	Methods	Word frequency						
		1	2	(2, 5]	(5, 10]	(10, 20]	(20, 50]	(50, ∞)
top-1000	Basic model	295	127	226	264	312	400	353
	CoRec	418	174	274	297	316	390	352
	Increment	**41.7%**	**37.0%**	**21.2%**	**12.5%**	**1.3%**	**-2.5%**	**-0.3%**
top-10000	Basic model	670	233	411	399	502	716	1173
	CoRec	800	269	453	425	530	730	1174
	Increment	**19.4%**	**15.5%**	**10.2%**	**6.5%**	**5.6%**	**2.0%**	**0.0%**

Table 5. Performance on commits that contain low-frequency words

Datasets	Methods	BLEU(%)	$p_1(\%)$	$p_2(\%)$	$p_3(\%)$	$p_4(\%)$	METEOR(%)	ROUGE-L(%)
low-freq -1000	Basic model	18.11	31.5	18.5	15.3	14.3	13.71	25.44
	CoRec	**20.98**	**37.0**	**23.2**	**19.6**	**18.5**	**16.55**	**27.92**
low-freq -10000	Basic model	33.22	51.5	41.7	41.6	43.1	25.65	40.04
	CoRec	**36.23**	**53.4**	**43.9**	**43.6**	**44.9**	**27.47**	**42.36**

p_n (n = 1,2,3,4) refers to the modified n-gram precision.

significantly improves Basic model, indicating the usefulness of our decay sampling mechanism. Comparing CoRec$^{-DecaySampling}$ with Basic model on two datasets, **CoRec$^{-DecaySampling}$ outperforms Basic model**. This is because we introduce retrieved commits to boost the neural machine translation model. The results show that the retrieved information can indeed enhance the performance of the neural model.

The performance of CoRec$^{-Retrieval}$ and CoRec$^{-DecaySampling}$ are similar on the two datasets. For *top-1000 dataset*, CoRec$^{-Retrieval}$ performs better on 35.0% of test commits than CoRec$^{-DecaySampling}$ and 34.1% of the test commits CoRec$^{-DecaySampling}$ performs better. As for the test commits in *top-10000 dataset*, CoRec$^{-Retrieval}$ performs better on 29.6% of the data and CoRec$^{-DecaySampling}$ performs better on 27.4% of the data. We extract these parts of commits from testing sets that CoRec$^{-DecaySampling}$ and CoRec$^{-Retrieval}$ perform differently, and extract the output of our retrieval module. We manually inspected the retrieval diffs and the generated commit messages. We find that CoRec$^{-DecaySampling}$ performs better on the commits that have similar diff in the training set and those where the low-frequency words appear more in the generated messages. The similar commits in the training set can guide the generation of high-quality commit messages.

Finally, the performance of CoRec **can still be further improved when combining both the main modules**. Although the n-grams scores of CoRec$^{-Retrieval}$ and CoRec are close, CoRec outperforms CoRec$^{-Retrieval}$ in corpus level. The reason is that the lengths of commit messages generated by CoRec are closer to the reference than messages generated by CoRec$^{-Retrieval}$. Thus, the length brevity penalty factor will make the BLEU score of CoRec$^{-Retrieval}$ lower. We use the approach from Neubig et al. [48] to conduct significance tests between our approach and its variants (with one major component turned off). The Wilcoxon signed-rank test [62] at a 95% significance level showed the improvement of our approach over its variants are all statistically significant. Comparing CoRec with CoRec$^{-Retrieval}$ and CoRec$^{-DecaySampling}$, the cliff's deltas are 0.165 and 0.184 for *top-1000 dataset*, and the cliff's deltas are 0.204 and 0.232 for *top-10000 dataset*, which are non-negligible.

These experimental results verify the importance of each of the components in our proposed composite CoRec method.

4.5.3 RQ3: Can CoRec *mitigate the low-frequency word problem?* Section 1 describes how CoRec uses retrieval information to address the problem of low-frequency words. We performed experiments on the two datasets mentioned above to show that the commit messages generated by our model contain more low-frequency words. We compare our method (CoRec) with the original attention encoder-decoder model (Basic model mentioned in Section 4.5.2), and perform word frequency statistical analysis on the generated messages. First, we collect all the words in generated messages. In order to determine the probability of these words appearing in the entire corpus, we take the number of times these words appear in the training set as their frequency. Since we are focusing on low-frequency vocabulary, then we divide these frequencies into 7 groups, e.g., 1, 2, $(2, 5]$, $(5, 10]$, $(10, 20]$, $(20, 50]$ and $(50, \infty)$. In order to better display the results, we also calculate the relative percentage of CoRec vs. Basic model for each group, where *Increment = CoRec/Basic model - 1*.

Table 4 shows the statistical results for word frequency. As can be seen from the table, CoRec **clearly increases the amount of low-frequency words**, such as words with frequencies 1, 2 and $(2, 5]$. For example, for the words of frequency 1, CoRec is 41.7% and 19.4% higher than Basic model on *top-1000 dataset* and *top-10000 dataset* respectively. Please note that we do not claim that all the low-frequency words added are the correctly generated words. Another phenomenon is that the relative percentage decreases gradually with increasing frequency. This indicates that for high-frequency vocabulary, our retrieval module has little influence on them. CoRec can still capture important information for generating high-frequency vocabulary.

To further check whether CoRec can generate low-frequency words correctly for the test commits, we counted the frequency of all words in two datasets and selected the commits that contain low-frequency words (word frequency <= 5) from the testing sets. The new commit sets from *top-1000 dataset* and *top-10000 dataset* are referred as *low-freq-1000* and *low-freq-10000*. Then we run CoRec on the two testing sets and the results are shown in Table 5.

From the table, we can see that CoRec performs better than the Basic model on both *low-freq-1000* and *low-freq-10000*. Compared to the Basic model on *low-freq-1000*, the relative improvements of CoRec are 15.8%, 20.7% and 9.7% w.r.t. BLEU, METEOR and ROUGE-L, respectively. As for the *low-freq-10000*, CoRec outperforms Basic model by 9.1%, 7.1% and 5.8% in terms of BLEU, METEOR and ROUGE-L, respectively. For the reason that the test commits are only the commits that contain low-frequency words, the results indicate that our approach has better ability to put the low-frequency words on the correct commits.

In summary, these results show that **our proposed** CoRec **can address the low-frequency word problem** while generating commit messages.

