Automatically Generating Commit Messages from Diffs using Neural Machine Translation

Siyuan Jiang, Ameer Armaly, and Collin McMillan
Department of Computer Science and Engineering
University of Notre Dame, Notre Dame, IN, USA
{sjiang1, aarmaly, cmc}@nd.edu

***Abstract*—Commit messages are a valuable resource in comprehension of software evolution, since they provide a record of changes such as feature additions and bug repairs. Unfortunately, programmers often neglect to write good commit messages. Different techniques have been proposed to help programmers by automatically writing these messages. These techniques are effective at describing what changed, but are often verbose and lack context for understanding the rationale behind a change. In contrast, humans write messages that are short and summarize the high level rationale. In this paper, we adapt Neural Machine Translation (NMT) to automatically "translate" diffs into commit messages. We trained an NMT algorithm using a corpus of diffs and human-written commit messages from the top 1k Github projects. We designed a filter to help ensure that we only trained the algorithm on higher-quality commit messages. Our evaluation uncovered a pattern in which the messages we generate tend to be either very high or very low quality. Therefore, we created a quality-assurance filter to detect cases in which we are unable to produce good messages, and return a warning instead.**

I. Introduction

Commit messages are natural language descriptions of changes in source code. When a programmer updates code, a typical procedure is to upload the change to a version control system with a short commit message to describe the purpose of the change, e.g., "adds support for 9 inch tablet screen size." The repository stores the message alongside a `diff` that represents the difference between the current and previous version of the affected files. The practice is extremely common: for this paper alone, we obtained over 2M diffs and messages from just 1k projects on GitHub.

Commit messages are useful because they help programmers to understand the high level rationale for a change without reading the low level implementation details. They serve a valuable purpose in comprehension of software evolution, and act as a record of feature additions and bug repairs [7]. Unfortunately, programmers sometimes neglect commit messages [12], [38], likely due to the same time and market pressures that have been reported to affect many types of documentation [48], [13], [26]. In short, programmers use commit messages but often avoid writing them themselves.

Automated generation of commit messages has been proposed as an alternative to manual efforts by programmers. For example, Buse *et al.* [7] describe DeltaDoc, a tool that summarizes what changed in the control flow of a program between code versions. Likewise, Cortes-Coy *et al.* [34] built ChangeScribe, which summarizes changes such as method additions. These and other existing techniques (see Section II) have been shown to be effective in answering questions about what changed and where from one code version to another.

What is missing from existing approaches is a short, high level description of the purpose behind commits. Current approaches are effective at summarizing what changed and where, but do not answer the question *why* [7]. Questions of *why* traditionally require human insight since they involve synthesis of different, complex data sources and context. However, as Mockus *et al.* [40] observed, many commit messages are similar and can be broadly categorized as related to bug repair, feature additions, etc. Plus, they follow similar grammatical patterns such as verb-direct object structure (e.g. "adds support for...") [24]. This observation leads us to believe that the text of commit messages can be learned and predicted if there is sufficient data. Our view is in line with the hypothesis of "naturalness" of software [21], that software artifacts follow patterns that can be learned from sufficiently large datasets.

In this paper, we adapt a neural machine translation (NMT) algorithm to the problem of commit message generation. Several NMT algorithms have been designed to translate between natural languages by training a neural network on pairs of sentences that humans have already translated. The datasets required are enormous by typical software engineering research standards, involving up to tens of millions of pairs of sentences [53], [36]. We trained an NMT algorithm using pairs of `diffs` and commit messages from 1k popular projects on GitHub. While we were able to obtain quite large datasets (over 2M commits), we encountered many commit messages that were gibberish or very low quality (a problem others have observed [12], [38]), which if left in the training data could be reflected in the NMT's output. Therefore, we designed a filter to ensure that we only trained the algorithm using messages with a verb-direct object pattern.

We investigate and report the effectiveness of the predictions from the process. We found promising results as well as key constraints on the accuracy of the predictions. In short, the NMT process performed quite well under select conditions, but poorly in others. We report these results and promising and poor conditions as a guide to other researchers and platform for advancement in this research area. To further promote advancement of the area, we make our implementation and data freely available in an online replication package.

Our approach has two key advantages that make it a supplement to, rather than a competitor of, existing automatic commit message generation techniques. First, we produce short summary messages rather than exhaustive descriptions of code changes. And second, our approach produces messages for changes to many types of software artifact in the repository, not solely source code.

A. The Problem

In this paper we target the problem of automatically generating commit messages. Commit messages are useful in the long term for program comprehension and maintainability, but cost significant time and effort in the short term. These short term pressures lead programmers to neglect writing commit messages, like other types of documentation [12], [38], [48], [13], [26]. Buse *et al.* [7] point out that programmers use commit messages for two reasons: 1) to summarize *what* changed, and 2) to briefly explain *why* the change was necessary. To date, research into commit message generation has exclusively focused on the question *what*. In this paper, we seek to begin answering *why*.

Existing commit message generation techniques produce relatively long messages that include details such as the methods that were added or the number of files changes (*what* information). While useful, these techniques are a complement to, rather than a replacement for, high level *why* information that humans write such as "adds support for 9 inch tablet screens." Normally, this high level information requires human judgment. But we hypothesize that there are patterns of commits, and that these patterns can be detected and used to generate messages for similar commits later. Given a large number of pairs of `diffs` and messages, we believe we can train an algorithm to write new messages for new commits, based on the new commits' similarity to older ones.

Please note that we do not claim to generate new insights for completely new types of commits – that task is likely to remain in the hands of human experts. However, we do aim to write messages that reflect knowledge that can be learned from records of previous commits. In the long run, we hope that this technology will help reduce manual effort by programmers in reading and understanding code changes in repositories.

B. Paper Overview

Figure 1 depicts an overview of our paper. We have divided the work into three segments: In Part A (Section IV), we present our approach to filtering for verb/direct-object (V-DO) commit message patterns and training an NMT algorithm to produce messages with this pattern. The V-DO filter was introduced because a large percentage of the messages in the repositories we downloaded were very low quality, and we needed to ensure that we trained the NMT algorithm only with examples matching an acceptable pattern. We then trained an NMT algorithm on the pairs of `diffs` and commit messages where the messages followed the V-DO pattern.

In Part B (Sections V and VI), we evaluate the quality of the commit messages produced by the algorithm with an automated method and a human study with 2 Ph.D. students and 18 professional programmers. We observe that while there are a significant number of positive results, there are also a significant number of negative results. Therefore, in Part C (Sections VII), we design a quality assurance (QA) filter to detect cases in which the NMT algorithm is likely to produce a negative result. We then modify our approach to produce a warning message instead of a commit message in those cases, and update our evaluation to show the effects of our modification. In short, we reduce the number of poor predicted messages by 44% at a cost of also mistakenly reducing high quality predictions by 11%.

II. Related Work

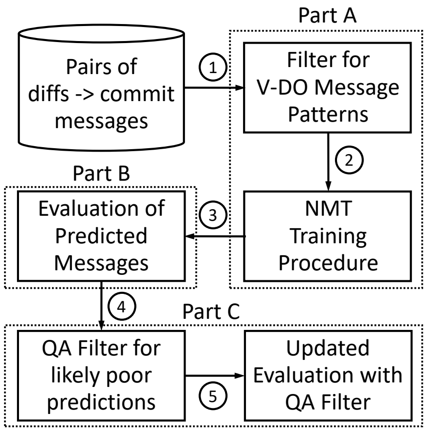
We split the related work into three categories: 1) the work that generates commit messages; 2) the work that summarizes source code; and 3) the work that applies deep learning algorithms in software engineering.

A. Commit Message Generation Techniques

We categorize the commit message generation techniques into three groups based on the inputs of the techniques. **The first group** uses code changes of a commit as an input, and summarizes the changes to generate the commit message. For example, Buse *et al.* have built DeltaDoc, which extracts path predicates of changed statements, and follows a set of predefined rules to generate a summary [7]. Similarly, Linares-Vásquez *et al.* have built ChangeScribe, which extracts changes between two Abstract Syntax Trees and summarizes the changes based on predefined rules [34].

