# Neural-machine-translation-based commit message generation: how far are we?

**Conference Paper** · September 2018

DOI: 10.1145/3238147.3238190

# Neural-Machine-Translation-Based Commit Message Generation: How Far Are We?

Zhongxin Liu
Zhejiang University
China
liu_zx@zju.edu.cn

Xin Xia
Monash University
Australia
xin.xia@monash.edu

Ahmed E. Hassan
Queen's University
Canada
ahmed@cs.queensu.ca

David Lo
Singapore Management University
Singapore
davidlo@smu.edu.sg

Zhenchang Xing
Australian National University
Australia
zhenchang.xing@anu.edu.au

Xinyu Wang
Zhejiang University
China
wangxinyu@zju.edu.cn

## ABSTRACT

Commit messages can be regarded as the documentation of software changes. These messages describe the content and purposes of changes, hence are useful for program comprehension and software maintenance. However, due to the lack of time and direct motivation, commit messages sometimes are neglected by developers. To address this problem, Jiang et al. proposed an approach (we refer to it as *NMT*), which leverages a neural machine translation algorithm to automatically generate short commit messages from code. The reported performance of their approach is promising, however, they did not explore why their approach performs well. Thus, in this paper, we first perform an in-depth analysis of their experimental results. We find that (1) Most of the test `diffs` from which *NMT* can generate high-quality messages are similar to one or more training `diffs` at the token level. (2) About 16% of the commit messages in Jiang et al.'s dataset are noisy due to being automatically generated or due to them describing repetitive trivial changes. (3) The performance of *NMT* declines by a large amount after removing such noisy commit messages. In addition, *NMT* is complicated and time-consuming. Inspired by our first finding, we proposed a simpler and faster approach, named *NNGen* (Nearest Neighbor Generator), to generate concise commit messages using the nearest neighbor algorithm. Our experimental results show that *NNGen* is over 2,600 times faster than *NMT*, and outperforms *NMT* in terms of BLEU (an accuracy measure that is widely used to evaluate machine translation systems) by 21%. Finally, we also discuss some observations for the road ahead for automated commit message generation to inspire other researchers.

## CCS CONCEPTS

• **Software and its engineering → Software maintenance tools**;

## KEYWORDS

Commit message generation, Nearest neighbor algorithm, Neural machine translation

**ACM Reference Format:**
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 2018 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE '18), September 3–7, 2018, Montpellier, France.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3238147.3238190

## 1 INTRODUCTION

In software projects, version control systems are widely used to manage the evolving code base. While committing a change to a version control system, developers document their changes using a commit message. A commit message is a free-form textual description of its corresponding change. The message may summarize *what* happened in the change and/or explain *why* the change was made [13, 44]. There is empirical evidence that the use of commit messages is commonplace in code that is managed with version control systems [13, 30, 44].

Documentation plays an important role in program comprehension and software maintenance [17, 49]. As the documentation of changes, commit messages can help developers understand the rationales behind changes before they dig into details [20, 23, 44, 64]. Commit messages also provide information to understand the evolution of software [13, 32]. However, due to the lack of direct motivation and time pressure, writing high-quality commit messages remains a neglected issue. Dyer et al. report that around 14% of the commit messages in 23K+ Java SourceForge projects were completely empty [18].

Many tools have been proposed to generate commit messages automatically [13, 16, 30, 38]. The commit messages created by them can assist developers in writing high-quality commit messages or replace empty commit messages. Given the `diff` of a change, most of these tools, e.g., DeltaDoc [13] and ChangeScribe [16, 38], are able to produce detailed messages which can answer what was changed and where this change happened. But their generated messages are verbose, and fail to reveal the rationale behind a change.

*ASE '18, September 3–7, 2018, Montpellier, France*

ACM ISBN 978-1-4503-5937-5/18/09…$15.00
https://doi.org/10.1145/3238147.3238190

It is hard to automatically generate high-quality commit messages, since answering why a change happened usually requires synthesis of different kinds of knowledge and context. However, recent studies report that commit messages follow some patterns [31, 44], and it is possible to learn patterns of software artifacts from large datasets [25]. Based on these findings, Jiang et al. proposed the adaption of a neural machine translation (NMT) technique to generate commit messages from change `diffs` [30]. In the remainder of the paper, for simplicity sake, we refer to their approach as *NMT*. *NMT* aims to learn how to write commit messages from prior changes and their commit messages. Different from prior work, *NMT* focuses on producing short messages which can reveal the rationales behind changes. Jiang et al. reported the performance of their approach using a dataset built from the top 1K Java projects in GitHub.

*NMT* has many advantages: (1) In contrast to existing commit message generation methods, *NMT* produces short summaries instead of exhaustive descriptions of changes. (2) *NMT* does not require manually defined templates, as needed by many prior tools, e.g., DeltaDoc and ChangeScribe. (3) *NMT* can generate commit messages for changes to many types of software artifacts, not only source code changes.

However, Jiang et al. did not explore why *NMT* performs so well in their paper. Understanding the applicable scenario of an approach can help us apply it in practice. So in this paper, we first investigate the rationale for *NMT*'s good performance.

Additionally, *NMT* is quite complex, and its training process is very slow. Jiang et al. spent 38 hours training their NMT model on an Nvidia GeForce GTX 1070 [30]. However, according to the suggestions of Fu and Menzies [19], it is a good practice to explore simple and fast techniques before applying deep learning methods on SE tasks. Therefore, we wish to investigate the construction of a much simpler and faster approach to address the same problem that is solved by *NMT*.

Our study aims to answer the following research questions:

**RQ1: Why does *NMT* perform so well?**

We conduct an analysis of the generated commit messages by *NMT* (using the data published on Jiang et al.'s website [1]). We randomly read 200 commits in Jiang et al.'s test results, and manually identify those high-quality generated commit messages by *NMT* (we call them *good messages*). Then, those identified *good messages* and their corresponding commits are further analyzed by us. From the analysis, we find the code `diffs` of most of the *good messages* are similar to one or more training `diffs` at the token level.

We also find that Jiang et al.'s dataset contains noisy commit messages, like messages that are automatically produced by other development tools, e.g., a continuous integration (CI) bot named *liferay-continuous-integration*, or messages that are written by human but contain little and redundant information, e.g., "update readme.md". Such a message describes neither what was changed in the readme file nor why the change happened, hence contains little information. In addition, since the information can be obtained easily by looking at the list of changed files, it is also redundant. It makes little sense to train and test approaches (e.g., *NMT*) for automated commit message generation on such noisy messages. Therefore, we manually derive the patterns of such noisy commit messages, and build a new dataset by deleting such noisy commit messages and their corresponding `diffs` from Jiang et al.'s dataset. We re-train *NMT* on this cleaned dataset and obtain a new model. Compared to the model trained on the original dataset, the performance of the new model declines by a large amount, and the BLEU score [51] (an accuracy measure that is widely used to evaluate machine translation systems) decreases by 55.5% from 31.92 to 14.19.

**RQ2: Can a simpler and faster approach outperform *NMT*?**

In RQ1, we found that the code `diffs` of most of the *good messages* are lexically similar to one or more `diffs` in the training set. Inspired by this finding, we propose a simpler and faster approach, named *NNGen* (Nearest Neighbor Generator), to automatically generate commit messages from `diffs`. Our approach is based on the nearest neighbor algorithm, and does not require a training process. To generate a commit message for a new `diff`, *NNGen* first finds the `diff` which is most similar to the new `diff`, i.e., the nearest neighbor, from the training set. Then it simply outputs the commit message of the nearest neighbor as the generated commit message.

Our experimental results show that *NNGen* outperforms *NMT* on Jiang et al.'s dataset and the cleaned dataset in terms of BLEU by a substantial margin. Moreover, it only takes 23 to 30 seconds to run *NNGen* on a CPU instead of 24 to 38 hours on a GPU, which means that *NNGen* is over 2,600 times faster than *NMT*. We also perform a human evaluation to compare *NNGen* and *NMT*. Our evaluation shows that *NNGen* performs better than *NMT*, and the improvement is significant.

Finally, we conduct a further analysis of automated commit message generation. We point out that only `diffs` and commit messages are not enough for this task. By answering the above research questions, we just move one step further, but there is still a long way to go.

The main contributions of this work are as follows:

(1) We perform an in-depth analysis of the experimental results in Jiang et al.'s work, and analyze the reasons of *NMT*'s good performance.
(2) We propose a simpler and faster approach called *NNGen* to generate short commit messages. *NNGen* is over 2,600 times faster than *NMT*, and significantly outperforms *NMT*.

The remainder of this paper is organized as follows. Section 2 introduces the background of our study. Section 3 describes our experimental settings, including research questions and dataset. Section 4-5 details our experiments and the experimental results of each research question respectively. Section 6 discusses the reason behind *NNGen*'s better performance, the cases where *NMT* outperforms *NNGen*, the implications of our study and threats to the validity of our reported findings. Section 7 discusses some observations for the road ahead for automated commit message generation. Section 8 surveys the related work. Section 9 concludes the paper.

