Project Proposal: `odit`, Open Dialogue for Git

KEVIN EWING and ZAYN MAKDESSI

ACM Reference Format:
Kevin Ewing and Zayn Makdessi. 2022. Project Proposal: `odit`, Open Dialogue for Git. 1, 1 (October 2022), 3 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 MOTIVATION

Collaboration is a huge aspect of software development, and therefore communication among developers is crucial to ensuring effective progress. One way developers allow others to understand their work, is through version control commit messages. Version control, the most popularly choice being git, allows developers to leave a trail of incremental changes one has made to the versioned repository, allowing coworkers to understand how a project has changed over time. Functionally, commit messages are manually entered by individual developers and oftentimes are not given enough thought. In practice, these commit messages often fail to fully explain the extent and implications of the code change. In other words, commit messages often fail to explain both *what* has changed in the repository, and *why*.

As outlined by Tian et al. [11], the shortcomings of commit messages often fall into five distinct groups. The first are single-word messages, which include “merge” or “polish." Submit-centered messages are those that only answer the *what*, for example, “loader changes.” Furthermore, scope-centered messages only convey the size of the changes without describing what has changed, “minor changes in test.” Next, are redundant messages including those like “add” or “delete.” Finally the commit message, “Kevin and Zayn Monday afternoon pair programming” or an empty string, would fall into the irrelevant message category as it does not say anything meaningful about what has changed.

Deficient commit messages inhibit productive collaboration. Therefore, efforts, using a variety of approaches, have been proposed and implemented to automatically generate these commit messages. To better understand our project foundation we will first discuss preexisting tools and models for interpreting programming language (PL) to natural language (NL) in Section 2. Before presenting the outline of our implementation in Section 3.

2 EXISTING TOOLS AND MODELS

Many tools and models have been proposed in recent years to solve this problem of converting programming language (PL) to natural language (NL). In general, however, methods fall into three main groups [12]. The first is a rule-based approach. Commits generated by this model are often not very abstracted from the commit differential as the predefined rules and templates classifying changes are limiting. Furthermore, these approaches can not truly cover all cases. Example tools of this technique include ChangeScribe [7], an Eclipse graphical user interface (GUI) plugin restricted to Java code change commits, and DeltaDocs [2]. The second major category of PL to NL translation is the retrieval-based approach. This strategy first categorizes new repository commit changes with past ones. It then recycles these past

Authors’ address: Kevin Ewing; Zayn Makdessi.

Manuscript submitted to ACM

commit messages and updates them to better fit the current change. This method achieves good results in accuracy and efficiency however it is largely limited by the existence of similar code changes and the quality of those corresponding past commit messages [10]. Implementations include NNGen proposed by Liu et al. [9]. Finally, are the learning-based models. Learning-based approaches correlate the semantic commit differential to source code and commit messages of large data sets. The primary competing models consist of Neural Machine Translation (NMT) [9], ATOM AST (Abstract Syntax Tree) [8], CoRec [12] and CommitBERT [6] (based on CodeBERT [4]).

All three of these methods come with thier own set of drawbacks and advantages. Rule-based approaches often struggle with rigidity and over-reporting, meaning they can not account for all possible code changes, and when many code changes occur the models struggle to prioritize changes. On the other hand, these models often are the quickest and do provide a more concise summarization than the raw commit differential. As previously mentioned, retrieval-based approaches are limited by the existence of similar code changes and the quality of those corresponding past commit messages. Finally, learning-based approaches are the most resource intensive and are limited by the quality of their training commit messages. Within the learning-based approach, there is much ongoing research on the effectiveness of varying methods. One cutting-edge approach examines the code change diffs as changes in the abstract syntax trees (AST) of programs instead of the plain text code. ATOM [8], an implementation of AST, outperforms both NNGen NMT models in terms of BLEU_4 score (a metric for evaluating a generated sentence to the meaning of a reference sentence) [3].

However, while these tools and models are becoming evermore proficient at abstracting *what* has changed, no model is capable of replacing the developer in describing *why* these changes have been made. This is therefore why we propose our Open Dialogue for Git, `odit`.