Supplementing the first group, **the second group** is based on related software documents. For example, Le *et al.* have built RCLinker, which links a bug report to the corresponding commit message [33]. Rastkar and Murphy have proposed to summarize multiple related documents for commits [47]. Integrating the ideas of the first and the second groups, Moreno *et al.* have built ARENA, which summarizes changes and finds related issues to generate release notes [41].

The third group is our technique using `diffs` (generated by "`git diff`") as inputs. Our technique is to translate a `diff`


Part A
Pairs of diffs -> commit messages
1
Filter for V-DO Message Patterns
2
Part B
Evaluation of Predicted Messages
3
NMT Training Procedure
4
Part C
QA Filter for likely poor predictions
5
Updated Evaluation with QA Filter
Fig. 1: The overview of our paper.

to a natural language sentence. Our technique supplements the first group in two ways. First, the techniques in the first group often generate multi-line summaries that contain pseudocode and template text. In contrast, our technique generates one-sentence descriptions, which can be used as a headline of the multi-line summaries. Second, our technique summarizes both code and non-code changes in `diff`s.

B. Source Code Summarization

Source code summarization techniques generate descriptions of source code pieces [42]. The algorithms of the techniques can be adapted to generate summaries for changes in commits. Code summarization can be categorized into two groups: extractive and abstractive. Extractive summarization extracts relevant parts of source code and uses the relevant parts as a summary [17]. Abstractive summarization includes information that is not explicitly in the source code. For example, Sridhara *et al.* has designed a Natural Language Generation (NLG) system to create summaries of Java methods [54]. First, the NLG system finds important statements of a Java method. Second, the system uses a text generation algorithm to transform a statement to a natural language description. This algorithm has predefined text templates for different statement types, such as `return` statements and `assignment` statements. Both DeltaDoc and ChangeScribe (discussed in Section II-A) follow the similar NLG design.

Besides the NLG approach to generate abstractive summaries, Iyer *et al.* have built Code-NN, which uses an Neural Machine Translation (NMT) algorithm to summarize code snippets [23]. This work is similar to our technique because our technique also uses an NMT algorithm. There are two key differences between our technique and Code-NN. First, the goal of Code-NN is summarizing code snippets and the goal of our technique is summarizing changes. Second, Code-NN parses code snippets and removed all the comments. In contrast, our technique's input is an entire `diff` with code, comments, and `diff` marks (e.g., + denoting insertion).

C. Deep Learning in Software Engineering

Deep learning algorithms are becoming more prevalent in Software Engineering research. Deep learning algorithms, as applied to software, automatically learn representations of software artifacts. For example, to detect code clones, traditional approaches predefine the representations of code fragments (some techniques use token sequences to represent code [27]; others use graphs [31], [9]). In contrast, the deep learning approach introduced by White *et al.* [57] learns the representations of code automatically. Similarly, deep learning algorithms are introduced in bug localization [32], software traceability [16], and code suggestions [58].

Our technique is similar to the work done by Gu *et al.* [15], because both our and their techniques use Neural Machine Translation (NMT). Gu *et al.* use NMT to translate natural language queries to API method sequences [15]. Similarly, several code generation techniques use NMT to translate natural language to programming language [35], [46]. In contrast, our technique translates `diff`s to natural language.

Our technique is also similar to the work by Alexandru *et al.* [5], which investigates the suitability of NMT for program comprehension. Alexandru *et al.* use NMT for source code tokenization and token annotation. While Alexandru *et al.* target on lower-level source code understanding (token-level), we target on understanding higher-level of mixtures of code and text (`diff`-level).

III. Background

We split the background section into three subsections. The first subsection is about the empirical studies on commit messages, which motivate us to generate short descriptions of commits. The second subsection describes *RNN Encoder-Decoder*, a popular Neural Network Translation model, which is an important background for the third subsection. The third subsection describes *attentional RNN Encoder-Decoder*, which is used in our work.

A. Commit Messages

Our work is motivated by the findings of the studies by Buse *et al.* [7] and by Jiang and McMillan [24]. The results of the two studies indicate three things. First, commit messages are pervasive and desired. Buse *et al.* examined 1k commits from five mature software projects and found that 99.1% of the commits have non-empty messages. Jiang and McMillan collected over 2M commit messages from 1k projects.

Second, human-written commit messages are short. In Buse *et al.*'s study, the average size of the 991 non-empty commit messages is 1.1 lines. Similarly, the study of Jiang and McMillan shows that 82% of the commit messages have only one sentence.

Third, commit messages contain various types of information not solely summaries of code changes. Buse *et al.* manually analyzed 375 commit messages and found that the messages are not only about what the changes are but also about why the changes are made. Supported by the three findings, our technique aims to generate one-sentence commit messages which mimic the human-written commit messages.

B. RNN Encoder-Decoder Model

Neural Machine Translation (NMT) is neural networks that model the translation process from a source language sequence $x = (x_1, ..., x_n)$ to a target language sequence $y = (y_1, ..., y_n)$ with the conditional probability $p(y|x)$ [5], [37]. Cho *et al.* introduced *RNN Encoder-Decoder* as an NMT model [10], which is commonly used and can produce state of the art translation performance [53], [36]. As a promising deep learning model, *RNN Encoder-Decoder* has been used in addressing other software engineering tasks [15], [5].

RNN Encoder-Decoder has two recurrent neural networks (RNNs). One RNN is used to transform source language sequences into vector representations. This RNN is called the encoder. The other RNN is used to transform the vector representations to the target language sequences, which is called the decoder.

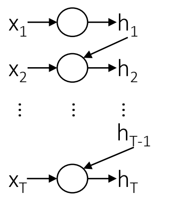


Fig. 2: The architecture of the encoder in *RNN Encoder-Decoder*

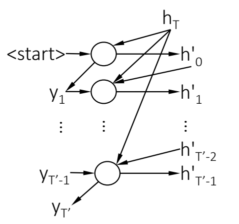


Fig. 3: The architecture of the decoder in *RNN Encoder-Decoder*

1) Encoder: The input of the encoder is a variable-length sequence $x = (x_1, ..., x_T)$. The encoder takes one symbol at a time as shown in Figure 2. As an RNN, the encoder has a hidden state h, which is a fixed-length vector. At a time step t, the encoder computes the hidden state h_t by:

$$h_t = f(h_{t-1}, x_t) \tag{1}$$

where f is a non-linear function. Two common options for f are long short-term memory (LSTM) [22] and the gated recurrent unit (GRU) [10] (due to space limit, we do not describe these two unit types in detail here). For example, Bahdanau *et al.* use GRU [6] and Sutskever *et al.* use LSTM [55]. The last symbol of x should be an end-of-sequence (`<eos>`) symbol which notifies the encoder to stop and output the final hidden state h_T, which is used as a vector representation of x.

2) Decoder: Figure 3 shows the RNN of the decoder. The output of the decoder is the target sequence $y = (y_1, ..., y_{T'})$. One input of the decoder is a `<start>` symbol denoting the beginning of the target sequence. At a time step t, the decoder computes the hidden state h'_t and the conditional distribution of the next symbol y_t by:

$$h'_t = f(h'_{t-1}, y_{t-1}, h_T) \tag{2}$$

$$p(y_t|y_{t-1}, ..., y_1, h_T) = g(h'_t, y_{t-1}, h_T) \tag{3}$$

where h_T (generated by the encoder) is called the *context vector*; f and g are non-linear functions. Function f here and f in Equation 1 are often the same. Function g must produce valid probabilities. For example, softmax can be used as g. The decoder finishes when it predicts an `<eos>` symbol.