# 2 BACKGROUND

## 2.1 Commit, `Diff`, Commit Messages

Jiang et al.'s dataset is extracted from Git repositories. Git [2] is one of the most popular version control systems. Each time a developer commits a change, Git will create a "commit" for this change and allow the developer to write a textual message called a "commit message" to describe the change. A commit in Git contains a change and a commit message (which may be empty). A change can be represented by a diff, which captures the difference between two program versions and can be generated using the *git diff* command in Git. In this work, by mentioning a commit, we are referring to the pair of a code diff and its corresponding commit message. Given a commit, we refer to its original commit message, which is extracted from the Git repository, as the *reference message*, the produced commit message by *NMT* as the *NMT message* and the produced commit message by *NNGen* as the *NNGen message*.

## 2.2 Jiang et al.'s NMT Approach

The NMT model adapted by Jiang et al. is the *attentional RNN Encoder-Decoder* model [11], which is an extension to the *RNN Encoder-Decoder* model [14]. The *RNN Encoder-Decoder* model was originally designed for translating between natural languages. There are two parts in this model: the encoder and the decoder, each of which is a Recurrent Neural Network (RNN). Given a source sentence, i.e., a sentence written in the source language, the encoder reads and encodes it into a fixed-length vector. This vector can be regarded as the intermediate representation of the source sentence, and contains the needed information for translation. The decoder outputs the target sentence from the encoded vector. The encoder and the decoder are jointly trained using a large number of pairs of source sentences and target sentences. In the machine translation community, this kind of dataset is referred to as a parallel corpora. Compared to the *RNN Encoder-Decoder* model, the *attentional RNN Encoder-Decoder* model introduces the attention mechanism to cope with long source sentences.

In Jiang et al.'s work, the source sentences are diffs, and the target sentences are *reference messages*. The parallel corpora are collected from GitHub, which contains pairs of historical diffs and the corresponding *reference messages*. After training on the special parallel corpora, Jiang et al.'s model can "translate" a new diff into a short textual description which may summarize the corresponding change.

## 2.3 BLEU

To align with Jiang et al., we use the BLEU-4 score [51] to evaluate the performance of *NNGen*. The BLEU score is an accuracy measure, that is widely used to assess the quality of machine translation systems [11, 14, 27, 29, 50]. The score first calculates the modified n-gram (for BLEU-4, n=1,2,3,4) precisions of a candidate sequence to the reference message, then measures the average modified n-gram precision with a penalty for overly short sentences. In our case, we regard a generated commit message (an *NMT message* or an *NNGen message*) as a candidate. Considering the fact that BLEU aims to match human judgment at a corpus level [51] and Jiang et al. use a corpus-level BLEU-4 score to evaluate *NMT*, we also calculate the BLEU-4 score at the corpus level.

In addition, *NNGen* leverages the BLEU-4 score internally to measure the similarity between two diffs. However, it calculates BLEU-4 score at the sentence level.

# 3 EXPERIMENTAL SETUP

## 3.1 Research Questions

Jiang et al. did not investigate why their approach performs so well. Understanding the reasons is important for applying *NMT* in practice. So, first of all, we want to investigate:

> RQ1: Why does *NMT* perform so well?

Additionally, *NMT* is quite complicated and its training process is very slow and costly (e.g., requiring specialized and dedicated hardware). Simple and fast methods are usually easier to be adopted in practice. Fu and Menzies also recommend the exploration of simple methods first while dealing with SE tasks [19]. Therefore, we would like to know:

> RQ2: Can a simpler and faster approach outperform *NMT*?

## 3.2 Dataset

Since we wish to investigate the reason behind *NMT*'s good performance and compare the performance of *NNGen* and *NMT*, we simply use Jiang et al.'s dataset to conduct our experiments. Jiang et al. have gratefully published their dataset [1]. To make our paper self-contained, we briefly describe the building process of Jiang et al.'s dataset in following paragraphs.

**Collecting Data:** Jiang et al. collected 2M commits from the most starred 1K Java projects in GitHub.

**Preprocessing:** They first extracted the first sentence of each collected commit message. Next, to reduce their vocabulary size and improve the performance of *NMT*, they removed commit ids from diffs, and removed issue ids from *reference messages*. Then, they removed merge commits, rollback commits and commits with a diff that is larger than 1MB. Finally, they broke *reference messages* and diffs into tokens. But they did not convert tokens into lowercase, and nor did they split the CamelCase tokens. After preprocessing, 1.8M commits remained.

**Filtering:** To apply the NMT algorithm, Jiang et al. needed to filter commits by length (i.e., the number of tokens in a sequence). They only kept commits with a diff length of no more than 100 and a *reference message* length of no more than 30. Only 75K commits meet these length requirements. In addition, Jiang et al. introduced the Verb-Direct Object (V-DO) filter for the *reference messages*. They did so because the NMT algorithm performs better on such pattern of messages. Their V-DO filter identifies the Verb-Direct Object pattern, e.g., "delete a method", through the "dobj" dependency in the Stanford CoreNLP library [40]. They removed the extracted sentences which do not begin with a "dobj" dependency. Jiang et al. only preserved 32K from the 75K messages that begin with a "dobj" dependency.

After preprocessing and filtering, Jiang et al. randomly divided the 32K commits into 3 sets, i.e., training set, validation set and test set. The training set contains 26K commits. The validation set and the test set each contain 3K commits.

# 4 RQ1: WHY DOES *NMT* PERFORM SO WELL?

## 4.1 Analyzing *NMT Messages*

To investigate RQ1, we closely analyze the generated commit messages by *NMT*, i.e., *NMT messages*. We first randomly select 200 commits from Jiang et al.'s test set. Then, the first author and a master student independently evaluate the *NMT messages* of these 200 commits.

In Jiang et al.'s work, they conducted a human study to evaluate the quality of *NMT messages*. Given a commit, human experts were

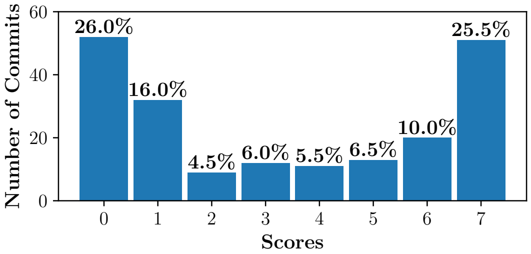
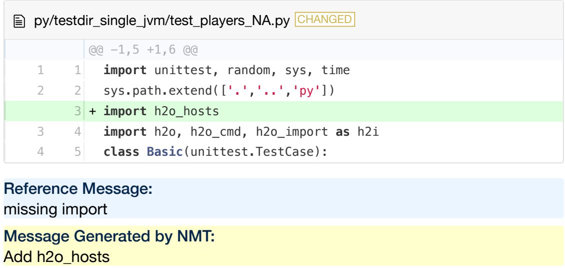


**Figure 1: Distribution of the scores of *NMT messages***



py/testdir_single_jvm/test_players_NA.py CHANGED

```
@@ -1,5 +1,6 @@
  import unittest, random, sys, time
  sys.path.extend(['.','..','py'])
+ import h2o_hosts
  import h2o, h2o_cmd, h2o_import as h2i
  class Basic(unittest.TestCase):
```

Reference Message:
missing import

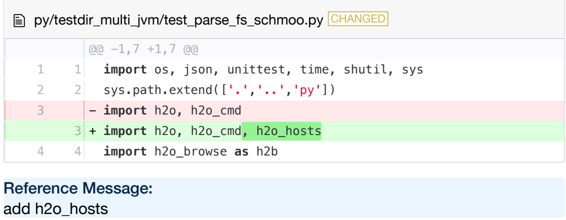
Message Generated by NMT:
Add h2o_hosts

**Figure 2: The commit of a *good message***

asked to read its *reference message* and *NMT message*, and give a score between 0 to 7 to measure the semantic similarity between the two messages. A score of 0 means that the two messages have nothing in common, and a score of 7 means that they are identical in meaning. To grade the 200 *NMT messages*, the two raters carefully read the scoring examples provided by Jiang et al. [1], and rated each *NMT messages* following Jiang et al.'s evaluation criterion. We find a high level of agreement between the two raters with a Cohen's Kappa coefficient [15] of 0.67, which shows a substantial agreement among the different raters. After rating, the two raters discussed their disagreements to reach consensus. Figure 1 shows the final distribution of the scores.

Next, we identify those high-quality commit messages generated by *NMT*. For simplicity sake, we refer to those messages as *good messages*. To align with Jiang et al., we regard the *NMT messages* that are scored 6 or 7 as *good messages*. We find 35.5% *good messages* from the 200 *NMT messages*, which is close to the results of Jiang et al.'s human study (30.7% *good messages*) on a different test set.