4.5.4 RQ4: What is the effect of our model on exposure bias? We introduce the context-aware model to eliminate *exposure bias* between training phase and testing phase. We analyze whether our trained models are fully exposed to the conditions encountered during testing and what is the effect on *exposure bias*. Section 3.1 describes how at the beginning of training, as the model is not well trained, the value of p is close to 1. As the model converges gradually, p gradually decreases and finally tends to 0. Hence, in our experimental settings, the maximum number of training steps is set to be 100k and 400k for *top-1000 dataset* and *top-10000 dataset* respectively. In this way, the value of p in Equation 7 will gradually change from 1 to 0 in our training settings. The training process changes from a fully guided scheme towards a less guided scheme.

In Table 3, our model CoRec **outperforms the CoRec$^{-DecaySampling}$ on both datasets.** The n-grams scores of our approach have been improved by 10.7%, 12.1%, 15.8% and 19.3% in terms of p_1, p_2, p_3 and p_4 for *top-1000 dataset*. The corresponding improvements on *top-10000 dataset* are 5.8%, 6.9%, 8.0% and 8.6% in terms of p_1, p_2, p_3 and p_4 respectively. As indicated by the improvements of

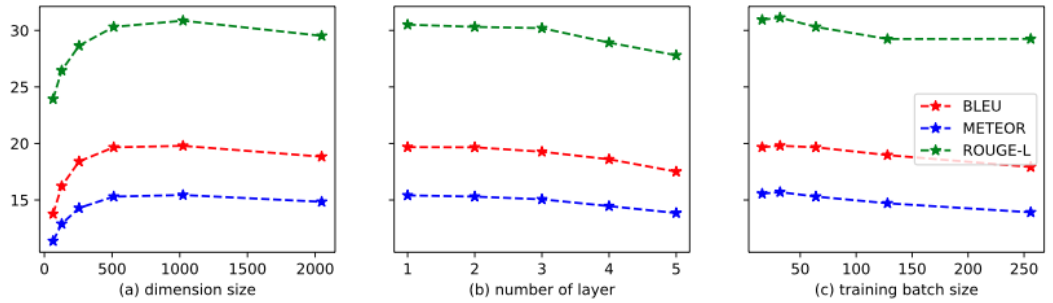

Fig. 6. The performance of CoRec under different parameter settings.

p_3 and p_4, the decay sampling mechanism does reduce the probability of consecutive word errors in the generated commit message by making model aware of the previous alignment choices. When we compare CoRec$^{-Retrieval}$ and Basic model in Table 3, a similar phenomenon also appears. It indicates that there are more ground truth words in the commit messages generated by approaches with decay sampling mechanism. However, compare CoRec with CoRec$^{-Retrieval}$, although the retrieval module brings improvement on the overall performance, the difference of n-grams scores is not obvious. We then count the words in generated messages which hit the ground truth words from the references, which we refer to as η. For the *top-1000 dataset*, the values of η are 5,867 and 5,493 for CoRec and CoRec$^{-DecaySampling}$ respectively. For the *top-10000 dataset*, the values of η are 19,781 and 18,062 for CoRec and CoRec$^{-DecaySampling}$ respectively. The improvements are $(5867/5493) - 1 = 6.8\%$ and $(19781/18062) - 1 = 9.5\%$. With the help of decay sampling mechanism, our approach has a better ability to predict the correct words. Even if the previous generated word is wrong, CoRec has been trained to predict in this situation. We can see that the models gain the ability to resolve errors that occur during testing.

4.6 Parameter Sensitivity

In this section, we study the effect of parameters on our proposed approach. The parameters involved include the embedding size and the dimensions of hidden states in LSTM k, the number of encoder layer and decoder layer l, and the training batch size B. We run our experiments on the *top-1000 dataset* and Figure 6 shows the results. We only report the BLEU, METEOR and ROUGE-L scores in the figure. In Figure 6(a), the BLEU, METEOR and ROUGE-L scores increase when the dimension size grows from 64 to 1024. However, when k grows from 512 to 1024, only a slight performance gain is observed. Considering the time cost, we fix the embedding size and the dimensions of hidden states as $k = 512$ in this work. Figure 6(b) shows that the BLEU, METEOR and ROUGE-L scores stay stable when the number of encoder layer and decoder layer varies from 1 to 3, and these scores decrease when we continue to increase the number of layers. We set the number of encoder layer and decoder layer l to 2. Figure 6(c) shows that when the training batch size is set to 32, the BLEU, METEOR and ROUGE-L scores reach the best values. As we continue to increase the training batch size, these scores gradually decrease. In this work, we fix the training batch size as 32.

5 HUMAN EVALUATION

Evaluation metrics we used in the experiments above calculate the textual similarity between reference and generated messages. However, since the generated message is not always perfect, we need to conduct a human evaluation to further evaluate CoRec's usefulness in practice. We invited

Table 6. Mean of Relevance, Usefulness and Content-Adequacy of different commit messages

	Relevance	Usefulness	Content-Adequacy
Reference message	4.2167	3.9833	3.7500
PtrGNCMsg	2.9533***	2.5633***	2.4433***
NNGen	2.8167***	2.4567***	2.3600***
CoRec	**3.4100**	**3.0967**	**2.9800**

***p-value<0.001

Table 7. The Fleiss Kappa values for the user study

	Relevance	Usefulness	Content-Adequacy
Reference message	48.33%	47.33%	41.33%
PtrGNCMsg	38.67%	46.67%	42.33%
NNGen	52.67%	64.67%	62.67%
CoRec	50.33%	49.67%	57.33%

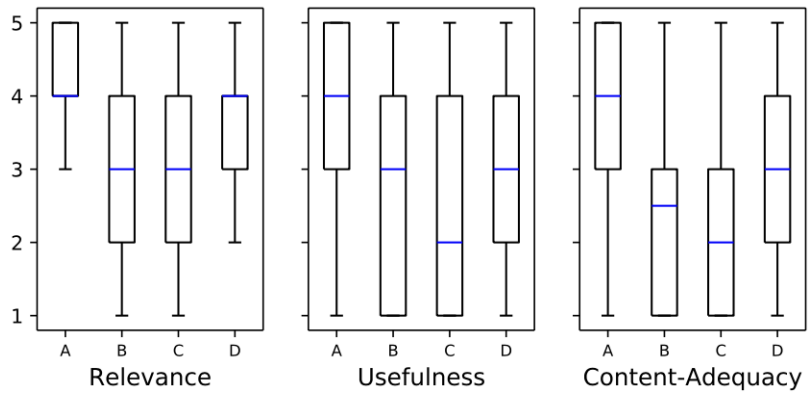

Fig. 7. Box plots of the Relevance, Usefulness and Content-Adequacy of different approaches, e.g., (A) Reference message, (B) PtrGNCMsg, (C) NNGen, (D) CoRec.