3) Training Goal: The encoder and the decoder are jointly trained to maximize the conditional log-likelihood:

$$\max_\theta \frac{1}{N} \sum_{i=1}^{N} \log p(y_i|x_i; \theta) \tag{4}$$

where θ is the set of the model parameters; N is the size of the training set; and each (x_i, y_i) is a pair of a source sequence and a target sequence in the training set.

C. Attentional RNN Encoder-Decoder and Nematus

Bahdanau *et al.* introduced the *attentional RNN Encoder-Decoder*, in which attention mechanism is introduced to deal with long source sequences [6]. We use this mechanism in our work because our source sequences, `diffs`, are much longer than natural language sentences. The attention mechanism includes several modifications in both the encoder and the decoder, which we describe in the following subsections.

1) Encoder: The encoder in the attentional model is a bidirectional RNN, which has two RNNs: forward and backward. The two RNNs have the same architecture. The forward RNN is the same as the RNN in the original *RNN Encoder-Decoder* model (Figure 2), which reads the source sequence x as it is ordered, from x_1 to x_T. The forward RNN generates a sequence of the hidden states $(\overrightarrow{h}_1, ... \overrightarrow{h}_T)$. In contrast, the backward RNN reads x in the reversed order, and generates a sequence of the hidden states $(\overleftarrow{h}_T, ... \overleftarrow{h}_1)$.

In the end, for each symbol x_i in x, the encoder outputs $h_i = [\overrightarrow{h_i}; \overleftarrow{h_i}]$, which is a concatenation of $\overrightarrow{h_i}$ and $\overleftarrow{h_i}$.

2) Decoder: The decoder computes the hidden state h'_t and the conditional distribution of the next symbol y_t by:

$$h'_t = f(h'_{t-1}, y_{t-1}, c_t) \tag{5}$$

$$p(y_t|y_{t-1}, ..., y_1, c_t) = g(h'_t, y_{t-1}, c_t) \tag{6}$$

where f and g are non-linear functions like f and g in Equations 2 and 3. c_t is the distinct context vector for y_t, and can be computed by

$$c_t = \sum_{i=1}^{T} \alpha_{ti} h_i \tag{7}$$

where T is the length of the input sequence; the weight α_{ti} can be trained jointly with the other components in the model, and h_i is generated by the encoder. Since c_t is designed to introduce the context's impact to y_t, *attentional RNN Encoder-Decoder* works better on long source sequences. Therefore, we use this NMT model in this paper rather than the original one.

IV. Approach

This section describes our approach, including the data set preparation and the NMT training procedure. This section corresponds to Part A in the paper overview Figure 1, and is detailed in Figure 4.

A. Preparing a Data Set for NMT

We used the commit data set provided by Jiang and McMillan [24], which contains 2M commits. The data set includes commits from top 1k Java projects (ordered by the number of stars) in Github. We describe how we prepared the data set for NMT algorithms as follows.

1) Preprocessing the Data Set: First, we extracted the first sentences from the commit messages. We used the first sentences as the target sequences because the first sentences often are the summaries of the entire commit messages. Similarly, Gu *et al.* used the first sentences of the API comments as their target sequences [15]. Second, we removed issue ids from the extracted sentences and removed commit ids from the `diffs`, because issue ids and commit ids are unique ids and increase the vocabularies of the source and the target languages dramatically, which in turn cause large memory use of NMT.

Third, we removed merge and rollback commits (the same practice done by Jiang and McMillan [24]). Merges and rollbacks are removed because the `diffs` of merges and rollbacks are often more than thousands of lines, which NMT is not suitable to translate. For the same reason, we also removed any `diff` that is larger than 1MB.

After the above steps, we have 1.8M commits remaining. Finally, we tokenized the extracted sentences and the `diffs` by white spaces and punctuations. We did not split CamelCase so that identifiers (e.g., class names or method names) are treated as individual words in this study.

2) Setting Maximum Sequence Lengths for NMT Training: A maximum sequence length for both source and target sequences need to be set for an *RNN Encoder-Decoder* [6], [50]. Since NMT is for translating natural language sentences, maximum sequence lengths for both source and target sequences are often set between 50 to 100 [6], [50]. Because the lengths of our source and target sequences are very different, we set the maximum sequence lengths separately.

For our target sequences, we set the maximum length at 30 tokens (including words and punctuations), because the first sentences from the commit messages tend to be short. In our data set, 98% of the first sentences have less than 30 tokens.

For our source sequences, we set the maximum length at 100 tokens because 100 is the largest maximum length used by NMT in natural language translation. Many configurations are possible, and optimizing the maximum `diff` length for generating commit messages is an area of future work. In pilot studies, a maximum length of 100 outperformed lengths of 50 and 200.

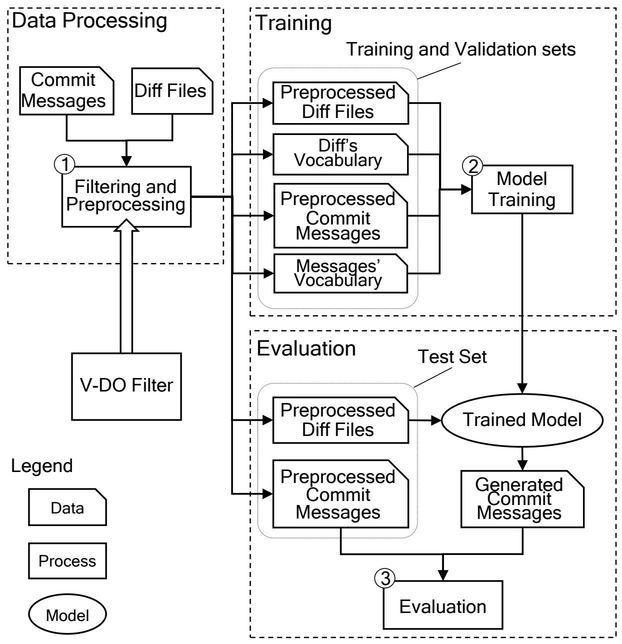


Fig. 4: The detailed process in Part A, Figure 1. There are three main steps: 1) filtering and preprocessing the data; 2) training a Neural Machine Translation model; 3) evaluating the model, which is Part B, Figure 1.

After applying the maximum lengths for source and target sequences (30 and 100), we have 75k commits remaining.

3) V-DO Filter: We introduced *Verb-Direct Object* (V-DO) filter because we found that the existing messages have different writing styles and some of the messages are poorly written, which may affect the performance of NMT.

To obtain a set of commit messages that are in a similar format, we filtered the messages for verb/direct-object pattern. We chose this pattern because a previous study shows that 47% of commit messages follow this pattern [24]. To find the pattern, we used a Natural Language Processing (NLP) tool, Stanford CoreNLP [39], to annotate the sentences with grammar dependencies. Grammar dependencies are a set of dependencies between parts of a sentences. Considering a phrase, “program a game”, this phrase has a dependency, which is called “dobj” in Stanford CoreNLP, where the governor is “program” and the dependent is “game”. For V-DO filter, we look for “dobj” dependencies which represent the verb/direct-object pattern.

For each sentence, we checked whether the sentence is begun with a “dobj” dependency. If the sentence is begun with a “dobj”, we mark the sentence as a “dobj” sentence. In the end, we have 32k commit messages that are “dobj” sentences.

4) Generating Training/Validation/Test Sets: We randomly selected 3k commits for testing, 3k commits for validation, and the rest 26k commits for training.

5) Selecting Vocabularies: NMT needs predefined vocabularies for commit messages and `diffs`. In the training set, the commit messages have 16k distinct tokens (words or punctuations) and the `diffs` have 65k distinct tokens. We selected all the 16k tokens in the commit messages to be the vocabulary of commit messages. We used the most frequent 50k tokens in the `diffs` to be the vocabulary of `diffs`. All the tokens that are not in the `diff` vocabulary only occur once in the training set. Additionally, the vocabulary size of 50k is often used by other NMT models [36].

B. NMT Training and Testing

In this section, we describe how we trained and tested an NMT model for generating commit messages.