After picking out *good messages*, we carefully read these messages and their corresponding commits again to try to recognize some simple patterns in them. But we do not find any obvious textual patterns which are shared among all *good messages*. It appears that *NMT* can produce various types of high-quality messages. Considering that machine learning methods learn from the training set before predicting, we then search the training set to find similar commit messages for each *good messages* through its keywords, and try to gain some insights from these similar training commit messages. We find that, for nearly every *good messages* (70 out of 71), we can find out one or more training commit messages that are nearly identical to the *good message* and the `diffs` of these training messages are similar to that of the *good message* at the token level. For example, Figure 2 presents a test commit, of which the *NMT message* is a *good message*. We refer to this *good message* as *message*$_1$. Figure 3 shows a commit, which is found in the training set by searching the *reference messages* that contain "h2o_hosts". We call the *reference message* in Figure 3 *message*$_2$. We can see that without considering the case, *message*$_1$ is identical to *message*$_2$, and *message*$_1$'s `diff` is similar to that of *message*$_2$ at the token level. This finding is surprising since it means that even using such a complicated NMT algorithm, *NMT* is still no better than a nearest-neighbor-based recommender.



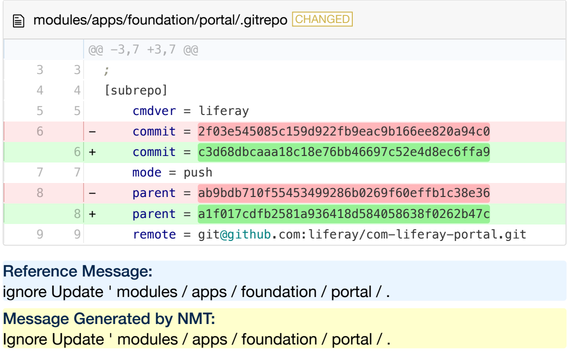
py/testdir_multi_jvm/test_parse_fs_schmoo.py CHANGED

```
@@ -1,7 +1,7 @@
  import os, json, unittest, time, shutil, sys
  sys.path.extend(['.','..','py'])
- import h2o, h2o_cmd
+ import h2o, h2o_cmd, h2o_hosts
  import h2o_browse as h2b
```

Reference Message:
add h2o_hosts

**Figure 3: A similar training commit to Figure 2**

By reading the commits of these *good messages*, we also observe that many (37%) of their *reference messages* are noisy. We identify two categories for such noisy messages. One category is named by us as *bot messages*, which refers to *reference messages* that are automatically generated by other development tools. The other category, which we call *trivial messages*, represents *reference messages* that are written by humans but contain little and redundant information that one can easily infer, for example, by just looking at the list of changed files.



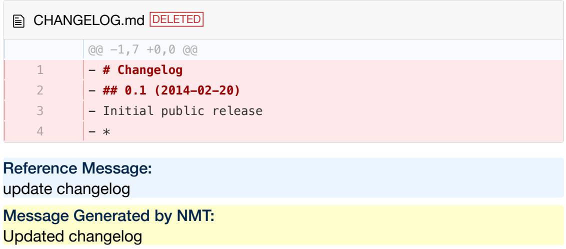
modules/apps/foundation/portal/.gitrepo CHANGED

```
@@ -3,7 +3,7 @@
  ;
  [subrepo]
      cmdver = liferay
-     commit = 2f03e545085c159d922fb9eac9b166ee820a94c0
+     commit = c3d68dbcaaa18c18e76bb46697c52e4d8ec6ffa9
      mode = push
-     parent = ab9bdb710f55453499286b0269f60effb1c38e36
+     parent = a1f017cdfb2581a936418d584058638f0262b47c
      remote = git@github.com:liferay/com-liferay-portal.git
```

Reference Message:
ignore Update ' modules / apps / foundation / portal / .

Message Generated by NMT:
Ignore Update ' modules / apps / foundation / portal / .

**Figure 4: An example of a *bot message***



CHANGELOG.md DELETED

```
@@ -1,7 +0,0 @@
- # Changelog
- ## 0.1 (2014-02-20)
- Initial public release
- *
```

Reference Message:
update changelog

Message Generated by NMT:
Updated changelog

**Figure 5: An example of a *trivial message***

Figure 4 shows an example of a *bot message*. The commit in Figure 4 is collected from the repository of *liferay-portal* [6]. We note that the *NMT message* is nearly the same as the *reference message*. However, after searching in GitHub, we find that this commit is pushed by a continuous integration (CI) bot named *liferay-continuous-integration*, which in turn automatically generates this *reference message*. Therefore, the *reference message* in Figure 4 is a *bot message*.

An example of a *trivial message* is presented in Figure 5. *NMT* also generates a nearly identical commit message to the *reference*

**Table 1: Our trivial message patterns**

| |
|---|
| update changelog/gitignore/readme [md/file] |
| prepare version [*version number*] |
| bump version [*version number*] |
| modify dockerfile/makefile |
| update submodule |

*"[]" means optional, "/" refers to "or" and *version number* refers to the version number introduced in a change.

**Table 2: Proportions of identified messages.**

| Dataset | *Bot Messages* | *trivial messages* | All |
|---|---|---|---|
| Original Training | 12.6% | 3.1% | 15.6% |
| Original Validation | 13.4% | 2.9% | 16.3% |
| Original Test | 12.8% | 3.2% | 16.0% |

*message*. However, both messages only mention that the changelog file was updated, and fail to describe what was changed in detail nor why this change occurred. Therefore, such a message contains little information. Since a programmer with rudimentary knowledge of version control systems is able to obtain the information by glancing the name of the changed file, the information is of little value. Moreover, this kind of messages can be automatically produced by some rule-based tools. For example, we can write a script to parse a new change. If the change only modified the changelog file of the project, our script will first extract the *filename* of the changed file, then simply output "update *filename*".

From these examples we can see that there is little useful information involved in these two categories of messages. Moreover, both *bot messages* and *trivial messages* can be generated through rule-based methods (e.g., *liferay-continuous-integration* or a simple script). Therefore, it makes little sense to learn from or produce these two kinds of messages through machine learning methods.

## 4.2 Evaluating *NMT* on the Cleaned Dataset

Based on the discovery of noisy messages, a question emerges in our mind: if we deleted such noisy messages and their corresponding `diffs` (i.e., the noisy commits) from Jiang et al.'s dataset and re-trained *NMT* on the new dataset, how much would the performance of *NMT* be affected?

In order to answer the above question, we first build a new dataset by removing the noisy commits from Jiang et al.'s dataset. To delete such noisy commits, we need to automatically identify *bot messages* and *trivial messages*. For *bot messages*, we only find the messages that are generated by *liferay-continuous-integration*. Since all such messages follow the same pattern, which is "ignore update *filename*", it is easy to identify them through a regular expression. However, there are more than one types of *trivial messages*. To identify them, we manually derive some common patterns of *trivial messages* by skimming the commits in Jiang et al.'s dataset. Table 1 presents our *trivial message* patterns. The exact regular expressions are available in our online appendix [3]. Table 2 shows the proportions of our identified *bot messages* and *trivial messages* in Jiang et al.'s dataset. We can see that *bot messages* and *trivial messages* are common in Jiang et al.'s dataset.

After identifying *bot messages* and *trivial messages*, their corresponding commits are regarded as noisy commits. We build the new training set, validation set and test set by deleting noisy commits from the training set, validation set and test set of Jiang et al.'s dataset, respectively. Please note we do not claim that we have found and deleted all noisy commits in Jiang et al.'s dataset. Only those commits of which the *reference messages* match our extracted patterns are cleaned by us.

**Table 3: BLEU-4 scores of original and new NMT models**

| Dataset | BLEU-4 | $p_1$ | $p_2$ | $p_3$ | $p_4$ |
|---|---|---|---|---|---|
| Original | 31.92 | 38.1 | 31.1 | 29.5 | 29.7 |
| Cleaned | 14.19 | 24.8 | 14.6 | 11.4 | 9.9 |

*"Original" refers to the original dataset provided by Jiang et al.'s. "Cleaned" refers to the cleaned dataset. The BLEU-4 scores are calculated on the whole test set. $p_n$ (n = 1,2,3,4) refers to the modified n-gram precision.

Then we re-train and test *NMT* on the cleaned dataset, the BLEU-4 score is computed to evaluate the new model, just like Jiang et al. The experimental results are shown in Table 3. Since the dataset, the implementation of *NMT* and the training and test scripts used by us are provided by Jiang et al., we simply use the results that are reported in their work [30] for performance comparison. We can see from Table 3 that the performance of the new model declines by a large amount. The BLEU-4 score of the new model is 55.5% lower than the original model. These results show that the good performance of *NMT* mainly comes from those noisy commits in Jiang et al.'s dataset.

In summary, after an in-depth analysis of *NMT messages*, we find that (1) The `diffs` of most (70 out of 71 in our randomly selected test set) *good messages* are similar to one or more training `diffs` at the token level. (2) About 16% of Jiang et al.'s commits are noisy commits. (3) The performance of *NMT* declines by a large amount after removing such noisy commits.