6 Ph.D. students who major in computer science to participate in our study. None are co-authors of this paper and all have industrial experience in Java programming ranging from 2 to 5 years.

We randomly choose 100 commits from the testing sets (50 from *top-1000 dataset* and 50 from *top-10000 dataset*) and their commit messages generated by PtrGNCMsg, NNGen and CoRec respectively. In order to prevent participants from being impatient when scoring because the questionnaire is too long, participants were divided into two groups, scoring the commits from *top-1000 dataset* and *top-10000 dataset* respectively. Each commit group is evaluated by 3 participants. In our study, we asked participants a number of questions; each question first presents the code changes of one commit, i.e., its *diff*, its reference message, and messages produced by PtrGNCMsg, NNGen and CoRec respectively. The participants were allowed to search for information they needed on the Internet.

Note that participants do not know which message is the reference message nor from which method each generated message comes from. The examples in our questionnaire are shuffled. Similar with Xu et al. [67], each participant was asked to score the four commit messages from three aspects, i.e., *relevance*, *usefulness* and *content-adequacy*. *Relevance* refers to how relevant the commit message is to the *diff*. Some commit messages (like "update xxx.java") contain little information even though they correctly describe the code changes in some ways. These commit messages are of little value for developers to understand the intention behind the code changes. *Usefulness* refers to how useful the commit message is for understanding the *diff*. *Content-adequacy* refers to how comprehensive the commit message is. The score ranges from 1 to 5. Score 1 means "irrelevant/useless/incomplete" and 5 means "highly relevant/useful/comprehensive". In this way, the ground truth and the automatically-generated messages from different approaches are evaluated in isolation.

Table 6 shows the mean of relevance, usefulness and content-adequacy scores of the ground truth and the generated messages from different approaches, and Figure 7 presents in box plots these scores of each commit message. We can see that **compared to the baselines,** CoRec **achieves the best performance in all three aspects**. The mean scores of reference commit messages are 4.2167, 3.9833 and 3.7500 respectively. This indicates that the quality of some of developers' commit messages need to be improved. Although NNGen can achieve better performance than PtrGNCMsg in automatic evaluation metrics (Section 4.5.1), the scores of NNGen in the three aspects are comparable to or even slightly lower than PtrGNCMsg. The reason is that some commit messages produced by NNGen and the ground truth may have some words matched but the semantic relevance is low. This condition reflects the shortcoming of the use of automatic evaluation metrics. Combining the advantages of retrieval-based and learning-based methods for commit message generation, our approach CoRec gets scores of 3.4100, 3.0967 and 2.9800 in terms of relevance, usefulness and content-adequacy, which is the closest to the reference message.

We also conducted a Wilcoxon signed-rank test [62] with a Bonferroni correction [4] to evaluate whether the differences between CoRec and the two baselines is statistically significant. The improvements of our approach over both NNGen and PtrGNCMsg are statistically significant on all three aspects at the confidence level of 99.9%. This indicates that CoRec statistically significantly improves the quality of the generated messages. We further use the Fleiss Kappa [17] to measure the agreement between participants. Table 7 shows the results. From the table, we can see that the Fleiss Kappa value of PtrGNCMsg for relevance indicates a fair agreement. The Fleiss Kappa values of NNGen for Usefulness and Content-Adequacy indicate substantial agreement. Most of the results in table 7 indicate a moderate agreement between participants.

In summary, our human evaluation shows that CoRec outperforms the state-of-the-art approach.

6 DEVELOPER FEEDBACK

To gain insights into whether our approach can help developers write well-commented changes, we used our approach to generate commit messages for a real-world scenario and then invited developers to participate in our survey. We further updated all of the top 10,000 repositories which we describe in Section 4.1 and collected all the commits submitted between December 20, 2019 and September 8, 2020. These commits are preprocessed according to the steps in Section 4.1. In the end we collected 2,656 commits from different open source repositories submitted by 1,154 developers. We then sent our survey to these developers and asked for their comments and suggestions about our approach. Each developer received an email including the list of commits submitted by the developer and the corresponding commit messages generated by our approach. Our survey includes the following questions: 1) Are you satisfied with the commit messages generated by our tool? 2) Do you think that our tool can help you write commit messages when you submit code? 3)

Table 8. Some comments from respondents

Comments
I think your tool would be useful as a auto-fill suggestion commit message tool–similar to how Gmail does it when composing Emails. I would like it to help aid me in writing meaningful commit messages by suggesting me related words.
Yes. It's better than my original messages. I think the tool is more valuable when I reviewing commits.
I'm satisfied with the generated commit messages. I'd need to see a wide range of changes and commit messages to say for certain. You should integrate with popular git clients (Source Tree, GitExtensions, and so on) in my opinion.
I give it at least 7 of 10 :) Yes I think your tool can be definitely be useful specially as the baseline for more human tweaking. Hopefully your work can become integrated into 'git commit' command someday.
Not bad. Automatic generation of commit messages might be useful for automatic cleanups (e.g. linters and the like) other automatic program transformations or automatic suggestion of patches and pull requests. It might also be useful in scenarios where commits are rewritten anyway (e.g. rebasing or migration from one VCS to another). For example, if you were migrating from SVN to Git, you could have a tool that recognizes missing or useless commit messages and suggests better ones.
I think I prefer my messages. Commit messages, especially on pull requests, are communication from one person to another. I wouldn't expect an ML project to guess what I was going to say with any accuracy. Commit messages are a) human-to-human communication and b) best used to explain why something has changed rather than what. Neither of those things are at all amenable to machine learning.
I think they were just as good as the poor msgs I provided. I am not sure these were good samples to test against. There were other changes that could have been used for additional details that were not included by my human commit nor the machine generated commit. Probably, at least for some projects. I am not sure this would be useful in a more structured development environment primarily because many of these msgs are based on the ticket descriptions.

Could you please give us some suggestions or comments for this tool? The participants could provide "yes" or "no" to the questions 1 and 2. The third question is a free-form textual question. We received 94 replies and the response rate is 8.1%, which is similar to some previous studies in software engineering [11, 60, 61]. Some representative comments are listed in Table 8.

In summary after analysing all of the responses received, we have the following findings:

- 68% of the respondents were satisfied with the commit messages generated by our tool for their commits.
- 73% of the respondents agreed or strongly agreed that our tool is helpful for writing commit messages. Even though some of the respondents were not very satisfied with our generated results, they still thought our tool was helpful to some extent.