1) Model: We used Nematus [51] in our work because it is robust, easy to use, and produced best constrained systems for seven translation directions (e.g., English to German, etc.) in WMT 2016 shared news translation task [53]. Nematus is based on Theano [56], and implements the *attentional RNN encoder-decoder* (see Section III-C) with several implementation differences [51].

2) Training Setting: We borrowed the training setting that Sennrich *et al.* used to produce the best translation systems in WMT 2016 [53]. The training goal is cross-entropy minimization [49]. The learning algorithm is stochastic gradient descent (SGD) with Adadelta [59], which automatically adapts the learning rate. The size of minibatches is 80; the size of word embeddings is 512; the size of hidden layers is 1024. For each epoch, the training set is reshuffled. The model is validated every 10k minibatches by BLEU [45], which is a commonly used similarity metric for machine translation. The maximum number of epochs is 5k; the maximum number of minibatches is 10M; and early stopping is used [51]. During the training, the model is saved every 30k minibatches. So after the training, a list of models are saved and the ensemble results of the last four models are used for evaluation.

One key difference between our and Sennrich *et al.*'s training processes is that Sennrich *et al.* used maximum sentence length of 50 for all the languages; we used 30 for commit messages and 100 for `diffs` as explained in Section IV-A2.

3) Training Details: We trained on the training set of 26k pairs of commit messages and `diffs`, with a validation set of 3k pairs. We conducted the training on an Nvidia GeForce GTX 1070 with 8GB memory. The learning algorithm stopped at 210k minibatches. Because a model is saved every 30k minibatches, seven models are saved from this training. The training process took 38 hours.

4) Testing Details: While we describe our evaluation in the next section, certain technical details are relevant here. We ran Nematus with the last four saved models on the testing set and we obtained the ensemble result. We used the same GPU as we used in training. The testing process took 4.5 minutes. We note that we followed the standard evaluation procedure for NMT and used a test set of 3k [36], [52], [10].

V. Evaluation Using An Automatic Metric

In this section, we evaluate the generated messages from our approach that we described in the last section. Our objective is to assess the similarity between the generated messages and the reference messages in the test set. This section corresponds to Part B in the paper overview Figure 1. Note that this evaluation is distinct from the experiment with human evaluators that we describe in Section VI, which is also a component of "Part B." In this section we ask:

RQ1 Compared to the messages generated by a baseline, are the messages generated by the NMT model more or less similar to the reference messages?

RQ2 Are the messages generated by the NMT model more or less similar to the reference messages when V-DO filter is enabled or disabled?

We ask RQ1 to evaluate the NMT model compared to a baseline, which we describe in the following subsection. We ask RQ2 in order to evaluate the impact of V-DO filter. In the following subsections, we first introduce the baseline for RQ1. Then, we introduce the metric for measuring similarity between two messages. Finally, we report our results for the research questions.

A. Baseline: MOSES

We used MOSES [30] as the baseline in RQ1. MOSES is a popular statistical machine translation software, which is often used as a baseline in evaluating machine translation systems [8], [29]. For example, Iyer *et al.* used MOSES as a baseline when they evaluated Code-NN [23]. To run MOSES for translating `diffs` to commit messages, we trained a 3-gram language model using KenLM [19], [20], which is the same procedure in the study of Iyer *et al.* [23]. We did not use Code-NN as a baseline, because, in our pilot study of running Code-NN [23] to generate commit messages, Code-NN did not generate comparable results. A possible reason is that Code-NN needs parsing source sequences and `diffs` are not suitable for parsing.

B. Similarity Metric: BLEU

BLEU [45] is widely used to measure the similarity between two sentences in evaluation of machine translation systems [28], [36], [35]. Additionally, BLEU is recommended for assessing an entire test set instead of a sentence [45]. The calculation of BLEU needs the modified n-gram precisions. For any n, the modified n-gram precision is calculated by:

$$p_n = \frac{\sum\limits_{(gen,ref)\in test} \sum\limits_{ngram\in gen} Cnt_{clip}(ngram)}{\sum\limits_{(gen,ref)\in test} \sum\limits_{ngram\in gen} Cnt_{gen}(ngram)} \tag{8}$$

$$\begin{aligned} Cnt_{clip}(ngram) = \\ min(Cnt_{gen}(ngram), Cnt_{ref}(ngram)) \end{aligned} \tag{9}$$

where $test$ is the set of pairs of the generated and the reference messages in the test set; gen is the set of distinct n-grams in a generated message; Cnt_{clip} is defined in Equation (9); Cnt_{gen} is the number of occurrences of an n-gram in a generated message; similarly, Cnt_{ref} is the number of the occurrences of an n-gram in a reference message. Then, BLEU is:

$$BLEU = BP \cdot exp(\sum_{n=1}^{N} \frac{1}{N} log(p_n)) \tag{10}$$

$$BP = \begin{cases} 1 & \text{if } c > r \\ e^{(1-r/c)} & \text{if } c \leq r \end{cases} \tag{11}$$

where N is the maximum number of grams; p_n is defined in Equation (8); BP is defined in Equation (11); r is the sum of the lengths of all the reference messages; c is the sum of the lengths of the generated messages. BLEU scores range from 0 to 100 (in percent). The default value of N is 4, which is used in our evaluation and is commonly used in other evaluations [28], [52], [36], [23], [35], [15].

C. RQ1: Compared to the Baseline

The first two rows in Table I list the BLEU scores of MOSES and the NMT model we trained in Section IV-B, which we refer to as *NMT1*. The BLEU score of our model is 31.92 while the BLEU score of MOSES is 3.63, so according to the BLEU metric, the messages generated by the NMT model are more similar to the reference messages than the messages generated by the baseline. One key reason that the attentional NMT model outperforms MOSES is that MOSES does not handle well very long source sequences with short target sequences. Particularly, MOSES depends on Giza++ [44] for word alignments between source and target sequences, and Giza++ becomes very inefficient when a source sequence is 9

TABLE I: BLEU Scores (%) of MOSES and Our Models on the Test Set

Model	BLEU	Len_{Gen}	Len_{Ref}	p_1	p_2	p_3	p_4
MOSES	3.63	129889	22872	8.3	3.6	2.7	2.1
NMT1	31.92	24344	22872	38.1	31.1	29.5	29.7
NMT2	32.81	21287	22872	40.1	34.0	33.4	34.3
	23.10*	20303	18658	30.2	23.3	20.7	19.6

MOSES is the baseline model. NMT1 is the NMT model with V-DO filter described in Section IV-B. NMT2 is a model trained without V-DO filter described in Section V-D. Len_{Gen} is the total length of the generated messages (c in Equation (11)). Len_{Ref} is the total length of the reference messages (r in Equation (11)). The modified n-gram precision p_n, where $n = 1, 2, 3, 4$, is defined in Equation (8).
* This BLEU score is calculated on a test set that is not V-DO filtered described in Section V-D. The other BLEU scores are tested on a V-DO filtered test set described in Section IV-A4.

TABLE II: BLEU Scores (%) on `diffs` of Different Lengths

Diff Length	BLEU	Len_{Gen}	Len_{Ref}	p_1	p_2	p_3	p_4
≤ 25	6.46	870	655	18.6	6.9	4.3	3.1
$> 25, \leq 50$	9.31	3627	3371	23.1	10.8	6.6	4.5
$> 50, \leq 75$	12.67	4779	4418	24.8	14.1	9.8	7.6
> 75	43.33	15068	14428	47.1	42.3	41.7	42.3

See Table I for explanation of each column name. The BLEU scores are calculated based on the test results generated by Model1, the NMT model with V-DO filter trained in Section IV-B.

times longer than the target sequence or vice versa [4]. Table I shows that the total length of the generated messages (Len_{Gen} in Table I) of MOSES is much longer than the total length of the reference messages, which may cause the modified n-gram precisions (p_1, p_2, p_3, and p_4), of MOSES to be small.