# 5 RQ2: CAN A SIMPLER AND FASTER APPROACH OUTPERFORM *NMT*?

*NMT* leverages the complex, slow and resource-consuming NMT algorithm to generate commit messages. Inspired by our first finding in RQ1, we propose a nearest-neighbor-based approach, named *NNGen*, which is simpler and faster than *NMT* while outperforming *NMT* in terms of BLEU-4 score on Jiang et al.'s dataset and the cleaned dataset.

## 5.1 *NNGen*

*NNGen* leverages the nearest neighbor (NN) algorithm to produce commit messages. The NN algorithm is a lazy learning method which is simple and does not require a training phase. Just like *NMT*, our approach takes as input a new `diff` and a training set, and outputs a one-sentence commit message for the new `diff`. Our approach first extracts `diffs` from the training set. Next, the training `diffs` and the new `diff` are represented as vectors in the form of "bags of words" [41]. In a bag-of-words model, the grammar and the word order of a `diff` are ignored, only term frequencies are kept. We refer to this kind of vector as a *diff vector*. Then, *NNGen* calculates the cosine similarity between the new *diff vector* and each training *diff vector*, and selects the top k training `diffs` with highest similarity scores. After that, the BLEU-4 score between the new `diff` and each of the top-k training `diffs` are computed. The training `diff` with the highest BLEU-4 score is regarded as the nearest neighbor of the new `diff`. Finally, our approach simply outputs the *reference message* of the nearest neighbor as the final result. In summary, given a new `diff`, our approach will first find

Table 4: BLEU-4 scores of *NMT* and *NNGen*

| Dataset | Approach | BLEU-4 | $p_1$ | $p_2$ | $p_3$ | $p_4$ |
|---|---|---|---|---|---|---|
| Original | *NMT* | 31.92 | 38.1 | 31.1 | 29.5 | 29.7 |
| | *NNGen* | **38.55** | **44.6** | **37.4** | **36.3** | **37.2** |
| Cleaned | *NMT* | 14.19 | 24.8 | 14.6 | 11.4 | 9.9 |
| | *NNGen* | **16.42** | **27.6** | **16.8** | **13.4** | **11.8** |

*The BLEU-4 scores are calculated on the test sets. $p_n$ (n = 1,2,3,4) refers to the modified n-gram precision.

Table 5: Time costs of *NMT* and *NNGen*

| Dataset | Approach | Device | Training Time | Testing Time |
|---|---|---|---|---|
| Original | *NMT* | GTX 1070 | 38 hours | 4.5 mins |
| | *NMT* | GTX 1080 | 34 hours | 17 mins |
| | *NNGen* | CPU | N/A | 30 secs |
| Cleaned | *NMT* | GTX 1080 | 24 hours | 13 mins |
| | *NNGen* | CPU | N/A | 23 secs |

*GTX 1070 and GTX 1080 refer to Nvidia GeForce GTX 1070 and 1080, respectively. CPU is Intel Core i5 2.5GHz.

its nearest neighbor in the training set, then reuse the *reference message* of the nearest neighbor as the generated message for the new `diff`.

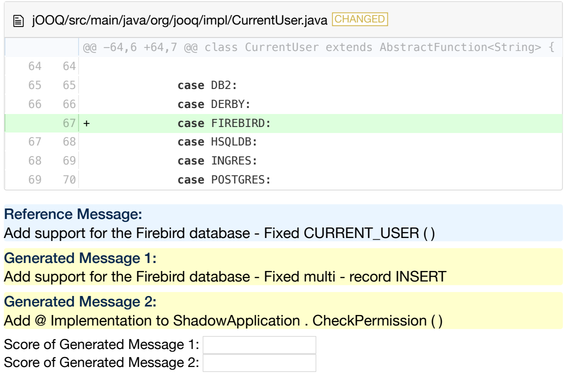
As described in Section 2.3, BLEU score [51] is a popular and automated metric for evaluating the quality of machine translation. It can also be used to measure the similarity between two sentences. Compared to cosine similarity, the BLEU score takes into account the order of words. However, the computation of BLEU score is relatively slow. To speed up our approach, we do not find the nearest neighbor by calculating BLEU scores for all the training `diffs`. Instead, we first use the cosine similarity between *diff vectors* to find the k nearest neighbor candidates. Then, the best candidate is selected according to BLEU scores. This strategy balances the time cost and accuracy of *NNGen*. By default, we set k as 5.

## 5.2 Automatic Evaluation

We evaluate the *NNGen* on Jiang et al.'s dataset and the cleaned dataset (described in 4.2) using the corpus-level BLEU-4 score, and compare the test results of *NNGen* with those of *NMT*. The test results of *NMT* on Jiang et al.'s dataset are reported in [30], and we directly use them for performance comparison. To evaluate *NMT* on the cleaned dataset, we use the implementation of *NMT* and the training and test scripts which are provided on Jiang et al.'s website [1], then train and test *NMT* using an Nvidia GeForce GTX 1080 with 8GB memory.

Table 4 presents the evaluation results for *NMT* and *NNGen*, we can see that our *NNGen* approach outperforms *NMT* in terms of BLEU-4 score on each dataset. The BLEU improvements achieved by our approach range from 16% to 21%. Moreover, All the modified n-gram precisions ($p_1$ - $p_4$ in Table 4) of our approach are higher than those of *NMT*.

To investigate whether *NNGen* is faster than *NMT*, we measure the time costs of *NNGen* on Jiang et al.'s dataset and the cleaned dataset, and compare them with the time costs of *NMT*. The time costs of *NMT* on Jiang et al.'s dataset is provided by Jiang et al. in [30]. Jiang et al. conducted the training and test of *NMT* on an Nvidia GeForce GTX 1070 GPU with 8GB memory. While evaluating *NMT*, we conduct the training and testing processes on an Nvidia GeForce GTX 1080 GPU with 8GB memory. However, *NNGen* is evaluated on a CPU (Intel Core i5 2.5GHz) with 8GB RAM. Table 5 shows the time costs of *NMT* and *NNGen*. For comparison, we also present the time costs of *NMT* on Jiang et al.'s dataset using our server. We can see that it takes 24 to 38 hours to train *NMT* and 4.5 to 17 minutes to test on the two datasets. Since *NNGen* does not need training, its training time is marked as "N/A". The time cost of its testing processes is only 23 to 33 seconds. This means *NNGen* is considerably (more than 2,600 times) faster than *NMT* on the two datasets.



jOOQ/src/main/java/org/jooq/impl/CurrentUser.java CHANGED

```
@@ -64,6 +64,7 @@ class CurrentUser extends AbstractFunction<String> {

                case DB2:
                case DERBY:
+               case FIREBIRD:
                case HSQLDB:
                case INGRES:
                case POSTGRES:
```

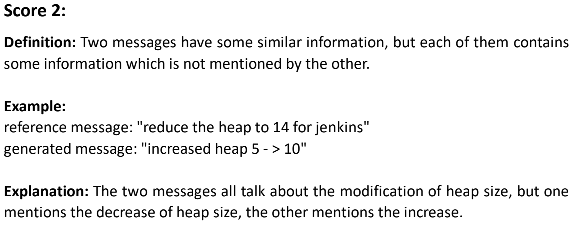
Reference Message:
Add support for the Firebird database - Fixed CURRENT_USER ( )

Generated Message 1:
Add support for the Firebird database - Fixed multi - record INSERT

Generated Message 2:
Add @ Implementation to ShadowApplication . CheckPermission ( )

Score of Generated Message 1:
Score of Generated Message 2:

Figure 6: A question in our survey



**Score 2:**

**Definition:** Two messages have some similar information, but each of them contains some information which is not mentioned by the other.

**Example:**
reference message: "reduce the heap to 14 for jenkins"
generated message: "increased heap 5 - > 10"

**Explanation:** The two messages all talk about the modification of heap size, but one mentions the decrease of heap size, the other mentions the increase.

Figure 7: A part of our scoring criterion

## 5.3 Human Evaluation

We also conduct a human evaluation to evaluate *NNGen* and compare *NNGen* with *NMT*. We invite 6 Ph.D. students to participate in our survey, all of whom are not co-authors, major in computer science and have industrial experience in Java programming (ranging from 1-4 years). Each participant is asked to read 100 commits and assess the semantic similarities between *reference messages* and each of commit messages generated by *NNGen* and *NMT*.

### 5.3.1 *Procedure.*

We randomly select 200 commits from the cleaned dataset (described in Section 4.2), divide them evenly into two groups and make a questionnaire for each group. In our questionnaires, each question first presents the information of one commit, i.e., its `diff`, its *reference message*, its *NMT message* and its *NNGen message*, then asks participants to give two scores between 0 to 4 to measure the semantic similarities between the *reference message* and the two generated messages. Score 0 means there is no similarity between the two messages, and score 4 means two messages are identical in meaning. Figure 6 shows one question in our survey. Participants are told that the first message is the *reference message*, but the order of the *NMT message* and the *NNGen message* is randomly decided. So, participants do not know which message is generated by which approach, and they are asked to enter to score each generated message separately.