Several respondents think that the generated commit messages hit some keywords, but their fluency and semantics were problematic. Many respondents also ask for a wide range of code changes and commit messages to investigate the performance of our tool further. Generating more fluent and natural commit message is also to be the direction of our future efforts. As mentioned in comment 6, a small number of respondents think that commit messages should communicate the intent of the change to the reader, while they do not think a tool can deduce the purpose in the general case.

Although some respondents indicated that the commit messages we generated were not as good as their own, they still expressed their hope that we can integrate this tool into IDE (Integrated Development Environment) or Git GUI client (like the comments 3 and 4 in Table 8). They think if the tool can be used as a suggestion for developers to give a draft commit message they can then modify or rewrite when they commit code changes, it will help people write better commit messages. As mentioned in comment 5, some respondents also suggested that our tool might be useful for automatic cleanups (e.g. linters), automatic program transformations or automatic suggestion of patches and pull requests.

This survey illustrates the promising usability of our tool in real-world development scenarios. 73% of the developers responded think that the automated commit message generation tool is useful and expect to use it in their daily development. Their suggestions also give us a lot of inspiration and direction for future work.

7 DISCUSSION

7.1 Influence of Data Filtering

In our data processing, there are mainly two kinds of filtering: (1) filter noisy commits (2) filter out commit messages that do not follow V-DO pattern. In previous work from Liu et al. [39], the experimental results indicated that Jiang et al.'s dataset [25] contains a lot of noisy commits. For example, noisy commits like "bump version xxx" or "ignore update xxx" often contain little information. These commits are of little value and the messages can be generated through rule-based methods. During data processing (see Section 4.1), we extend the patterns form Liu et al. [39] to filter out the noisy commits (shown in Table 1). The purpose of V-DO filtering is to get relatively high quality commit messages. In this way, the quality of messages we collect and train our approach with will be higher. Although previous works from Jiang et al. [25] and Liu et al. [39] have studied the influence of these low quality commit messages, the effects their removal has on CoRec's performance are not clear. In this section, we study the influence of these filtering on CoRec.

We refer to the dataset from Jiang et al. [25] which contains noisy commits as the *noisy-top-1000*. For the commits from top 10,000 repositories in Github, we do not remove the noisy data during the preprocessing process to make it more realistic. The dataset is re-splitted and 90% of these data are selected as the training set. We refer to this new dataset as *noisy-top-10000*. Since we do not have the original commits of Jiang et al.'s dataset without V-DO filtering, we only use the commits without V-DO filtering from our top 10,000 repositories as the dataset. Other processes are the same as described in Section 4.1. We refer to this new dataset as *no-vdo-top-10000*. Then we retrain our approach and run the testing phase on these new datasets.

The experimental results are presented in Table 9. Comparing the third and fifth rows in Table 9, the result shows the influence of V-DO filtering on Corec for our *top-10000 dataset*. If we do not apply the V-DO filtering on our *top-10000 dataset*, the performance of Corec will decrease by 28%, 26% and 21% in terms of BLEU, METEOR and ROUGE-L, respectively. In order to obtain a set of commit messages that are in a similar format, we apply the V-DO pattern. However, the removed commit messages that do not follow the V-DO also include some commits which are difficult for our approach to handle with.

From the table we can see that if we do not remove the noisy data in the dataset provided by Jiang et al. [25], the performance of Corec will increase by 108%, 73% and 34% in terms of BLEU, METEOR and ROUGE-L, respectively. For our *top-10000 dataset*, the existence of noisy data in the dataset will improve the performance of Corec by 56%, 58% and 43% in terms of BLEU, METEOR and ROUGE-L, respectively. The same as what has been reported by Liu et al. [39], the performance of Corec declines by a large amount when the noisy commits are removed. It seems that these

Table 9. The influence of noisy data on CoRec.

Datasets	BLEU(%)	$p_1(\%)$	$p_2(\%)$	$p_3(\%)$	$p_4(\%)$	METEOR(%)	ROUGE-L(%)
top-1000	19.80	33.1	22.2	19.0	17.9	15.69	31.13
noisy-top-1000	41.15	50.2	43.3	43.1	45.0	27.07	41.57
top-10000	41.26	55.1	44.9	44.3	45.4	26.87	47.20
noisy-top-10000	64.30	73.6	67.8	67.5	68.3	42.40	67.41
no-vdo-top-10000	29.58	46.8	35.3	34.1	34.4	19.90	37.24

p_n (n = 1,2,3,4) refers to the modified n-gram precision.

Table 10. Some examples of automatically-generated messages

	Diff	Generated messages
Case 1	— / dev / null +++ b / cname + selenide . org \no newline at end of file	**Reference:** added cname file for selenide . org **PtrGNCMsg:** create cname **NNGen:** added cname file **CoRec:** added cname file for handling selenide . org

Case 2	— a / build . gradle
	+++ b / build . gradle
	wrapper {
	}
	ext {
	- dropwizard = ' 1 . 0 . 0 '
	+ dropwizard = ' 1 . 0 . 2 '
	guice = ' 4 . 1 . 0 '
	}
		Reference: update dropwizard to 1 . 0 . 2
		
		PtrGNCMsg: update support library
		
		NNGen: update dw to 0 . 9 . 1
		
		CoRec: update dropwizard 1 . 0 . 2

Case 3	— a /binding/src/main/java/ch/cyberduck/binding/application/nsdatepicker.java
	+++ b /binding/src/main/java/ch/cyberduck/binding/application/nsdatepicker.java
	public abstract class nsdatepicker extends nscontrol {
	}
	public abstract nsdate datevalue ( ) ;
	+
	+ public abstract void setdatevalue (nsdate value) ;
	}
		Reference: add setter .
		
		PtrGNCMsg: add setdatevalue setdatevalue nsdate nsdate nsdate nsdate value) . . .
		
		NNGen: added abstracttrackprovider <issue_id>( )
		
		CoRec: add setter .

Case 4	— a /LeonidasExamples/project.properties
	+++ b /LeonidasExamples/project.properties
	# Project target .
	target = android - 17
	- android.library.reference.1 = ../ParticleSystemView
	+ android.library.reference.1 = ../LeonidasLib
		Reference: changing name of the lib
		
		PtrGNCMsg: fixed project . properties
		
		NNGen: add ABS as dependency to library (fix # 559)
		
		CoRec: added project . properties to the last project

noisy commits are easy for our approach to generate and it makes little sense to learn and generate these kinds of commit messages through learning-based methods. To mitigate the influence of these noisy data, we combine the keywords and patterns to filter out noisy commits which contain little information. We clean up our *top-10000 dataset* carefully and keep our experimental results as far away from the influence of noise as much as possible.

Table 11. The performance of CoRec on dataset split by project.