To further examine the messages generated by our model, we split the test set by the lengths of the `diffs` into four groups and calculated BLEU scores separately for each group. Figure 5 shows the distribution of the lengths of `diffs` in the test set and Table II shows the BLEU scores for the `diffs`. This table shows that the `diffs` that have more than 75 tokens have the highest BLEU score. One possible reason is that there are many more `diffs` that have more than 75 tokens than the other smaller `diffs`. Figure 6 shows the distribution of the `diff` lengths in the training set. This figure shows that the training set is populated by larger `diffs`, which may cause the model to fit the larger `diffs` better.

In Table II, the modified 4-gram precision, p_4, is 7.6 when `diff` lengths are between 25 and 50, and becomes 42.3 when `diff` lengths are larger than 75. This increase of p_4 means that the number of the 4-grams that are shared by the generated and reference messages increase dramatically when the lengths of `diffs` increase to more than 75 tokens. In contrast, p_4 changes much less (3.1 to 4.5, 4.5 to 7.6) in other cases.

D. RQ2: Impact of V-DO Filter

Besides NMT1 (the NMT model trained with V-DO filter in Section IV), we trained another model without V-DO filter, which we refer to as *NMT2*. In this subsection, we compare NMT1 and NMT2 to see the impact of V-DO filter.

1) Data Set and Training Process for NMT2: Without V-DO filter, the data set has 75k commits. First, we extracted the test set that is used by NMT1 so that we can compare the test results. Then, from the remaining 72k commits, we randomly selected 3k commits to be another test set, which may contain messages that do not follow the V-DO pattern. We refer to the first test set as *Test1* (with V-DO filter), and the second test set as *Test2* (without V-DO filter).

Then, we randomly selected 3k for validation and used the rest 66k commits for training. We note that the training set of NMT1 has only 26k commits, so NMT2 has 2.5 times more training data than NMT1. The training set includes 45k distinct tokens in commit messages and 110k distinct tokens in `diffs`. Similar to the vocabulary setting we used in Section IV-A4, we used all the 45k tokens to be the vocabulary of commit messages. We used the most frequent 100k tokens in `diffs` to be the vocabulary of `diffs`. All the tokens that are not included in the vocabulary only occur once in the training set. We followed the same process described in Section IV-B. The training process took 41 hours. The testing process for Test1 took 21.5 minutes and Test2 took 20 minutes.

2) Results: The third and fourth rows in Table I show the BLEU scores of NMT2 on Test1 and Test2, which are 32.81 and 23.10 respectively. Comparing the BLEU scores of NMT1 and Test1, the result shows that the messages generated by NMT2 are more similar to the reference messages in Test1. This finding indicates that although the training set without V-DO filter has low-quality messages, there are valuable commits that do not follow the V-DO pattern but help the NMT model improve over Test1 which follow the V-DO pattern.

However, the BLEU score of Test2 is about 10 percent lower than the BLEU score of Test1, which means that NMT2 does not perform well over the commits that do not follow the V-DO pattern. For example, a reference message in Test2 is "7807cb6

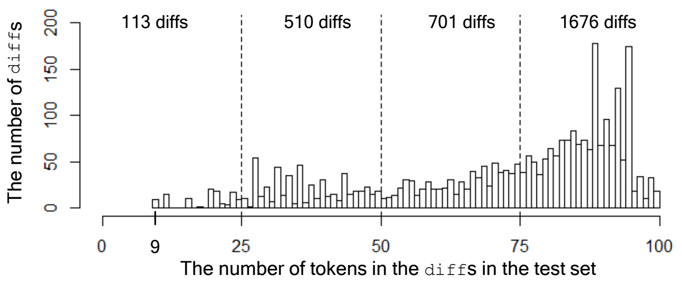


Fig. 5: The distribution of the lengths of `diffs` in the test set

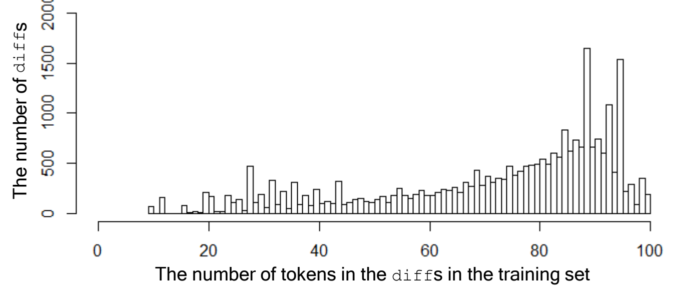


Fig. 6: The distribution of the lengths of `diffs` in the training set

ca7a229", which should be version numbers. For such reference messages in Test2, the NMT model cannot generate the same version numbers and is not meant to generate such numbers. However, similar messages in the training set cause the NMT model to try to generate such numbers for commit messages. For example, a generated message in Test2 is "Dd38b1cc2 92007d1d7" while the reference message is "Run only on jdk7 for the moment".

VI. Human Evaluation

In this section, we ask human experts to evaluate the generated messages by the NMT model we described in Section IV. In Section V, we evaluated our model by the automatic metric, BLEU. Our human study complements the evaluation that uses BLEU in two ways. First, although BLEU is a widely used metric that enables us to compare our model with others and to deliver reproducibility, BLEU is not recommended for evaluating individual sentences [45]. Our human study can show how our model perform on individual messages. Second, BLEU calculates the textual similarity between the generated and the reference messages, while the human study can evaluate the semantic similarity.

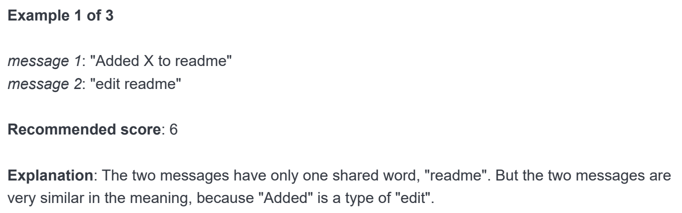
In this study, we hired 20 participants for 30 minutes each to evaluate the similarity in a survey study. Two participants are computer science Ph.D. students and 18 participants are professional programmers with 2 to 14 years experience. In the rest of this subsection, we describe our survey design, the process of conducting the survey, and the survey results.

A. Survey Design

We introduce our survey in the first page as: "This survey will ask you to compare two commit messages by their meaning. You will be able to select a score between 0 to 7, where 0 means there is no similarity and 7 means that two messages are identical." We permitted the participants to search the internet for unfamiliar concepts. Then, we gave three scoring examples with recommended scores of 6, 3, and 1. Due to space limit, we present only the first example in Figure 7 (all the other examples are available in our online appendix, Section XI). Then, in the remaining pages of the survey, each page has one pair of the messages, and we asked the participants to score the similarity by meaning. Note that the participants do not know who/what generated the messages. The order of the messages in every page is randomly decided. In the end of the page, there is an optional text box for the participants to enter their justifications. A formal qualitative study about the participants' comments will need to be performed in the future but is beyond the scope of this study. Figure 8 shows one page of the survey.

B. Survey Procedure

First, the pairs of generated/reference messages are randomly ordered in a list. Then, for each participant, a survey is generated with the messages in the list from a given starting point. For example, for the first three participants, the surveys are generated with the messages starting from the first pair in the list. In 30 minutes, the first participant was able to score 107 pairs; the second participant was able to score 61 pairs; the third participant was able to score 99 pairs. So the first 61 pairs of messages were evaluated by three participants. For the fourth participant, we generated a survey starting from the 62th pair and the participant stopped at 99th pair in 30 minutes. So after the first four participants, we have 99 pairs scored by three participants. Although it would be ideal if we obtain three scores for every pair, we did not enforce all the pairs being scored by three participants because we want to have more pairs scored with the limited number of participants. In the end, 226 pairs were scored by three participants, 522 pairs were scored by two participants, and 235 pairs were scored by one participant.