**Table 6: The results of our user study**

| Approach | Low | Medium | High | Mean Score |
|---|---|---|---|---|
| *NMT* | 63.8% | 8.8% | 27.4% | 1.34 |
| *NNGen* | **57.9%** | **14.3%** | **27.8%** | **1.46** |

*"Low", "Medium" and "High" refer to low-quality, medium-quality and high-quality messages, respectively.

Each commit group is evaluated by 3 participants. Our scoring criterion is listed in the beginning of each questionnaire to guide participants. Figure 7 presents a part of our scoring criterion. Our complete scoring criterion can be found in our online appendix [3]. In addition, Participants are allowed to search the Internet for related information.

Different from Jiang et al.'s human study, first, we select commits from the cleaned dataset instead of Jiang et al.'s dataset, since it is meaningless to evaluate the commit messages generated for noisy commits. Second, the score range of our survey is 0-4 instead of 0-7. A large score range requires our participants to spend more efforts distinguishing subtle semantic differences, but in this work, we care more about the rough quality of generated messages instead of subtle semantic differences. In addition, a 5-point scale is widely used in prior software engineering studies[34, 35, 62, 63]. Third, we also provide `diffs` in our questionnaires to help participants make their judgments

*5.3.2 Results.* We obtain 600 pairs of scores from our human evaluation. Each pair contains a score for the *NMT message* and a score for the corresponding *NNGen message*. We regard a score of 0 and 1 as low quality, a score of 2 as medium quality and a score of 3 and 4 as high quality. Table 6 presents the results of our user study. We can see that the proportion of high-quality *NNGen messages* is slightly higher than that of high-quality *NMT messages*. 14.3% of the *NNGen messages* are of medium quality, while for *NMT messages*, the proportion is only 8.8%. The number of low-quality *NNGen messages* is smaller than that of low-quality *NMT messages*. Moreover, the mean score of *NNGen messages* is higher than that of *NMT messages*. These results show that *NNGen* outperforms *NMT*. We also use a Wilcoxon signed-rank test [60] at a 95% significance level to check whether the performance differences between *NNGen* and *NMT* are significant. The p-value is 0.01, which means the improvement achieved by *NNGen* is significant.

In summary, *NNGen* is much simpler and faster than *NMT*. Our experimental results show that *NNGen* outperforms *NMT* in terms of BLEU-4 score on Jiang et al.'s dataset and the cleaned dataset. In addition, our human evaluation shows that *NNGen* outperforms *NMT*, and the performance improvement is statistically significant.

## 6 DISCUSSION

### 6.1 Why Does *NNGen* Perform Better?

Given a new `diff`, *NNGen* first finds the `diff` which is most similar to it at the token level from the training set, then simply outputs the commit message of the training `diff` as the generated commit message. Hence, we speculate that the reason of *NNGen*'s better performance is that given a test commit, there is a high chance that there will always exist a very similar training commit to it.

To verify our conjecture, we conduct an experiment on Jiang et al.'s dataset. We first convert all `diffs` in the training set and the test set into *diff vectors* (described in Section 5.1). Then, we compute the cosine similarities between each test *diff vector* and each training *diff vector*. Finally, for each test `diff`, we select the training `diff` with the highest cosine similarity score, and record this score. There are 3,000 commits in Jiang et al.'s test set, hence we obtain a set of 3,000 cosine similarity scores.

To reduce the potential variance caused by how Jiang et al.'s dataset is divided, we also perform a 10-fold cross-validation. Specifically, we first shuffle the 32,208 commits in Jiang et al.'s dataset and divide these commits into 10 folds. Then we run the same procedure as the above experiment 10 times. Each time one fold of 3,220 (3,228 for the last run) commits is used as the test set, and the remaining folds of 28,988 (28,980 for the last run) commits is used as the training set. We obtain a set of 32,208 cosine similarity scores from the 10-fold cross-validation.

We visualize the distribution of each set of cosine similarity scores using a violin plot [26], as shown in Figure 8. The left plot is the distribution of the scores of Jiang et al.'s test set, and the right one is the distribution of the scores obtained from the 10-fold cross-validation. The visualization results show that the distribution of

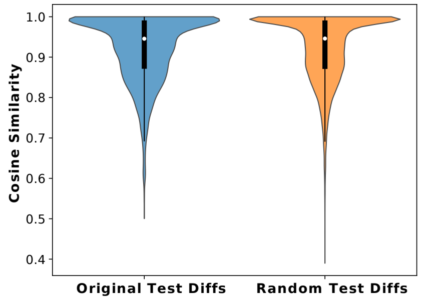
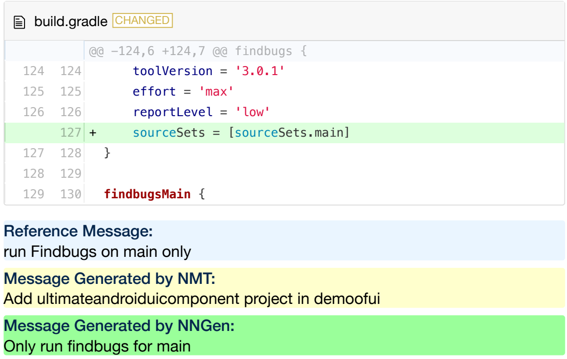


**Figure 8: Distribution of the cosine similarities between test `diffs` and their nearest neighbors**


build.gradle CHANGED

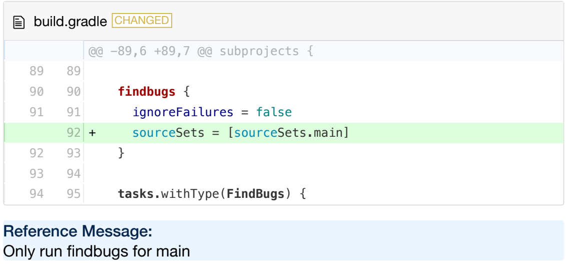
```
@@ -124,6 +124,7 @@ findbugs {
        toolVersion = '3.0.1'
        effort = 'max'
        reportLevel = 'low'
+       sourceSets = [sourceSets.main]
    }

    findbugsMain {
```

Reference Message:
run Findbugs on main only

Message Generated by NMT:
Add ultimateandroiduicomponent project in demoofui

Message Generated by NNGen:
Only run findbugs for main

**Figure 9: A test commit**


build.gradle CHANGED

```
@@ -89,6 +89,7 @@ subprojects {

    findbugs {
        ignoreFailures = false
+       sourceSets = [sourceSets.main]
    }

    tasks.withType(FindBugs) {
```

Reference Message:
Only run findbugs for main

**Figure 10: The nearest neighbor found by *NNGen***

each set of scores is highly skewed. Specifically, for each set, the cosine similarities between most test `diffs` and their most similar training `diffs` are higher than 0.7. Hence, the majority of test commits are very similar to another training commit at the token level.

We also manual examine some examples from Jiang et al.'s dataset to explain the better performance of *NNGen*. Figure 9 and 10 show one of such examples. Figure 9 presents a test commit, its *NMT message* and its *NNGen message*. Its nearest neighbor found by NNGen is shown in Figure 10. We can see that the `diffs` of the two commits are similar at the token level, and their *reference messages* are identical in meaning. By finding out this nearest neighbor, *NNGen* produces a high-quality commit message for this test commit. However, the commit message generated by *NMT* is not relevant to this test commit at all, which means *NMT* fails to generate a good commit message.

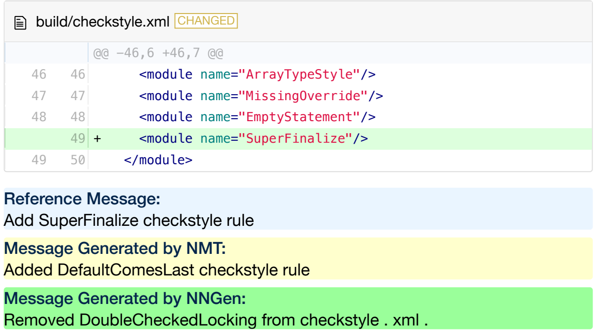
In summary, we argue that given a test commit, there is a high chance that there will exist a training commit which is very similar to it, and for those test commits that are very similar to another training commit, *NNGen* can generate better commit messages than *NMT*.

## 6.2 Where Does *NMT* Perform Better?

Although the overall performance of *NNGen* is better than that of *NMT* on the two datasets, in some cases, *NMT messages* obtain higher scores than their corresponding *NNGen messages*. To figure out such cases, we compare the average scores of each *NMT message* and its corresponding *NNGen message*, and find there are 30 commits where *NMT* performs better than *NNGen*.