Dataset	Methods	BLEU(%)	p_1(%)	p_2(%)	p_3(%)	p_4(%)	METEOR(%)	ROUGE-L(%)
top-10000-sbp	PtrGNCMsg	15.34	31.3	16.1	12.8	11.9	14.25	29.93
	NNGen	29.42	35.8	28.3	27.4	27.1	19.26	34.86
	CoReC	**32.87**	**44.8**	**35.7**	**35.7**	**37.6**	**21.66**	**39.24**

p_n (n = 1,2,3,4) refers to the modified n-gram precision.

7.2 Inter Project Performance of CoRec

LeClair et al. [33] reported that the randomly split datasets will significantly improve the BLEU scores in code summarization task. This is because the training set and test set are likely to have some similar commits from the same projects. This can inflate the BLEU score and give a misleading performance indication the algorithms. In this subsection, we discuss an investigation into inter project performance of CoRec and its baselines.

Different from common general practices to date, we randomly divide the collected projects into training/validation/test sets. 90% of our top-10000 projects are randomly selected as the training projects. The commits extracted from i.e. training projects are placed into the training set. We infer this new dataset as *top-10000-sbp*. The CoRec and the baseline approaches are retrained and tested with the *top-10000-sbp* dataset. The results of each approach are presented in Table 11.

Compared with the results of *top-10000* from Section 4.5.1, we can see that all the approaches perform better on the *top-10000* dataset. For the *top-10000-sbp* dataset, the BLEU scores drop 38%, 18% and 20% in terms of PtrGNCMsg, NNGen and CoRec respectively. Nevertheless, CoRec still performs better than the baseline approaches for all of the evaluation metrics. Since our approach makes use of the similar commits in the training set, the decline in performance is predictable. The experimental results are similar to the findings from LeClair et al. [33] when the dataset is split by project. If trained on intra-project, our approach performs well when generating commit messages for the same project. But for inter project commit message generation it performs much less well.

7.3 Strengths and Limitations of CoReC

The previous methods for automatically generating commit message from *git diff* fall into three main categories, e.g., rule based, retrieval based and deep learning based. Although the state-of-the-art method NNGen [39] has achieved good results on Jiang's dataset [25], it will perform poorly if there is no similar commit in the dataset. In recent years, Neural Machine Translation (NMT) has been more often used to accomplish this task. However this NMT-based method is more inclined to generate high-frequency words and ignore low-frequency words. In addition, the discrepancy between training and testing, which we named *exposure bias*, still exits in NMT-based methods. These two critical issues provides us the motivation for further research.

We propose the CoRec (Context-Aware Retrieval-based Deep Commit Message Generation) to address the aforementioned limitations. We manually examined some examples from generated messages to explain the better performance of CoRec. We randomly choose some generated commit messages and selected four representative examples to show in the qualitative results (see Table 10).

In case 1, the reference commit message is "added cname file for selenide . org", which contains the low-frequency word "selenide". The NMT-based method PtrGNCMsg prefers to capture high-frequency words, which generates "create cname", but ignore the low-frequency words. However, with the guide of retrieval module, CoRec generates "added cname file for handling selenide . org", which takes into account both low-frequency and high-frequency words. Case 2 also shows

the benefit of the retrieval module in CoRec. Obviously, for messages containing low-frequency words, CoRec can significantly improve its generation quality. The existence of a decay sampling mechanism enables our model to have the ability to predict the gold word even if the previous prediction is wrong. With the help of our retrieval module, CoRec can handle the commits with similar diff in our training set well. In many cases, the commit messages generated by CoRec are exactly the same as the reference. Thus CoRec significantly improves the quality of the generated commit message compared to baselines. However, there are still some commits (like case 4) which are difficult for CoRec to deal with. Our proposed approach has the potential to focus on the wrong content in *diffs* and thus generates low-quality messages.

Our approach has limitations. A major one is that CoRec can not deal with long *diffs* well, which is also a major limitation of other learning based methods. We will continue our research in the future to overcome this difficulty.

CoRec needs to spend extra time to search for retrieval commits and this extra time will increase as the data set grows. However, due to the fast computation of the cosine similarity, CoRec only takes 10ms on average for retrieving a similar commit. The time consumption mainly lies in the training process of our approach. It takes 13 hours and 5 minutes to train CoRec on the *top-10000 dataset*. For the *top-1000 dataset*, the training time of CoRec is 2 hours and 57 minutes. However, we can train our model offline in advance, so the time consumption of testing is vital. In the testing phase, the average time taken to generate a message for a new coming commit is 15ms, which is very acceptable for practical usage.

In addition, retrieval commits will re-weight the probability distributions generated by our approach and potentially lead to ungrammatical sentences. In order to investigate how often does CoRec produce an ungrammatical message because of retrieval commits. We randomly sample 200 test commits (100 commits from top-1000 test set and 100 commits from top-10000 test set) and use CoRec and CoRec$^{-Retrieval}$ to generate commit messages. Then we extract the messages from the samples that two approaches generate differently (17 from the top-1000 test dataset and 23 from the top-10000 test dataset) and ask three participants to judge whether the messages are grammatically correct. Each of the participants passed CET6 (College English Test Band-6 [3]) and got a score of more than 500 points. On average, there are only 3.0 commit messages for *top-1000 dataset* and 2.0 commit messages for *top-10000 dataset* that change from grammatically correct to grammatically incorrect. Although there is influence from the retrieval commits, the frequency that CoRec generates ungrammatical messages because of this re-weighting is very low.

Another limitation is that CoRec needs extra memory for retrieval information calculation at testing time, which may cause excessive load on a GPU-based platform. To address this we suggest reducing the mini-batch size appropriately in the testing phase. We still need to conduct further user studies on the appropriateness of CoRec in industrial settings.

7.4 Threats to Validity

7.4.1 Internal Validity. Threats to internal validity are related to potential errors in the code implementation and experimental settings. We directly used the public code of PtrGNCMsg [37], and set *max-cover-rate* to 0.9 according to the paper. We first performed experiments on Jiang's dataset [25], and the results are consistent with those in their paper. However, we are not sure whether this *max-cover-rate* is suitable for our experimental datasets.

7.4.2 External Validity. The main external validity threat relates to the quality and generalizability of our dataset. The first key threat to validity is that the retrieval commits may not always have high similarities. Although we take into account the similarity in Equation 14, when the dataset is small, the impact of this factor will be amplified. Threats to external validity relate to the generalizability

of our results. The dataset from Jiang et al. [25] only includes 1k java projects from Github. To mitigate this threat, we have provided a large-scale dataset which is 10 times larger than Jiang's dataset. The way the dataset was constructed is also related to the threat. Previous work from LeClair et al. [33] reports that the BLEU scores are much higher for randomly split datasets than for datasets split by project. We investigate the influence in Section 7.2. Although the BLEU scores have dropped, our approach still outperforms the baselines when the dataset is split by project.