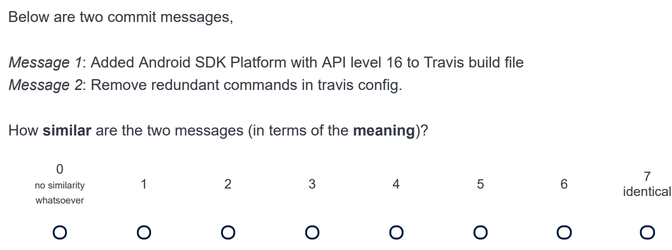

Example 1 of 3

message 1: "Added X to readme"
message 2: "edit readme"

Recommended score: 6

Explanation: The two messages have only one shared word, "readme". But the two messages are very similar in the meaning, because "Added" is a type of "edit".

Fig. 7: An scoring example we gave to the participants in the survey study.


Below are two commit messages,

Message 1: Added Android SDK Platform with API level 16 to Travis build file
Message 2: Remove redundant commands in travis config.

How **similar** are the two messages (in terms of the **meaning**)?

0 no similarity whatsoever	1	2	3	4	5	6	7 identical
○	○	○	○	○	○	○	○

Fig. 8: One of the pages that we ask the participants to score the similarity. There is an optional text box for the participants to write their justifications in the end of the page. This text box is omitted due to space limit.

C. Results

Figure 9 shows the distribution of the median scores of the semantic similarity of the generated/reference messages. To be conservative, we round down the median scores. For example, if a generated message has two scores, 1 and 2, and the median score is 1.5, we round down the median score to 1. In total, 983 generated commit messages have scores made by the participants. Zero and seven are the two most frequent scores. There are 248 messages scored 0 and 234 messages scored 7, which shows that the performance of our model tends to be either good or bad.

VII. Quality Assurance Filter

Based on the results from our study with human evaluators (Section VI), we propose a quality assurance filter (QA filter) to automatically detect the `diffs` for which the NMT model does not generate good commit messages. By building this filter, we investigate whether it is possible to automatically learn the cases where our NMT model does not perform well.

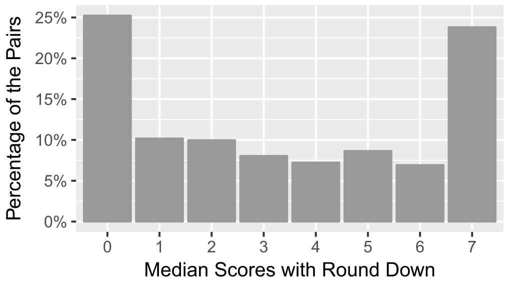


Fig. 9: The distribution of the median scores obtained in the human study. There are 983 scores in the figure. Each score is the median score of the scores made by one to three human experts for a generated message. The scores range from 0 to 7, where 0 denotes the generated message is not similar to the reference message at all, and 7 denotes the generated message is identical to the reference message. The most frequent scores are 0 and 7. There are 248 messages scored 0 and 234 messages scored 7. For the rest of the scores, the number of messages ranges from 68 to 100.

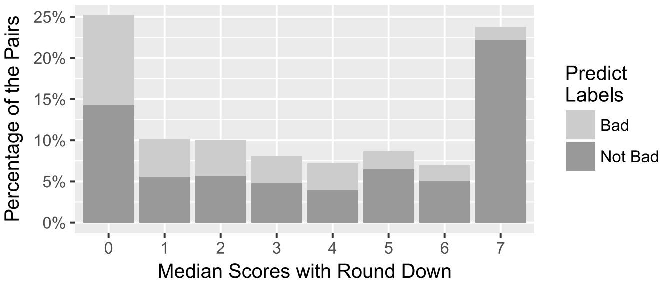


Fig. 11: The predict results of the cross evaluation of QA filter. QA filter reduced 108 messages that are scored 0, 45 messages that are scored 1, 42 messages that are scored 2, 32 messages that are scored 3, 32 messages that are scored 4, 21 messages that are scored 5, 18 messages that are scored 6, and 16 messages that are scored 7. We note that although we trained the QA filter with binary labels, "bad" and "not bad", the evaluation result shows that QA filter is able to reduce more messages for lower scores.

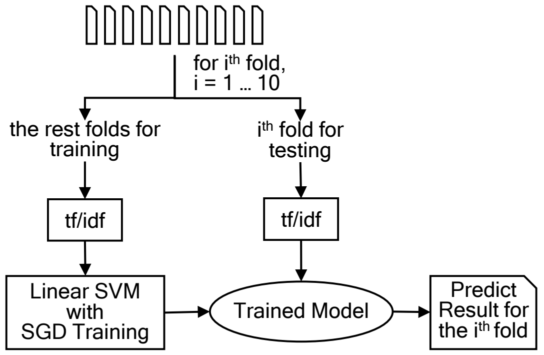
In this section, we describe the method of our filter, how we evaluate the filter, and the performance of the filter. This section corresponds to Part C in the paper overview Figure 1.

A. QA Filter

Our method of QA filter has three steps. First, we prepared the gold set. We used the evaluated messages and the corresponding `diff`s in the human study as our gold set. For each `diff` and the corresponding generated message, there is a score we obtained in the human study (Figure 9) that indicates whether the generated message for the `diff` is similar to the reference message (i.e., the actual human-written message). To be conservative, we labeled the `diff`s that have scores of zero or one as "bad" and all the other `diff`s as not "bad".

Second, we extracted the features of the `diff`s. We used *term frequency/inverse document frequency* (tf/idf) for every word in a `diff` as the features. Tf/idf is widely used in machine learning for text processing [18], which is computed based on the frequency of a word in a `diff` and whether the word is common in the other `diff`s.

Finally, we used the data set of `diff`s and their labels to train a linear SVM using stochastic gradient descent (SGD) as the learning algorithm. After we trained the SVM, to predict whether the NMT model will generate a "bad" commit message for a `diff`, we extract tf/idfs from the `diff` and run the trained SVM with the tf/idfs.



for i^{th} fold, i = 1 ... 10
the rest folds for training
i^{th} fold for testing
tf/idf
tf/idf
Linear SVM with SGD Training
Trained Model
Predict Result for the i^{th} fold

Fig. 10: Outline of our cross-validation process.

B. Cross-Validation Evaluation

Figure 10 illustrates our 10-fold cross-validation process. We shuffled the gold set first, and split the gold set into 10 folds. For each fold, we trained a SVM model on the other 9 folds, and tested the SVM model on the one fold. In the end, we obtained the test results for every fold. Figure 11 shows the predicts of all the folds. In terms of detecting `diff`s for which the NMT model will generate "bad" messages, QA filter has 44.9% precision and 43.8% recall. Furthermore, if we label the messages with scores of 6 or 7 as "good", in this evaluation, QA filter reduced 44% of the "bad" messages at a cost of 11% of the "good" messages.

VIII. Example Result

Table III shows a representative example of a generated message that was rated highly by the human experts. It includes the generated and reference messages, three scores made by three participants, and the corresponding `diff`. In this example, the reference message refers to a replacement of a call to a function called `deactivate()` with a call to a function `close()`. To a human reader, that is evident from the `diff`: a call to `deactivate()` is removed and a call to `close()` is added. The NMT algorithm also picked up on this change, generating text "Close instead of mCursor.Deactivate."

IX. Threats to Validity

One threat to validity is that our approach is experimented on only Java projects in Git repositories, so they may not be representative of all the commits. However, Java is a popular programming language [3], [1], [2], which is used in a large number of projects. In the future, we will extend our approach to other programming languages.