We manually analyze these 30 commits and their generated commit messages. We find that for 20 out of the 30 commits, the *NMT messages* and *NNGen messages* are all of low quality (i.e., Score 0 or 1). In each case, the meanings of the *reference message* and the two generated messages are different, but *NMT* generates the right verb at the beginning of the *NMT message*. Therefore, the average scores of the 20 *NMT messages* are all 1, and those of the corresponding *NNGen messages* are all 0. For example, the *reference message* of a test commit is "Add AbstractProcessingFilter.getAuthenticationDetailsSource()", its *NMT message* is "Added getter for authoritiesPopulator", and its *NNGen message* is "Properly handle empty layout in getFirstVisiblePosition()". We can see that the meanings of the three messages are different, but *NMT*generates the correct verb "add" at the beginning. Thus, this *NMT message* is better than this *NNGen message*.

As for the other 10 commits, we find that without considering the case, the *NMT messages* of 9 commits are identical to one or more training commit messages. In these cases, the reason of *NMT*'s better performance may be that *NMT* captures better nearest neighbors than *NNGen*. Figure 11 shows an example. After comparing the training `diffs` of the nearest neighbors captured by *NMT* and *NNGen* with this test `diff`, we find that although the training `diff` selected by *NNGen* is lexically more similar to this test `diff`, the meaning and the writing style of this *NMT message* is closer to this *reference message*. Thus, this *NMT message* obtains higher average score.



build/checkstyle.xml CHANGED

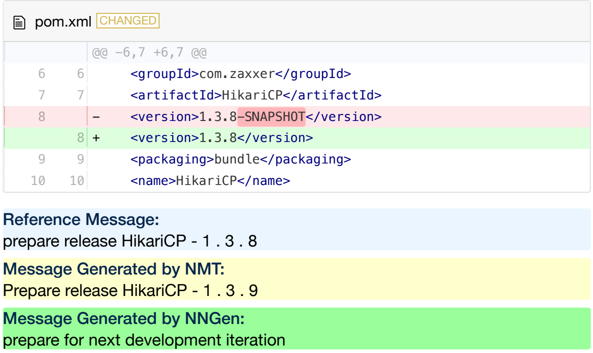
```
@@ -46,6 +46,7 @@
        <module name="ArrayTypeStyle"/>
        <module name="MissingOverride"/>
        <module name="EmptyStatement"/>
+       <module name="SuperFinalize"/>
    </module>
```

Reference Message:
Add SuperFinalize checkstyle rule

Message Generated by NMT:
Added DefaultComesLast checkstyle rule

Message Generated by NNGen:
Removed DoubleCheckedLocking from checkstyle . xml .

**Figure 11: An example where *NMT* performs better**



pom.xml CHANGED

```
@@ -6,7 +6,7 @@
    <groupId>com.zaxxer</groupId>
    <artifactId>HikariCP</artifactId>
-   <version>1.3.8-SNAPSHOT</version>
+   <version>1.3.8</version>
    <packaging>bundle</packaging>
    <name>HikariCP</name>
```

Reference Message:
prepare release HikariCP - 1 . 3 . 8

Message Generated by NMT:
Prepare release HikariCP - 1 . 3 . 9

Message Generated by NNGen:
prepare for next development iteration

**Figure 12: Another example where *NMT* performs better**

There is also a special case where the *NMT message* can not be found in the training set, as shown in Figure 12. The three messages are similar in the meaning, while compared to this *NNGen message*, this *NMT message* is more specific and accurate. Therefore, *NMT* performs better in this case. After further searching in the training set, we find that there exist some training commit messages which share the same pattern as this *NMT message*. For example, a training commit message is "prepare release HikariCP–1.3.2". This means *NMT* has the ability to generalize, but its generalization ability is very limited so that its overall performance is worse than our simple approach, i.e., *NNGen*.

Due to the space constraint, more details of the aforementioned examples can be found in our online appendix [3].

## 6.3 Implications

From this work, we distill some general suggestions which is beyond the specific task and approaches.

**Clean up the data carefully.** In software repositories, there may exist some noisy data, e.g., the noisy commits in Jiang et al.'s dataset. It makes little sense to train and test our models on the noisy data. Worse still, we may get misleading results if we conduct experiments on dataset with noisy examples [21]. Therefore, we recommend researchers to always clean up their datasets carefully before training and testing their models.

**Consider simple approaches first.** Our study shows that it is worth trying simple and fast methods before applying complicated and time-consuming techniques on software engineering tasks. This "try-with-simpler" practice is also recommended by Fu and Menzies [19]. Implementing and applying simple methods only costs a little effort, but may bring a deep understanding of the data. Moreover, for some SE tasks, simple approaches are able to achieve equal or even better performance in less time, e.g., *NNGen* vs *NMT*.

## 6.4 Threats to Validity

One of the threats to validity is about the manual inspection in Section 4. We ask two raters to evaluate the quality of 200 randomly selected *NMT messages* according to Jiang et al.’s criterion. We cannot guarantee that our judgments are fully in line with the results of Jiang et al.’s human study. However, our scores are only leveraged to identify high quality commit messages generated by *NMT*, and are not used for performance comparison. Additionally, the proportion of high quality commit messages found by us is close to (a little higher than) that observed in Jiang et al.’s human evaluation on a different test set, which makes us more confident about our judgments.

The second threat to validity is about the evaluation of *NMT* on the cleaned dataset. To build the cleaned dataset, we remove about 16% of commits from Jiang et al.’s dataset. The reduction of the dataset may be one of the reasons of the decrease of *NMT*’s BLEU score on the cleaned dataset.

Another threat to validity is that we only compare *NNGen* and *NMT* on Jiang et al.’s dataset and the cleaned dataset. On the two datasets, *NNGen* outperforms *NMT* by a substantial margin. But we do not claim that this finding can be generalized to all datasets. We only stress that compared to *NMT*, the simplicity, the speed and the effectiveness of *NNGen* make it a competitive baseline on `diff`-natural language translation task.

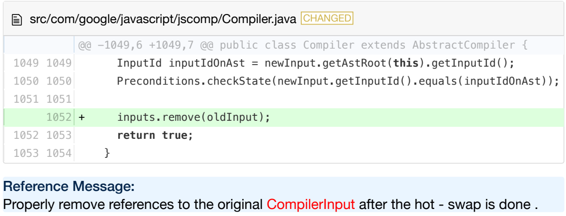
# 7 ROAD AHEAD FOR COMMIT MESSAGE GENERATION

Based on our above mentioned findings, we now present some observations about the road ahead for commit message generation.

**Only `diffs` and historical commit messages are not enough for commit message generation.** Both *NMT* and *NNGen* only take `diffs` and historical commit messages as input, so they cannot generate tokens that are not contained in the training set and the test `diffs`, but only appear in the test *reference messages*. We call these tokens *unique tokens*, and refer to the test commits of which the *reference messages* contain at least one *unique token* as *unique commits*. To figure out the amount of *unique tokens* in Jiang et al.’s dataset, we first tokenize all the `diffs` and *reference messages*, convert all tokens into lowercase, and remove the tokens which are numbers. Then, we build two vocabularies, one of which is constructed from the tokens in the training set and the test `diffs`, and the other vocabulary is formed from only the tokens in the test *reference messages*. Finally, we compare the two vocabularies. We find that there are 296 *unique tokens* in Jiang et al.’s dataset, and 6% (180 out of 3,000) of the test commits are *unique commits*. Figure 13 and 14 show two *unique commits* in Jiang et al.’s dataset.

Figure 13 presents a commit in the *closure-compiler* project [5]. The token “CompilerInput” is a *unique token*. After searching in GitHub and reading more lines before the `diff`, we find that “CompilerInput” is the type of the variable “oldInput” (also the element type of the list “inputs”). The token “CompilerInput” provides the type information which is not contained in the `diff`, yet it plays an important role in describing this commit.

Figure 14 is extracted from the repository of *ExoPlayer* [4]. The token “TsExtractor” is a *unique token*. By reading the documentation of *ExoPlayer*, we find that TsExtractor is a class of *Exoplayer*, and is used to extract data from the MPEG-2 TS container format [8]. This information is project-specific knowledge. Without the token “TsExtractor”, the *reference message* would fail to reveal the specific rationale behind this commit. Therefore, this token is essential.

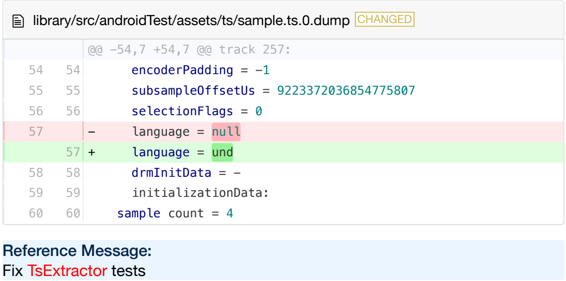


src/com/google/javascript/jscomp/Compiler.java CHANGED

```
@@ -1049,6 +1049,7 @@ public class Compiler extends AbstractCompiler {
        InputId inputIdOnAst = newInput.getAstRoot(this).getInputId();
        Preconditions.checkState(newInput.getInputId().equals(inputIdOnAst));

+       inputs.remove(oldInput);
        return true;
      }
```

Reference Message:
Properly remove references to the original CompilerInput after the hot - swap is done .