Our dataset is constructed from Java repositories in Github. Java is a popular programming language and is used in a large amount of projects. We believe that our results will generalize to other programming languages but have not yet tested this. In the future, we plan to collect even more commits from more repositories and to try to extend our approach to other programming languages.

The second key threat to external validity is that we only use the first sentences in the commit messages as the target, though this is the same as used in other approaches [25][39][37]. However, many developers usually summarize the entire commit message in the first sentence. Similarly, Gu et al. [20] use the first sentence of JavaDoc as the summary of the API comments.

The third key threat to external validity relates to the filtering of our dataset. When we construct the dataset, our processing process is similar to that of previous works [25][39][37]; we all filter out a large amount of commits to ensure the quality of the generated messages. Since the existing messages have different writing styles and some of the messages are poorly written, we use the V-DO pattern to obtain a set of relatively good-quality commit messages. We only filter according to the quality of commit messages submitted by developers, and there is no restriction on the type of code changes. The noisy commits are removed because they convey little valuable information and are easy to generate. The influence of the filtering on our approach has been reported in Section 7.1.

7.4.3 Construct Validity. The construct validity concerns the relation between theory and observation. In this study, one threat to validity relates to our user study in Section 5. Although we have minimized the number of commits each participant needs to evaluate to prevent participants from impatiently scoring, we cannot guarantee that the scores of all participants were actually carefully considered. In addition, each participant's evaluation criteria is also a variable factor. Ideally we need further evaluators and with greater experience in future studies.

Another threat relates to suitability of our evaluation metrics. To investigate the effectiveness of our approach and baselines, we use BLEU, METEOR and ROUGE-L in our experiments. These metrics are classical evaluation measures in Neural Machine Translation. Except for some previous works in commit message generation [25, 37, 39, 68], some similar tasks in software engineering also use these evaluation metrics to measure the performance, e.g. code summarization [22, 23, 32, 70] and pull request generation [40].

8 RELATED WORK

We describe key related work on commit message generation, neural machine translation, and exposure bias.

Commit Message Generation. A number of techniques have been proposed for automatic commit message generation. These can be divided into three categories:

Rule-based: DeltaDoc [10] and ChangeScribe [36][13] filled a pre-defined template to generate the commit messages with extracted information. Shen et al. [55] extracted code changes based on pre-defined types of changed methods and constrained the length of the generated message by removing repeated information in the changes. However, these approaches can only deal with pre-defined types of code changes. Limitations are that they can not cover some cases and explain the reason or purpose behind code changes.

Retrieval-based: The commit messages generated by retrieval-based methods [39][24] are relatively more readable and sometimes contain low-frequency words like method names. Liu et al. [39] and Huang et al. [24] utilized Information Retrieval (IR) to reuse the commit messages of similar changes. Specifically, Huang et al. [24] used syntactic similarity and semantic similarity between their changed code fragments to measure the similarity between two commits. Liu et al. [39] generated concise commit messages using the nearest neighbor algorithm. Although Liu et al.'s method has achieved very good results in terms of effectiveness and efficiency, retrieval based approaches cannot output accurate messages when there are no similar *diffs* in the dataset. Thus retrieval-based methods heavily rely on whether the dataset is broad enough.

Learning-based: In recent years, deep learning models have been used to translate the *git diffs* into commit messages [42][25][41][37][38]. For example, Jiang et al. [25] and Loyola et al. [42][41] used attentional encoder-decoder model to generate messages. Liu et al. [37] proposed a pointer-generator network PtrGNCMsg to address out-of-vocabulary (OOV) issue. The above approaches all directly use *git diffs* which include both code changes and non-code changes. Xu et al. [68] proposed jointly modeling code structure and semantics for better performance. The proposed approach CODISUM keeps only the source code in the *diff* files as input.

Although the results of these approaches are promising, they still suffer from two restrictions: low-frequency vocabulary problem and *exposure bias* issue. Our method take advantages of both retrieval-based methods and NMT-based methods to address the low-frequency vocabulary problem. We further propose use of a decay sampling mechanism to mitigate *exposure bias*. These innovations make our proposed method CoRec perform better than the state-of-the-art work.

Retrieval-guided Neural Machine Translation. In the Natural Language Processing (NLP) community, retrieval-guided neural machine translation have been proposed in [16][34][19][69][70].

Amin et al. [16] used retrieved pairs to fine-tune the model and introduced a dynamic method to learn more efficiently from the retrieved set. Gu et al. [19] proposed search engine guided neural machine translation (SEG-NMT). SEG-NMT needs additional encoders for the retrieved sentences and will bring high overhead for calculating in the training phase. Different from their work, our proposed approach does not need additional encoders but reuses the trained model to retrieve similar *diffs*.

Zhang et al. [69] extracted translation pieces. These refer to n-grams occurring in the retrieved target sentences that also match words that overlap between the input and retrieved source sentences. However, their method is only suitable for datasets where the source sentence and target sentence have the property of word alignment. In the commit message generation task, there are few direct correspondences between the words of *git diff* and commit messages. Similar to our approach, Zhang et al. [70] take advantages of both neural and retrieval-based techniques to automatically perform source code summarization. However, their approach only trains a traditional attentional encoder-decoder model without considering the gap (*exposure bias*) between training phase and testing phase. We employ a decay sampling mechanism [71][9] when training the model to mitigate this issue and the experimental results (as shown in Section 4.5.2) have shown its effectiveness.

Exposure Bias. Ranzato et al. [52] first pointed out the *exposure bias* problem i.e. that using teacher forcing means the model has never been trained on its own errors. In order to address this issue, Bengio et al. [9] proposed a scheduled sampling for sequence-to-sequence RNN models that randomly pick the gold target or generated word as the input of decoder embedding at training. Goyal et al. [18] proposed a novel approach based on scheduled sampling that uses a differentiable approximation of previous predictions inside the training objective by incorporating a continuous relaxation of argmax. Zhang et al. [71] selected *oracle words* from predicted words. This selected not only with a word-by-word greedy search but also with a sentence-level search, and then sampled

as context from the *oracle words* and ground truth words. By contrast, our approach focuses on the previous alignment choices rather than the predicted words and our sampling mechanism can adapt to different sizes of datasets.

