Commit Message Generation
Advances & Opportunities

Oshita Saxena
Mentors: Prof. Venkatesh Vinayakarao (CMI) and Dr. Monika Gupta (IBM)

Chennai Mathematical Institute
IBM Global Remote Mentoring (GRM) Program

June 14, 2021 - September 14, 2021

Abstract

It has become a prevalent practice in software engineering projects to collaborate through version control systems like GIT. The code change along with a log message is documented as a commit on GitHub. The log message describes new features or fix bugs in the changed code. But due to time constraints programmers avoid writing good quality log messages. Hence the need for automated log message generation. During the GRM program we studied and analysed some existing log message generation techniques that are NMT [3], NNGen [4] and LogGen [2]. However, our focus is on LogGen which is currently the state-of-the-art approach. It was observed that the good results for LogGen model were based only on syntactic of the code changes. The code change embeddings that were claimed to be representative of the semantics of the code change gave poor results when used for log message generation.

1 Summary

The data set includes commits from top 1k Java projects on GitHub. Each commit consists of a code change and corresponding log message. The code change indicates the lines of code removed or added by way of minus and plus signs respectively. Given an unseen code change our aim is to generate an appropriate log message.

1.1 NMT

Neural Machine Translation (NMT) [1], originally designed for natural language translations, is neural networks that model the translation process from a source language sequence to a target language sequence. RNN Encoder-Decoder has been used for this model. It has two recurrent neural networks.

1) Encoder: The encoder is a bidirectional recurrent neural network that reads the code change tokens $(x_1, x_2, \cdots, x_T)$ into a sequence of backward hidden states $(\overleftarrow{h_1}, \overleftarrow{h_2}, \cdots, \overleftarrow{h_T})$ and forward hidden states $(\overrightarrow{h_1}, \overrightarrow{h_2}, \cdots, \overrightarrow{h_T})$. For each token x_i, its vector representation is computed as $h_i = [\overrightarrow{h_i}; \overleftarrow{h_i}]$, which is a concatenation of $\overrightarrow{h_i}$ and $\overleftarrow{h_i}$.

2) Decoder: The decoder computes the conditional probability of the next symbol y_t using the context vector c_t and the previous symbol y_{t-1}. The context vector c_t is distinct for y_t and is computed by

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

where T is the length of the input sequence and the weight α_{ti} can be jointly trained with other parameters.

As a result of the bidirectional RNN, the representation of each code token has information from preceding tokens as well as the following tokens. Context vector is the part where attention is assigned to each code token. Through the context vector the model searches for a set of positions in the encoder hidden states where the most relevant information is stored.

1.2 NNGen

A later study [4] on NMT based log message generation revealed that the high quality messages generated by the model were identical to some messages in the training data set. Also, the code changes of most of the high quality messages generated were similar to one or more training code changes at the token level. Hence, they came up with a simpler and faster approach, Nearest Neighbour Generator (NNGen). NNGen is based on the nearest neighbour algorithm and does not require training. Here, the code changes are encoded via the bag-of-words algorithm. The bag-of-words vector representation is helpful in comparing the code changes at the token level or in other words, syntactically. However, the grammar and the word order are ignored, only term frequencies remain. For an unseen code change NNGen computes its vector representation and finds the top k nearest code change vectors from the training data set based on cosine similarity. These k code changes are compared with the unseen code change by way of BLEU-4 and log message corresponding to the code change with highest BLEU-4 score is generated.

1.3 LogGen

The bag-of-words encoding, as mentioned earlier, fails to capture the token order and hence the semantics of code change. The LogGen model uses code change vectors generated by the CC2Vec [2] paradigm and then uses the nearest neighbour as in NNGen model. Through this paradigm, we get distributed representations of the code changes such that vectors of similar code changes lie close to each other. This is done by exploiting the hierarchical structure of code changes by using attention mechanism in a Hierarchical Attention Network (HAN). The CC2Vec framework has the following five parts:

1) Preprocessing: Splits the code change into files and constructs code and log message vocabulary.

2) Input layer: Constructs two three dimensional matrices $\mathcal{B}_a$ and $\mathcal{B}_r$ representing added and removed code lines.

3) Feature extraction layer: The HAN first constructs two embedding vectors e_a and e_r for added and removed code lines. Then the comparison layer gives a feature vector e_{fi} which is the file embedding vector for i^{th} file.

4) Feature fusion layer: Concatenates the file embeddings to get a single patch embedding vector, e_p.

5) Word prediction layer: Maps the vector representation of the code change to a word vector extracted

	LogGen	*NNGen*	*NMT*
Score	**20.48**	16.42	14.19

Table 1: Performance of each approach

from the first line of log message; the word vector indicates the probabilities that various words describe the patch.

CC2Vec employs the first line of the log message of a patch to guide the learning of a suitable vector that represents the code change.

2 Observations and Experiments

Although there is a remarkable improvement in LogGen in terms of BLEU-4, there is still a long way to go. In this section, some observations related to LogGen will be discussed that are entirely based on experiments.

2.1 Observations

1) Value of k? While examining the public source code[1], it was observed that there was no default value of k in the code. It was also reported [6] that the implementation of LogGen is inconsistent with the descriptions in the CC2Vec paper. After changing the code according to the paper, the BLEU-4 score drops to as low as 2.3 for $k = 4$ and 1.97 for $k = 20$, as shown in Figure 1 and 2. This suggests that the good results were based only on the syntactic similarity among code changes.