Figure 13: An example of a *unique commit*.



library/src/androidTest/assets/ts/sample.ts.0.dump CHANGED

```
@@ -54,7 +54,7 @@ track 257:
        encoderPadding = -1
        subsampleOffsetUs = 9223372036854775807
        selectionFlags = 0
-       language = null
+       language = und
        drmInitData = -
        initializationData:
      sample count = 4
```

Reference Message:
Fix TsExtractor tests

Figure 14: Another example of a *unique commit*.

These two examples highlight that there exists some information which cannot be found in the `diffs` and prior historical commit messages, yet such information is essential for high-quality commit messages. Hence, only `diffs` and historical commit messages are not enough for commit messages generation. This task requires synthesis of information from different data sources.

**Take more types of information into consideration.** According to the above finding, we recommend the inclusion of other types of information when designing new approaches for commit message generation. We have showed two types of information which is useful for commit message generation, i.e., context information of `diffs` (Figure 13) and project-specific knowledge (Figure 14). To capture the former one, we can simply extend `diffs` to contain more lines of code, or leverage program analysis (for code changes) and semantic analysis techniques to extract data dependencies and type information from context of `diffs` (e.g., the lines before and after `diffs`, the code which `diffs` are dependent on, etc.) and add extracted information into our dataset. To capture project-specific knowledge, other types of documents in software repositories can be used while training and generating. For example, if a new commit aims to fix a bug, the corresponding bug report may provide important context information of the bug. In addition, We can also build a knowledge base (e.g., Knowledge Graph) for each project and incorporate these knowledge bases with machine learning methods.

**We still have a long way to go.** We showed that if an approach only takes `diffs` and prior historical commit messages as input, it cannot generate perfect commit messages for *unique commits*. Assume that there was an approach, which performed perfectly on the test commits that are not *unique commits*, and performed like *NMT* on the *unique commits*. Specifically, for the test commits without *unique tokens*, this perfect approach would generate the same messages as their *reference messages*. For *unique commits*, this perfect approach would generate messages that are identical to their *NMT messages*. The BLEU score of this perfect approach on Jiang et al.'s dataset would be 92.94. But now, the BLEU scores of *NMT* and *NNGen* are only 31.92 and 38.55, respectively. The performance differences between this perfect approach and *NMT* and *NNGen* are considerable. Moreover, after deleting the noisy commits in Jiang et al.'s dataset, the performance of both *NMT* and *NNGen* declines, which shows that the two approaches are still not powerful enough. In addition, it is worth mentioning that Jiang et al.'s dataset represents only 1.75% out of 2 million commits that are collected from GitHub. The performance of *NMT* and *NNGen* on big-scale datasets is still an open question. Therefore, there is still a long way to go to automatically generate commit messages.

# 8 RELATED WORK

## 8.1 Commit Message Generation

A number of techniques have been proposed to automatically generate commit messages. Some of them take source code changes as input. For instance, DeltaDoc [13] obtains path predicates by symbolically executing source code changes, then generates commit messages using a set of predefined rules and transformations. ChangeScribe [16, 38] first extracts the stereotype, the type and the impact set of a commit by analyzing corresponding source code changes and the abstract syntax trees. Then it fills predefined templates with the extracted information to document this commit. The approach proposed by Shen et al. is similar to ChangeScribe, but it constrains the length of the generated message by removing repeated information in the change [55]. Le et al. proposed a framework to infer the semantic differences between two versions of a code base through dynamic analysis [36]. Commit messages that are generated by these tools are usually verbose, and cannot describe the intent of commits concisely.

Several approaches make use of various documents in software repositories to produce commit messages. For example, to answer *why* a change happened, Rastkar and Murphy proposed an approach to extract motivational information of commits from multiple relevant documents [53]. Moreno et al. have built ARENA [46, 47], a tool which combines multiple kinds of changes, i.e., changes to source code, libraries, documentation and licenses, with issues from software repositories to generate release notes. Hassan and Holt proposed an approach, named *Source Sticky Notes*, to better explain the static dependencies of a software system using historical modification records [24].

In addition, Huang et al. proposed an approach to produce commit messages for code changes by reusing the commit messages of similar existing commits [28]. The similarity between two commits is measured by syntactic similarity and semantic similarity between their changed code fragments. Their approach only focuses on code changes. However, we aims to generate commit messages from `diffs`, which includes both code changes and non-code changes. We cannot analyze the code syntax of non-code changes. Moreover, we find that in Jiang et al.'s dataset, over 70% of the changes are non-code changes. Hence, Huang et al.'s approach only solves part of this problem.

## 8.2 Source Code Summarization

Source code summarization techniques may also be used to generate commit messages. Most of the techniques adapt a two phase framework to summarize source code. Specifically, they first select important content from source code, then transform the selected content into natural language descriptions through predefined rules. For example, to summarize Java method, the framework proposed by Sridhara et al. first identifies significant statements of a Java method according to structural and linguistic clues, then expresses extracted content in natural language using predefined text templates [56]. This framework has been extended to summarize high-level actions within methods [57], crosscutting source code concern [54], Java classes [45, 48], C++ methods [9], context of Java methods [42, 43] and object-related statement sequences [59]. Some studies also leverage this framework to generate explanation or summarization for special types of code, e.g., exceptions [12], failed tests [65], parameters of Java methods [58], unit test cases [33, 37] and database usages and constrains [39].

Some work leverages information retrieval techniques to summarize source code [10, 22, 61, 62]. For example, Wong et al. proposed a method which mines code-description mappings from StackOverflow [7] and automatically generates code comments by reusing the descriptions of similar code snippets in the extracted database [62].

In addition, machine translation techniques are also applied to produce source code summaries [29, 52]. CODE-NN [29], proposed by Iyer et al., leverages an NMT algorithm to generate descriptions for C# and SQL code. Phan et al. adapted phrase-based statistical machine translation to translate between behavioral exception documentation and source code of API methods [52].

# 9 CONCLUSION

Automatically generating high-quality commit messages is a challenging task. Recently, Jiang et al. proposed leveraging an NMT algorithm to generate one-sentence commit messages from `diffs`. Their approach (*NMT*) learns from historical data, summarizes commits using one-sentence messages and shows promising results. However, they do not investigate why *NMT* performs so well, and *NMT* is quite complicated and time-consuming.

In this paper, we first analyze Jiang et al.'s experimental results. We find that there are noisy commit messages in their dataset, and that the good performance of *NMT* benefits from those noisy commit messages. Then, inspired by our findings, we propose a simple, nearest-neighbor-based approach, named *NNGen*, to generate short commit messages from `diffs`. Our experimental results show that *NNGen* is much faster and performs better than *NMT* on Jiang et al.'s dataset and the cleaned dataset. Finally, we conduct a further analysis of commit message generation, and discuss some challenges in the road ahead for this task to inspire other researchers.

# ACKNOWLEDGEMENT

Xin Xia and Xinyu Wang are the corresponding authors. This research was partially supported by the National Key Research and Development Program of China (2018YFB1003904) and NSFC Program (No. 61602403).

## REFERENCES

[1] 2017. Jiang et al.'s website. https://sjiang1.github.io/commitgen/.

[2] 2018. Git. https://git-scm.com/.

[3] 2018. Our online appendix. https://goo.gl/63B976.

[4] 2018. ExoPlayer. https://github.com/google/ExoPlayer.

[5] 2018. Google Closure Compiler. https://github.com/google/closure-compiler.

[6] 2018. Liferay Portal. https://github.com/liferay/liferay-portal.

[7] 2018. Stack Overflow. https://stackoverflow.com/.

[8] 2018. TsExtractor in ExoPlayer. https://goo.gl/Dsbdjf.

[9] Nahla J Abid, Natalia Dragan, Michael L Collard, and Jonathan I Maletic. 2015. Using stereotypes in the automatic generation of natural language summaries for c++ methods. In *Software Maintenance and Evolution (ICSME), 2015 IEEE International Conference on*. IEEE, 561–565.

[10] Miltiadis Allamanis, Earl T Barr, Christian Bird, and Charles Sutton. 2015. Suggesting accurate method and class names. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM, 38–49.

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

[12] Raymond PL Buse and Westley R Weimer. 2008. Automatic documentation inference for exceptions. In *Proceedings of the 2008 international symposium on Software testing and analysis*. ACM, 273–282.

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

[14] 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).

[15] Jacob Cohen. 1968. Weighted kappa: Nominal scale agreement provision for scaled disagreement or partial credit. *Psychological bulletin* 70, 4 (1968), 213.

[16] 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 *IEEE International Working Conference on Source Code Analysis and Manipulation*. 275–284.

[17] Sergio Cozzetti B de Souza, Nicolas Anquetil, and Kathia M de Oliveira. 2005. A study of the documentation essential to software maintenance. In *Proceedings of the 23rd annual international conference on Design of communication: documenting & designing for pervasive information*. ACM, 68–75.