Another threat to validity is the quality of the commit messages. We collected actual human-written commit messages from Github, and used V-DO filter to obtain a set of relatively good-quality commit messages. But the human-written messages may not contain all the useful information that should

TABLE III: Example Result

Diff:

`-- a/core/.../CursorToBulkCursorAdaptor.java`
`+++ b/core/.../CursorToBulkCursorAdaptor.java`
`@@ -143,8 +143,7 @@ public final class CursorToBulkCursorAdaptor ...`
` public void close() {`
` maybeUnregisterObserverProxy();`
`- mCursor.deactivate();`
`-`
`+ mCursor.close();`
` }`
` public int requery(IContentObserver observer, ...`

Generated Message: "CursorToBulkCursorAdapter . Close must call mCursor . Close instead of mCursor . Deactivate . "
Reference Message: "Call close ( ) instead of deactivate ( ) in CursorToBulkCursorAdaptor . close ( ) "
Scores: 7, 6, 7

be in a commit message. And programmers are often lazy and write poor messages [11]. However, our objective in this paper is to generate commit messages that can be learned from the history of the repositories. Further improvement on human-written messages falls outside the scope of this paper.

Another threat to validity is about the human study because of the limited number of the participants. We cannot guarantee that every final score for a generated commit message is fair. We tried to mitigate this threat by hiring as many professional programmers as we can, and having 23% of the evaluated messages scored by three participants and 53% of the evaluated messages scored by two participants.

Additionally, our use of Stanford CoreNLP [39] may not yield the best performance of Stanford CoreNLP. Because Stanford CoreNLP is trained on natural languages, CoreNLP may not perform well on the commit messages, which contain special identifiers, such as variable names, and software programming jargons. Several studies in sentiment analysis for software engineering [25], [43] have discussed the problem of using NLP tools without customization for software engineering. In our future work, we may work on this problem so that we can provide NMT with better training data.

X. Discussion and Conclusion

The key advancement that this paper makes to the state-of-the-art is a technique to generate short commit messages that summarize the high-level rationale for a change to software. As we note in Section I-A, we do not claim to be able to provide new insights for completely novel changes to software – that task is likely to remain in human hands for the foreseeable future. Instead, we learn from knowledge stored in a repository of changes that have already been described in commit messages. Several authors in the related literature have observed that many code changes follow similar patterns, and have a similar high-level rationale (e.g., [40], [24]). Traditionally programmers still need to manually write commit messages from scratch, even in cases where a commit has a rationale that has been described before. What this paper does is automate writing commit messages based on knowledge in a repository of past changes.

Our strategy was, in a nutshell, to 1) collect a large repository of commits from large projects, 2) filter the commits to ensure relatively high-quality commit messages, and 3) train a Neural Machine Translation algorithm to "translate" from `diffs` to commit messages using the filtered repository. We then evaluated the generated commit messages in two ways. First we conducted an automated evaluation using accepted metrics and procedures from the relevant NMT literature (Section V). Second, as a verification and for deeper analysis, we also conducted an experiment with human evaluators (Section VI).

What we discovered is that the NMT algorithm succeeded in identifying cases where the commit had a similar rationale to others in the repository. The evidence for this is the large bar for item 7 in Figure 9 – it means that the human evaluators rated a large number of the generated messages as very closely matching the reference messages. However, the algorithm also generated substantial noise in the form of low quality messages (note the large bar for item 0). A likely explanation is that these include the cases that involve new insights which the NMT algorithm is unable to provide. While creating these new insights from the data is currently beyond the power of existing neural network-based machine learning (a problem observed across application domains [14]), at a minimum we would like to return a warning message to the programmer to indicate that we are unable to generate a message, rather than return a low quality message. Therefore we created a Quality Assurance filter in Section VII. This filter helped reduce the number of low quality predictions, as evident in the reduced bar for item 0 in Figure 11.

While we do view our work as meaningfully advancing the state-of-the-art, we by no means claim this work is definitive or completed. We release our complete data set and implementation via an online appendix, noted at the end of Section IV. Our hope is that other researchers will use this data set and implementation for further research efforts. Generally speaking, future improvements are likely to lie in targeted training for certain types of commits, combined with detection of change types. It is probable that very high quality predictions are possible for some types of software changes, but not others. This work provides a foundation for those and other future developments.

XI. Reproducibility

Our data sets, scripts, and results are accessible via: **https://sjiang1.github.io/commitgen**

Acknowledgments

This work was partially supported by the NSF CCF-1452959 and CNS-1510329 grants, and the Office of Naval Research grant N000141410037. Any opinions, findings, and conclusions expressed herein are the authors' and do not necessarily reflect those of the sponsors.

References

[1] PYPL popularity of programming language. http://pypl.github.io/PYPL.html. Accessed: 2017-05-11.

[2] Stackoverflow developer survey results 2016. https://insights.stackoverflow.com/survey/2016#technology. Accessed: 2017-05-11.

[3] TIOBE index for may 2017. https://www.tiobe.com/tiobe-index/. Accessed: 2017-05-11.

[4] *Moses Specification*, 5 2012. Deliverable D1.1.

[5] C. V. Alexandru, S. Panichella, and H. C. Gall. Replicating parser behavior using neural machine translation. In *2017 IEEE 25th International Conference on Program Comprehension (ICPC)*, 2017.

[6] D. Bahdanau, K. Cho, and Y. Bengio. Neural machine translation by jointly learning to align and translate. *CoRR*, abs/1409.0473, 2014.

[7] R. P. Buse and W. R. Weimer. Automatically documenting program changes. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ASE '10, pages 33–42, New York, NY, USA, 2010. ACM.

[8] C. Callison-Burch, P. Koehn, C. Monz, and O. F. Zaidan. Findings of the 2011 workshop on statistical machine translation. In *Proceedings of the Sixth Workshop on Statistical Machine Translation*, WMT '11, pages 22–64, Stroudsburg, PA, USA, 2011. Association for Computational Linguistics.

[9] K. Chen, P. Liu, and Y. Zhang. Achieving accuracy and scalability simultaneously in detecting application clones on android markets. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 175–186, New York, NY, USA, 2014. ACM.

[10] K. Cho, B. van Merrienboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio. Learning phrase representations using rnn encoder–decoder for statistical machine translation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1724–1734, Doha, Qatar, October 2014. Association for Computational Linguistics.

[11] L. F. Cortés-Coy, M. Linares-Vásquez, J. Aponte, and D. Poshyvanyk. On automatically generating commit messages via summarization of source code changes. In *2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation*, pages 275–284, Sept 2014.

[12] R. Dyer, H. A. Nguyen, H. Rajan, and T. N. Nguyen. Boa: A language and infrastructure for analyzing ultra-large-scale software repositories. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 422–431, May 2013.

[13] B. Fluri, M. Wursch, and H. C. Gall. Do code and comments co-evolve? on the relation between source code and comment changes. In *Proceedings of the 14th Working Conference on Reverse Engineering*, WCRE '07, pages 70–79, Washington, DC, USA, 2007. IEEE Computer Society.

[14] I. Goodfellow, Y. Bengio, and A. Courville. *Deep learning*. MIT Press, 2016.

[15] X. Gu, H. Zhang, D. Zhang, and S. Kim. Deep api learning. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pages 631–642, New York, NY, USA, 2016. ACM.

[16] J. Guo, J. Cheng, and J. Cleland-Huang. Semantically enhanced software traceability using deep learning techniques. In *Proceedings of the 2017 International Conference on Software Engineering*, ICSE '17, 2017.

[17] S. Haiduc, J. Aponte, L. Moreno, and A. Marcus. On the use of automated text summarization techniques for summarizing source code. In *Proceedings of the 2010 17th Working Conference on Reverse Engineering*, WCRE '10, pages 35–44, Washington, DC, USA, 2010. IEEE Computer Society.

[18] S. Haiduc, J. Aponte, L. Moreno, and A. Marcus. On the use of automated text summarization techniques for summarizing source code. In *2010 17th Working Conference on Reverse Engineering*, pages 35–44, Oct 2010.

[19] K. Heafield. Kenlm: Faster and smaller language model queries. In *Proceedings of the Sixth Workshop on Statistical Machine Translation*, WMT '11, pages 187–197, Stroudsburg, PA, USA, 2011. Association for Computational Linguistics.

[20] K. Heafield, I. Pouzyrevsky, J. H. Clark, and P. Koehn. Scalable modified Kneser-Ney language model estimation. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics*, Sofia, Bulgaria, 8 2013.