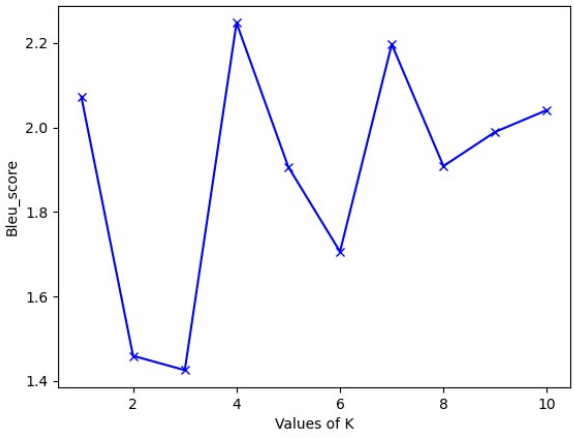

Figure 1: BLEU-4 score for k=1 to 10

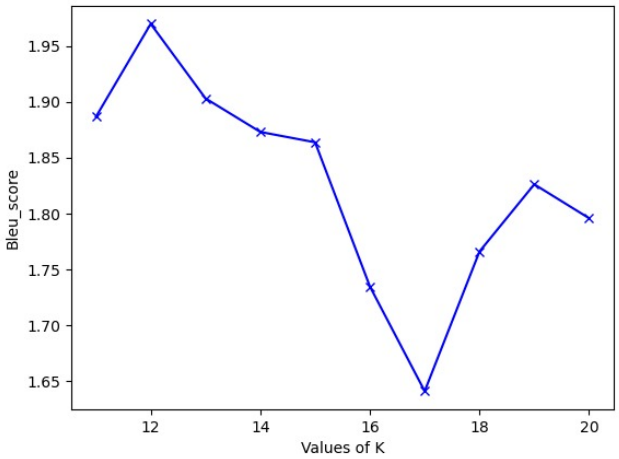

Figure 2: BLEU-4 score for k=11 to 20

2) High cosine similarities: The above observation led to the inspection of top 100 nearest code changes based on CC2Vec representation and cosine similarity as distance metric. For each test code change, the top 100 training code changes were recorded in an excel sheet[2] and examined. Very less number of log messages in the top 100 patches are similar in meaning to the actual log message. Then after inspecting the cosine similarities among the vectors, it was observed that the cosine similarity between some test code change vectors and training code change vectors was in the range 0.9 to 1. Which suggests that all training code changes are somewhat similar to the test code change, but this should not be the case if the vectors capture the meaning and hierarchical structure.

2.2 Experiments

1) Using jaccard instead of BLEU: The LogGen method compares the test code change with the top k training code changes using BLEU-4 score. After replacing BLEU-4 with jaccard similarity here, it was observed that the BLEU-4 increased with k, as shown in Figure 3. The results were similar for edit distance in place of BLEU (edit distance was minimised). This suggests that similarity of code diffs at the token level is a better way to measure similarity for log message generation.

[1] https://github.com/CC2Vec/CC2Vec

[2] link to the sheet

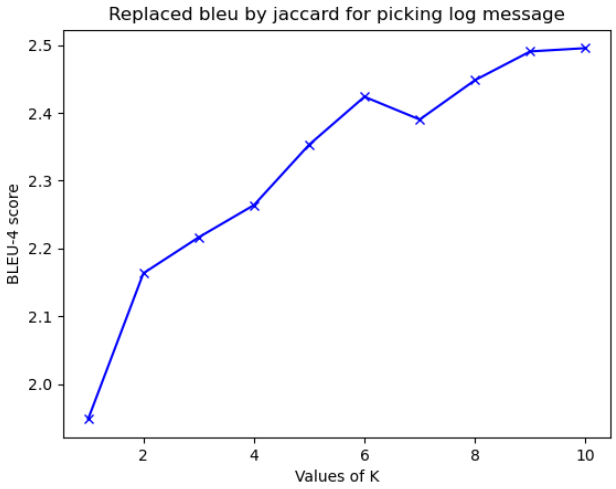

Figure 3: Jaccard instead of BLEU for picking message

2) Text summarization: Assuming that the CC2Vec representation of code changes efficiently capture their semantics and hierarchical structure, the way top k patches are handled can be modified. Hence, BERT extractive summarization [5] was applied on top k log messages to return one sentence summary of k messages. But there was no improvement in the results. Probable reasons for no improvement:

- BERT extractive summarization is used for summarizing natural language paragraphs. A key property of paragraphs is that context is passed over to different positions in the text. But there is no such thing happening in a collection of k log messages. Each log message is independent of other log messages and complete in itself. Also, log messages are not necessarily complete sentences following the rules of english grammar.
- This summarization method is computing embeddings for each log message and clustering them. The number of clusters is the number of sentences in the summary specified by the user. For our purpose, the number of clusters should be one. It becomes challenging for the model return just one log message that is properly representative of all the k messages.

3 Conclusion

From the above discussion it is clear that there is a lot of room for improvement for LogGen. The idea of attention can be exploited to generate log messages.
The experiments are available at https://github.com/oshita30/IBM-GRM.git.

References

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

[2] Thong Hoang, Hong Jin Kang, David Lo, and Julia Lawall. Cc2vec. *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, Jun 2020.

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

[4] 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*, pages 373–384, 2018.

[5] Derek Miller. Leveraging bert for extractive text summarization on lectures. *arXiv preprint arXiv:1906.04165*, 2019.

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