Other studies have suggested that sentence-level metrics, e.g., BLEU, can bring some flexibility to the generation [52][56][65]. Ranzato et al. [52] borrowed ideas from REINFORCE algorithm [63] and scheduled sampling [9]. This approach avoids *exposure bias* by using model predictions at training time and utilizes a sequence level loss to optimize results. Similarly, beam search optimization [65] and Minimum Risk Training (MRT) [56] also directly optimize the evaluation metric.

9 CONCLUSION

Automatic commit message generation is helpful to aid developers in writing well-commented changes. We proposed a novel approach CoRec (Context-Aware Retrieval-based Deep Commit Message Generation). We build upon the machine translation model, and use the most similar commit retrieved to enhance the performance of commit message generation. Furthermore, we introduce a decay sampling mechanism to make the model fully exposed to the circumstance at inference, which mitigates the *exposure bias* problem. The experimental results demonstrate that CoRec significantly outperforms state-of-the-art approaches in commit message generation. To the issue of no large-scale dataset for commit message generation analysis, we built a well-cleaned dataset which includes commits extracted from 10,000 repositories in Github. We have made this dataset publicly available for community research.

In the future, we plan to incorporate additional information like bug reports and code structure analysis to further improve automated commit message generation.

ACKNOWLEDGEMENTS

This research was partially supported by the Australian Research Council's Discovery Early Career Researcher Award (DECRA) funding scheme (DE200100021), ARC Laureate Fellowship funding scheme (FL190100035), ARC Discovery grant (DP200100020), and ...

REFERENCES

[1] 2019. Git. https://git-scm.com/.
[2] 2019. NLTK. http://www.nltk.org/.
[3] 2020. College English Test. https://en.wikipedia.org/wiki/College_English_Test.
[4] Hervé Abdi. 2007. Bonferroni and Šidák corrections for multiple comparisons. *Encyclopedia of measurement and statistics* 3 (2007), 103–107.
[5] John Aldrich et al. 1997. RA Fisher and the making of maximum likelihood 1912-1922. *Statistical science* 12, 3 (1997), 162–176.
[6] Philip Arthur, Graham Neubig, and Satoshi Nakamura. 2016. Incorporating discrete translation lexicons into neural machine translation. *arXiv preprint arXiv:1606.02006* (2016).
[7] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. 2014. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473* (2014).
[8] Satanjeev Banerjee and Alon Lavie. 2005. METEOR: An automatic metric for MT evaluation with improved correlation with human judgments. In *Proceedings of the acl workshop on intrinsic and extrinsic evaluation measures for machine translation and/or summarization*. 65–72.
[9] Samy Bengio, Oriol Vinyals, Navdeep Jaitly, and Noam Shazeer. 2015. Scheduled sampling for sequence prediction with recurrent neural networks. In *Advances in Neural Information Processing Systems*. 1171–1179.
[10] Raymond PL Buse and Westley R Weimer. 2010. Automatically documenting program changes. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*. 33–42.
[11] Jiachi Chen, Xin Xia, David Lo, John Grundy, Xiapu Luo, and Ting Chen. 2020. Defining Smart Contract Defects on Ethereum. *IEEE Transactions on Software Engineering* (2020).
[12] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. 2014. Learning phrase representations using RNN encoder-decoder for statistical machine translation.

arXiv preprint arXiv:1406.1078 (2014).

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

[14] Brian De Alwis and Jonathan Sillito. 2009. Why are software projects moving from centralized to decentralized version control systems?. In *2009 ICSE Workshop on Cooperative and Human Aspects on Software Engineering*. IEEE, 36–39.

[15] Robert Dyer, Hoan Anh Nguyen, Hridesh Rajan, and Tien N Nguyen. 2013. Boa: A language and infrastructure for analyzing ultra-large-scale software repositories. In *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 422–431.

[16] M Amin Farajian, Marco Turchi, Matteo Negri, and Marcello Federico. 2017. Multi-domain neural machine translation through unsupervised adaptation. In *Proceedings of the Second Conference on Machine Translation*. 127–137.

[17] Joseph L Fleiss. 1971. Measuring nominal scale agreement among many raters. *Psychological bulletin* 76, 5 (1971), 378.

[18] Kartik Goyal, Chris Dyer, and Taylor Berg-Kirkpatrick. 2017. Differentiable scheduled sampling for credit assignment. *arXiv preprint arXiv:1704.06970* (2017).

[19] Jiatao Gu, Yong Wang, Kyunghyun Cho, and Victor OK Li. 2018. Search engine guided neural machine translation. In *Thirty-Second AAAI Conference on Artificial Intelligence*.

[20] Xiaodong Gu, Hongyu Zhang, Dongmei Zhang, and Sunghun Kim. 2016. Deep API learning. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 631–642.

[21] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation* 9, 8 (1997), 1735–1780.

[22] Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. 2018. Deep code comment generation. In *2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC)*. IEEE, 200–20010.

[23] Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. 2020. Deep code comment generation with hybrid lexical and syntactical information. *Empirical Software Engineering* 25, 3 (2020), 2179–2217.

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

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

[26] Siyuan Jiang and Collin McMillan. 2017. Towards automatic generation of short summaries of commits. In *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*. IEEE, 320–323.

[27] Łukasz Kaiser and Ilya Sutskever. 2015. Neural gpus learn algorithms. *arXiv preprint arXiv:1511.08228* (2015).

[28] Mira Kajko-Mattsson. 2005. A survey of documentation practice within corrective maintenance. *Empirical Software Engineering* 10, 1 (2005), 31–55.

[29] Yoon Kim. 2014. Convolutional neural networks for sentence classification. *arXiv preprint arXiv:1408.5882* (2014).

[30] Diederik P Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* (2014).

[31] Guillaume Klein, Yoon Kim, Yuntian Deng, Jean Senellart, and Alexander M Rush. 2017. Opennmt: Open-source toolkit for neural machine translation. *arXiv preprint arXiv:1701.02810* (2017).

[32] Alexander LeClair, Siyuan Jiang, and Collin McMillan. 2019. A neural model for generating natural language summaries of program subroutines. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 795–806.

[33] Alexander LeClair and Collin McMillan. 2019. Recommendations for Datasets for Source Code Summarization. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. 3931–3937.

[34] Xiaoqing Li, Jiajun Zhang, and Chengqing Zong. 2016. One sentence one model for neural machine translation. *arXiv preprint arXiv:1609.06490* (2016).

[35] Chin-Yew Lin. 2004. ROUGE: A Packagefor Automatic Evaluation of Summaries. In *ProceedingsofWorkshop on Text Summarization Branches Out, Post2Conference Workshop of ACL*.

[36] Mario Linares-Vásquez, Luis Fernando Cortés-Coy, Jairo Aponte, and Denys Poshyvanyk. 2015. Changescribe: A tool for automatically generating commit messages. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 2. IEEE, 709–712.