[21] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu. On the naturalness of software. In *Proceedings of the 2012 International Conference on Software Engineering*, ICSE 2012, pages 837–847, Piscataway, NJ, USA, 2012. IEEE Press.

[22] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural Computation*, 9:1735–1780, 1997.

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

[24] S. Jiang and C. McMillan. Towards automatic generation of short summaries of commits. In *2017 IEEE 25th International Conference on Program Comprehension (ICPC)*, 2017.

[25] R. Jongeling, S. Datta, and A. Serebrenik. Choosing your weapons: On sentiment analysis tools for software engineering research. In *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 531–535, Sept 2015.

[26] M. Kajko-Mattsson. A survey of documentation practice within corrective maintenance. *Empirical Softw. Engg.*, 10(1):31–55, Jan. 2005.

[27] T. Kamiya, S. Kusumoto, and K. Inoue. Ccfinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, 28(7):654–670, Jul 2002.

[28] G. Klein, Y. Kim, Y. Deng, J. Senellart, and A. M. Rush. OpenNMT: Open-Source Toolkit for Neural Machine Translation. *ArXiv e-prints*.

[29] P. Koehn and H. Hoang. Factored translation models. In *Proceedings of the 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning (EMNLP-CoNLL)*, pages 868–876, Prague, Czech Republic, June 2007. Association for Computational Linguistics.

[30] P. Koehn, H. Hoang, A. Birch, C. Callison-Burch, M. Federico, N. Bertoldi, B. Cowan, W. Shen, C. Moran, R. Zens, et al. Moses: Open source toolkit for statistical machine translation. In *Proceedings of the 45th annual meeting of the ACL on interactive poster and demonstration sessions*, pages 177–180. Association for Computational Linguistics, 2007.

[31] J. Krinke. Identifying similar code with program dependence graphs. In *Proceedings Eighth Working Conference on Reverse Engineering*, pages 301–309, 2001.

[32] A. N. Lam, A. T. Nguyen, H. A. Nguyen, and T. N. Nguyen. Combining deep learning with information retrieval to localize buggy files for bug reports (n). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 476–481, Nov 2015.

[33] T. D. B. Le, M. Linares-Vasquez, D. Lo, and D. Poshyvanyk. Rclinker: Automated linking of issue reports and commits leveraging rich contextual information. In *2015 IEEE 23rd ICPC*, pages 36–47.

[34] M. Linares-Vásquez, L. F. Cortés-Coy, J. Aponte, and D. Poshyvanyk. Changescribe: A tool for automatically generating commit messages. In *2015 IEEE/ACM 37th IEEE ICSE*, volume 2, pages 709–712.

[35] W. Ling, P. Blunsom, E. Grefenstette, K. M. Hermann, T. Kočiský, F. Wang, and A. Senior. Latent predictor networks for code generation. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 599–609, Berlin, Germany, August 2016. Association for Computational Linguistics.

[36] M. Luong and C. D. Manning. Achieving open vocabulary neural machine translation with hybrid word-character models. *CoRR*, abs/1604.00788, 2016.

[37] M. Luong, H. Pham, and C. D. Manning. Effective approaches to attention-based neural machine translation. *CoRR*, abs/1508.04025, 2015.

[38] W. Maalej and H. J. Happel. Can development work describe itself? In *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*, pages 191–200, May 2010.

[39] C. D. Manning, M. Surdeanu, J. Bauer, J. Finkel, S. J. Bethard, and D. McClosky. The Stanford CoreNLP natural language processing toolkit. In *Association for Computational Linguistics (ACL) System Demonstrations*, pages 55–60, 2014.

[40] A. Mockus and L. G. Votta. Identifying reasons for software changes using historic databases. In *Proceedings 2000 International Conference on Software Maintenance*, pages 120–130, 2000.

[41] L. Moreno, G. Bavota, M. D. Penta, R. Oliveto, A. Marcus, and G. Canfora. ARENA: An approach for the automated generation of release notes. *IEEE Transactions on Software Engineering*, 43(2):106–127, Feb 2017.

[42] N. Nazar, Y. Hu, and H. Jiang. Summarizing software artifacts: A literature review. *Journal of Computer Science and Technology*, 31(5):883–909, Sep 2016.

[43] N. Novielli, F. Calefato, and F. Lanubile. The challenges of sentiment detection in the social programmer ecosystem. In *Proceedings of the 7th International Workshop on Social Software Engineering*, SSE 2015, pages 33–40. ACM, 2015.

[44] F. J. Och and H. Ney. A systematic comparison of various statistical alignment models. *Computational Linguistics*, 29(1):19–51, 2003.

[45] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu. Bleu: A method for automatic evaluation of machine translation. In *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics*, ACL '02, pages 311–318, Stroudsburg, PA, USA, 2002. Association for Computational Linguistics.

[46] G. N. Pengcheng Yin. A syntactic neural model for general-purpose code generation. In *Annual Meeting of the Association for Computational Linguistics*. Association for Computational Linguistics, 2017.

[47] S. Rastkar and G. C. Murphy. Why did this code change? In *Proceedings of the 2013 ICSE*, ICSE '13, pages 1193–1196, 2013.

[48] T. Roehm, R. Tiarks, R. Koschke, and W. Maalej. How do professional developers comprehend software? In *Proceedings of the 2012 International Conference on Software Engineering*, ICSE 2012, pages 255–265, Piscataway, NJ, USA, 2012. IEEE Press.

[49] R. Y. Rubinstein and D. P. Kroese. *The cross-entropy method: a unified approach to combinatorial optimization, Monte-Carlo simulation and machine learning*. Springer Science & Business Media, 2013.

[50] R. Sennrich, O. Firat, K. Cho, A. Birch, B. Haddow, J. Hitschler, M. Junczys-Dowmunt, S. L"aubli, A. V. Miceli Barone, J. Mokry, and M. Nadejde. Nematus: a toolkit for neural machine translation. In *Proceedings of the Software Demonstrations of the 15th Conference of the European Chapter of the Association for Computational Linguistics*, pages 65–68, Valencia, Spain, April 2017. Association for Computational pages 65–68, Valencia, Spain, April 2017. Association for Computational Linguistics.

[51] R. Sennrich, O. Firat, K. Cho, A. Birch, B. Haddow, J. Hitschler, M. Junczys-Dowmunt, S. Läubli, A. V. Miceli Barone, J. Mokry, and M. Nadejde. Nematus: a Toolkit for Neural Machine Translation. In *Proceedings of the Demonstrations at the 15th Conference of the European Chapter of the Association for Computational Linguistics*, 2017.

[52] R. Sennrich and B. Haddow. Linguistic input features improve neural machine translation. *CoRR*, abs/1606.02892, 2016.

[53] R. Sennrich, B. Haddow, and A. Birch. Edinburgh neural machine translation systems for WMT 16. *CoRR*, abs/1606.02891, 2016.

[54] G. Sridhara, E. Hill, D. Muppaneni, L. Pollock, and K. Vijay-Shanker. Towards automatically generating summary comments for java methods. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, ASE '10, pages 43–52, New York, NY, USA, 2010. ACM.

[55] I. Sutskever, O. Vinyals, and Q. V. Le. Sequence to sequence learning with neural networks. In Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 27*, pages 3104–3112. Curran Associates, Inc., 2014.

[56] Theano Development Team. Theano: A Python framework for fast computation of mathematical expressions. *arXiv e-prints*, abs/1605.02688, May 2016.

[57] M. White, M. Tufano, C. Vendome, and D. Poshyvanyk. Deep learning code fragments for code clone detection. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE 2016, pages 87–98, New York, NY, USA, 2016. ACM.

[58] M. White, C. Vendome, M. Linares-Vasquez, and D. Poshyvanyk. Toward deep learning software repositories. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, pages 334–345, May 2015.

[59] M. D. Zeiler. Adadelta: an adaptive learning rate method. *arXiv preprint arXiv:1212.5701*, 2012.