3 OUR SOLUTION

We propose a python wrapper for git to improve both developer experience and encourage helpful commit messages. On top of hoping to achieve a much more organic feeling dialogue with git, we also want to integrate CommitBERT [6] to auto-generate the *what has changed* aspect of commit messages. We then plan to prompt users with *why these changes have been made*. By integrating the *what* and *why* we will achieve more detailed commit messages that are up to collaborative standards [5]. This is our attempt to mitigate the shortcomings of natural language generation models at parsing the "*why changes have been made*" aspect of commit messages while still relieving some of the burdens on developers. The *why* question will therefore be an optional response that should be left blank when only trivial changes have been made where the reasoning is self-explanatory.

Another potential use for our open dialogue for git, is for learning developers. Developers with little experience in git might use our tool as a teaching aid. We hope to embrace this potential use and encourage learning of git by returning the complete git command, or series of commands that could have been run instead of interacting with the wrapper. We hope this would help build developers' experience with git as well as allow reproducibility of effects without having to use the wrapper in scripting applications.

We selected commitBERT as our PL to NL model due mostly to its simplicity as well as its language support (Python, PHP, Go, Java, JavaScript, and Ruby) [6]. We also plan to use GitPython python library [1].

The first milestone would be getting CodeBERT parsing *what* has changed from commit differentials. The next milestone would be to have a working dialogue a user is faced with when making changes through this tool. I feel as though we have a lot of leeway in terms of project scope and the amount of git tools we can integrate within our

wrapper. If we have extra time we can add more features, and if we are limited we may remove some. Our final goal however, regardless, will be a command line dialogue tool.

REFERENCES

[1] GitPython/index.rst at d438e088278f2df10b3c38bd635d7207cb7548a6 · gitpython-developers/GitPython.

[2] Raymond P.L. Buse and Westley R. Weimer. Automatically documenting program changes. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ASE '10, page 33–42, New York, NY, USA, 2010. Association for Computing Machinery.

[3] Khashayar Etemadi and Martin Monperrus. On the relevance of cross-project learning with nearest neighbours for commit message generation. In *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*, ICSEW'20, page 470–475, New York, NY, USA, 2020. Association for Computing Machinery.

[4] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. Codebert: A pre-trained model for programming and natural languages. *CoRR*, abs/2002.08155, 2020.

[5] Emitza Guzman, David Azócar, and Yang Li. Sentiment analysis of commit comments in github: An empirical study. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, page 352–355, New York, NY, USA, 2014. Association for Computing Machinery.

[6] Tae-Hwan Jung. Commitbert: Commit message generation using pre-trained programming language model. *CoRR*, abs/2105.14242, 2021.

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

[8] Shangqing Liu, Cuiyun Gao, Sen Chen, Lun Yiu Nie, and Yang Liu. Atom: Commit message generation based on abstract syntax tree and hybrid ranking, 2019.

[9] Zhongxin Liu, Xin Xia, Ahmed E. Hassan, David Lo, Zhenchang Xing, and Xinyu Wang. Neural-machine-translation-based commit message generation: How far are we? In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ASE 2018, page 373–384, New York, NY, USA, 2018. Association for Computing Machinery.

[10] Wei Tao, Yanlin Wang, Ensheng Shi, Lun Du, Shi Han, Hongyu Zhang, Dongmei Zhang, and Wenqiang Zhang. On the evaluation of commit message generation models: An experimental study. *CoRR*, abs/2107.05373, 2021.

[11] Yingchen Tian, Yuxia Zhang, Klaas-Jan Stol, Lin Jiang, and Hui Liu. What makes a good commit message? In *Proceedings of the 44th International Conference on Software Engineering*, ICSE '22, page 2389–2401, New York, NY, USA, 2022. Association for Computing Machinery.

[12] Haoye Wang, Xin Xia, David Lo, Qiang He, Xinyu Wang, and John Grundy. Context-aware retrieval-based deep commit message generation. *ACM Trans. Softw. Eng. Methodol.*, 30(4), jul 2021.