[37] Qin Liu, Zihe Liu, Hongming Zhu, Hongfei Fan, Bowen Du, and Yu Qian. 2019. Generating commit messages from diffs using pointer-generator network. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE, 299–309.

[38] Shangqing Liu, Cuiyun Gao, Sen Chen, Lun Yiu Nie, and Yang Liu. 2019. ATOM: Commit Message Generation Based on Abstract Syntax Tree and Hybrid Ranking. *arXiv preprint arXiv:1912.02972* (2019).

[39] Zhongxin Liu, Xin Xia, Ahmed E Hassan, David Lo, Zhenchang Xing, and Xinyu Wang. 2018. Neural-machine-translation-based commit message generation: how far are we?. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. 373–384.

[40] Zhongxin Liu, Xin Xia, Christoph Treude, David Lo, and Shanping Li. 2019. Automatic generation of pull request descriptions. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 176–188.

[41] Pablo Loyola, Edison Marrese-Taylor, Jorge Balazs, Yutaka Matsuo, and Fumiko Satoh. 2018. Content aware source code change description generation. In *Proceedings of the 11th International Conference on Natural Language Generation*. 119–128.

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

[43] Minh-Thang Luong, Hieu Pham, and Christopher D Manning. 2015. Effective approaches to attention-based neural machine translation. *arXiv preprint arXiv:1508.04025* (2015).

[44] Walid Maalej and Hans-Jörg Happel. 2010. Can development work describe itself?. In *2010 7th IEEE working conference on mining software repositories (MSR 2010)*. IEEE, 191–200.

[45] Christopher D Manning, Mihai Surdeanu, John Bauer, Jenny Rose Finkel, Steven Bethard, and David McClosky. 2014. The Stanford CoreNLP natural language processing toolkit. In *Proceedings of 52nd annual meeting of the association for computational linguistics: system demonstrations*. 55–60.

[46] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781* (2013).

[47] Audris Mockus and Lawrence G Votta. 2000. Identifying Reasons for Software Changes using Historic Databases.. In *icsm*. 120–130.

[48] Graham Neubig, Zi-Yi Dou, Junjie Hu, Paul Michel, Danish Pruthi, Xinyi Wang, and John Wieting. 2019. compare-mt: A tool for holistic comparison of language generation systems. *arXiv preprint arXiv:1903.07926* (2019).

[49] Yusuf Sulistyo Nugroho, Hideaki Hata, and Kenichi Matsumoto. 2020. How different are different diff algorithms in Git? *Empirical Software Engineering* 25, 1 (2020), 790–823.

[50] Yusuke Oda, Hiroyuki Fudaba, Graham Neubig, Hideaki Hata, Sakriani Sakti, Tomoki Toda, and Satoshi Nakamura. 2015. Learning to generate pseudo-code from source code using statistical machine translation (t). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 574–584.

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

[52] Marc'Aurelio Ranzato, Sumit Chopra, Michael Auli, and Wojciech Zaremba. 2015. Sequence level training with recurrent neural networks. *arXiv preprint arXiv:1511.06732* (2015).

[53] Mike Schuster and Kuldip K Paliwal. 1997. Bidirectional recurrent neural networks. *IEEE transactions on Signal Processing* 45, 11 (1997), 2673–2681.

[54] Shikhar Sharma, Layla El Asri, Hannes Schulz, and Jeremie Zumer. 2017. Relevance of Unsupervised Metrics in Task-Oriented Dialogue for Evaluating Natural Language Generation. *CoRR* abs/1706.09799 (2017). http://arxiv.org/abs/1706.09799

[55] Jinfeng Shen, Xiaobing Sun, Bin Li, Hui Yang, and Jiajun Hu. 2016. On automatic summarization of what and why information in source code changes. In *2016 IEEE 40th Annual Computer Software and Applications Conference (COMPSAC)*, Vol. 1. IEEE, 103–112.

[56] Shiqi Shen, Yong Cheng, Zhongjun He, Wei He, Hua Wu, Maosong Sun, and Yang Liu. 2015. Minimum risk training for neural machine translation. *arXiv preprint arXiv:1512.02433* (2015).

[57] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. 2014. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*. 3104–3112.

[58] Christoph Tillmann and Hermann Ney. 2003. Word reordering and a dynamic programming beam search algorithm for statistical machine translation. *Computational linguistics* 29, 1 (2003), 97–133.

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

[60] Zhiyuan Wan, Xin Xia, Ahmed E Hassan, David Lo, Jianwei Yin, and Xiaohu Yang. 2018. Perceptions, expectations, and challenges in defect prediction. *IEEE Transactions on Software Engineering* (2018).

[61] Zhiyuan Wan, Xin Xia, David Lo, and Gail C Murphy. 2019. How does machine learning change software development practices? *IEEE Transactions on Software Engineering* (2019).

[62] Frank Wilcoxon. 1992. Individual comparisons by ranking methods. In *Breakthroughs in statistics*. Springer, 196–202.

[63] Ronald J Williams. 1992. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning* 8, 3-4 (1992), 229–256.

[64] Ronald J Williams and David Zipser. 1989. A learning algorithm for continually running fully recurrent neural networks. *Neural computation* 1, 2 (1989), 270–280.

[65] Sam Wiseman and Alexander M Rush. 2016. Sequence-to-sequence learning as beam-search optimization. *arXiv preprint arXiv:1606.02960* (2016).

[66] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, et al. 2016. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144* (2016).

[67] Bowen Xu, Zhenchang Xing, Xin Xia, and David Lo. 2017. AnswerBot: Automated generation of answer summary to developers' technical questions. 706–716. https://doi.org/10.1109/ASE.2017.8115681

[68] Shengbin Xu, Yuan Yao, Feng Xu, Tianxiao Gu, Hanghang Tong, and Jian Lu. 2019. Commit message generation for source code changes. In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI-19. International Joint Conferences on Artificial Intelligence Organization*, Vol. 7. 3975–3981.

[69] Jingyi Zhang, Masao Utiyama, Eiichro Sumita, Graham Neubig, and Satoshi Nakamura. 2018. Guiding neural machine translation with retrieved translation pieces. *arXiv preprint arXiv:1804.02559* (2018).

[70] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, and Xudong Liu. 2020. Retrieval-based Neural Source Code Summarization. In *Proceedings of the 42nd International Conference on Software Engineering*.

[71] Wen Zhang, Yang Feng, Fandong Meng, Di You, and Qun Liu. 2019. Bridging the gap between training and inference for neural machine translation. *arXiv preprint arXiv:1906.02448* (2019).