[18] 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 *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press, 422–431.

[19] Wei Fu and Tim Menzies. 2017. Easy over hard: a case study on deep learning. In *Joint Meeting on Foundations of Software Engineering*. 49–60.

[20] Ying Fu, Meng Yan, Xiaohong Zhang, Ling Xu, Dan Yang, and Jeffrey D Kymer. 2015. Automated classification of software change messages by semi-supervised Latent Dirichlet Allocation. *Information and Software Technology* 57 (2015), 369–377.

[21] Baljinder Ghotra, Shane McIntosh, and Ahmed E Hassan. 2015. Revisiting the impact of classification techniques on the performance of defect prediction models. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*. IEEE Press, 789–800.

[22] Sonia Haiduc, Jairo Aponte, Laura Moreno, and Andrian Marcus. 2010. On the use of automated text summarization techniques for summarizing source code. In *Reverse Engineering (WCRE), 2010 17th Working Conference on*. IEEE, 35–44.

[23] Ahmed E Hassan. 2008. Automated classification of change messages in open source projects. In *Proceedings of the 2008 ACM symposium on Applied computing*. ACM, 837–841.

[24] Ahmed E Hassan and Richard C Holt. 2004. Using development history sticky notes to understand software architecture. In *Program Comprehension, 2004. Proceedings. 12th IEEE International Workshop on*. IEEE, 183–192.

[25] Abram Hindle, Earl T Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. 2012. On the naturalness of software. In *Software Engineering (ICSE), 2012 34th International Conference on*. IEEE, 837–847.

[26] Jerry L Hintze and Ray D Nelson. 1998. Violin plots: a box plot-density trace synergism. *The American Statistician* 52, 2 (1998), 181–184.

[27] Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. 2018. Deep code comment generation. In *Proceedings of the 26th Conference on Program Comprehension, ICPC 2018, Gothenburg, Sweden, May 27-28, 2018*. 200–210. https://doi.org/10.1145/3196321.3196334

[28] Yuan Huang, Qiaoyang Zheng, Xiangping Chen, Yingfei Xiong, Zhiyong Liu, and Xiaonan Luo. 2017. Mining Version Control System for Automatically Generating Commit Comment. In *Empirical Software Engineering and Measurement (ESEM), 2017 ACM/IEEE International Symposium on*. IEEE, 414–423.

[29] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2016. Summarizing source code using a neural attention model. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, Vol. 1. 2073–2083.

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

[31] Siyuan Jiang and Collin McMillan. 2017. Towards automatic generation of short summaries of commits. In *Proceedings of the 25th International Conference on Program Comprehension*. IEEE Press, 320–323.

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

[33] Manabu Kamimura and Gail C Murphy. 2013. Towards generating human-oriented summaries of unit test cases. In *Program Comprehension (ICPC), 2013 IEEE 21st International Conference on*. IEEE, 215–218.

[34] Pavneet Singh Kochhar, Xin Xia, David Lo, and Shanping Li. 2016. Practitioners' expectations on automated fault localization. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*. ACM, 165–176.

[35] Jan-Peter Krämer, Joel Brandt, and Jan Borchers. 2016. Using runtime traces to improve documentation and unit test authoring for dynamic languages. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*. ACM, 3232–3237.

[36] Tien-Duy B Le, Jooyong Yi, David Lo, Ferdian Thung, and Abhik Roychoudhury. 2014. Dynamic inference of change contracts. In *Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on*. IEEE, 451–455.

[37] Boyang Li, Christopher Vendome, Mario Linares-Vásquez, Denys Poshyvanyk, and Nicholas A Kraft. 2016. Automatically documenting unit test cases. In *Software Testing, Verification and Validation (ICST), 2016 IEEE International Conference on*. IEEE, 341–352.

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

[39] Mario Linares-Vásquez, Boyang Li, Christopher Vendome, and Denys Poshyvanyk. 2016. Documenting database usages and schema constraints in database-centric applications. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*. ACM, 270–281.

[40] Christopher Manning, Mihai Surdeanu, John Bauer, Jenny 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.

[41] Christopher D Manning, Prabhakar Raghavan, Hinrich Schütze, et al. 2008. *Introduction to information retrieval*. Vol. 1. Cambridge university press Cambridge.

[42] Paul W McBurney and Collin McMillan. 2014. Automatic documentation generation via source code summarization of method context. In *Proceedings of the 22nd International Conference on Program Comprehension*. ACM, 279–290.

[43] Paul W McBurney and Collin McMillan. 2016. Automatic source code summarization of context for java methods. *IEEE Transactions on Software Engineering* 42, 2 (2016), 103–119.

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

[45] Laura Moreno, Jairo Aponte, Giriprasad Sridhara, Andrian Marcus, Lori Pollock, and K Vijay-Shanker. 2013. Automatic generation of natural language summaries for java classes. In *Program Comprehension (ICPC), 2013 IEEE 21st International Conference on*. IEEE, 23–32.

[46] Laura Moreno, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Andrian Marcus, and Gerardo Canfora. 2014. Automatic generation of release notes. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 484–495.

[47] Laura Moreno, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Andrian Marcus, and Gerardo Canfora. 2017. ARENA: an approach for the automated generation of release notes. *IEEE Transactions on Software Engineering* 43, 2 (2017), 106–127.

[48] Laura Moreno, Andrian Marcus, Lori Pollock, and K Vijay-Shanker. 2013. Jsummarizer: An automatic generator of natural language summaries for java classes. In *Program Comprehension (ICPC), 2013 IEEE 21st International Conference on*. IEEE, 230–232.

[49] David G Novick and Karen Ward. 2006. What users say they want in documentation. In *Proceedings of the 24th annual ACM international conference on Design of communication*. ACM, 84–91.

[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 *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*. 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] Hung Phan, Hoan Anh Nguyen, Tien N Nguyen, and Hridesh Rajan. 2017. Statistical learning for inference between implementations and documentation. In *Software Engineering: New Ideas and Emerging Technologies Results Track (ICSE-NIER), 2017 IEEE/ACM 39th International Conference on*. IEEE, 27–30.

[53] Sarah Rastkar and Gail C Murphy. 2013. Why did this code change?. In *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press, 1193–1196.

[54] Sarah Rastkar, Gail C Murphy, and Alexander WJ Bradley. 2011. Generating natural language summaries for crosscutting source code concerns. In *Software Maintenance (ICSM), 2011 27th IEEE International Conference on*. IEEE, 103–112.

[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 *Computer Software and Applications Conference (COMPSAC), 2016 IEEE 40th Annual*, Vol. 1. IEEE, 103–112.

[56] Giriprasad Sridhara, Emily Hill, Divya Muppaneni, Lori Pollock, and K Vijay-Shanker. 2010. Towards automatically generating summary comments for java methods. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*. ACM, 43–52.

[57] Giriprasad Sridhara, Lori Pollock, and K Vijay-Shanker. 2011. Automatically detecting and describing high level actions within methods. In *Proceedings of the 33rd International Conference on Software Engineering*. ACM, 101–110.

[58] Giriprasad Sridhara, Lori Pollock, and K Vijay-Shanker. 2011. Generating parameter comments and integrating with method summaries. In *Program Comprehension (ICPC), 2011 IEEE 19th International Conference on*. IEEE, 71–80.

[59] Xiaoran Wang, Lori Pollock, and K Vijay-Shanker. 2017. Automatically generating natural language descriptions for object-related statement sequences. In *Software Analysis, Evolution and Reengineering (SANER), 2017 IEEE 24th International Conference on*. IEEE, 205–216.

[60] Frank Wilcoxon. 1945. Individual comparisons by ranking methods. *Biometrics bulletin* 1, 6 (1945), 80–83.

[61] Edmund Wong, Taiyue Liu, and Lin Tan. 2015. Clocom: Mining existing source code for automatic comment generation. In *Software Analysis, Evolution and Reengineering (SANER), 2015 IEEE 22nd International Conference on*. IEEE, 380–389.

[62] Edmund Wong, Jinqiu Yang, and Lin Tan. 2013. Autocomment: Mining question and answer sites for automatic comment generation. In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*. IEEE, 562–567.

[63] Xin Xia, Lingfeng Bao, David Lo, Pavneet Singh Kochhar, Ahmed E Hassan, and Zhenchang Xing. 2017. What do developers search for on the web? *Empirical Software Engineering* 22, 6 (2017), 3149–3185.

[64] Meng Yan, Ying Fu, Xiaohong Zhang, Dan Yang, Ling Xu, and Jeffrey D Kymer. 2016. Automatically classifying software changes via discriminative topic model: Supporting multi-category and cross-project. *Journal of Systems and Software* 113 (2016), 296–308.

[65] Sai Zhang, Cheng Zhang, and Michael D Ernst. 2011. Automated documentation inference to explain failed tests. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*. IEEE Computer Society, 63–72.