Towards Predicting Source Code Changes Based on Natural Language Processing Models: An Empirical Evaluation

Yuto Kaibe
Graduate School of Advanced Science and Engineering, Hiroshima University
Higashi Hiroshima, Japan

Hiroyuki Okamura
Graduate School of Advanced Science and Engineering, Hiroshima University
Higashi Hiroshima, Japan
okamu@hiroshima-u.ac.jp

Tadashi Dohi
Graduate School of Advanced Science and Engineering, Hiroshima University
Higashi Hiroshima, Japan
dohi@hiroshima-u.ac.jp

***Abstract*—In this paper, we investigate the prediction of software code changes using a natural language processing (NLP) model. NLP is one of the most rapidly developing fields in recent years, allowing various tasks related to natural language to be performed using large-scale models. In particular, BERT (bidirectional encoder representations from transformers) is a well-known model for encoding the input sentences of natural language into an appropriate vector space and is used for various classification tasks. In this paper, we use CuBERT (code understanding BERT), which was trained on programming languages as data in the pre-training stage, to perform tasks related to program code. Specifically, we run a regression problem where the output is the number of code changes.**

***Index Terms*—software code changes, prediction, BERT, Code understanding BERT**

I. Introduction

Agile development has gained a lot of traction in software development in recent years and is the development style in which software is developed with a small cycle consisting of build and test phases. In agile development, refactoring is one of the most important activities for developing high-quality software. Refactoring is a technique for restructuring and rewriting software code without changing its interfaces. Therefore, we observe the frequent changes in source code in agile development.

On the other hand, the frequency of source code changes is available as a metric to check the health of software development. It is well known that such changes occur frequently in the early phase of software development because user requirements and related software architectures are not stable in the early phase. It can also be seen that the frequency of source code changes gradually decreases as the software matures. This phenomenon is similar to the growth of software reliability, i.e., the number of detected defects decreases as software testing progresses in traditional waterfall software development. In traditional software reliability engineering, researchers have discussed the trends of defects detected in the testing process with statistical models [1], [2], and have evaluated the quality of software products statistically. This means that even in agile development, we obtain statistical information about the quality of software products from the empirical data of the number of source code changes.

Popular bug count analyses are fault-prone module prediction and bug prediction. Fault-prone module prediction is the problem of determining a software module that is expected to contain software bugs, and bug prediction is the problem of estimating the number of software bugs contained in the software. Fault-prone module prediction and bug prediction belong to the discriminant and regression problems in statistics, respectively, whose input variables are given by software metrics. The software metrics are quantitative values that summarize the characteristics of software, such as lines of code and complexity. In practice, however, it is difficult to determine the best software metrics for defect analysis. In addition, there is no good practice on which metrics should be measured to improve prediction accuracy.

Outside of programming languages, the field of Natural Language Processing (NLP) has seen dramatic improvements in machine learning techniques. In particular, BERT (bidirectional encoder representations from transformers) [3] is one of the most successful NLP models in a variety of tasks such as language translation. A key point of BERT is the technique of embedding, where tokens are numerically embedded in the vector space. This concept allows us to skip the procedure of extracting and selecting good metrics and is expected to be useful for the analysis of programming languages. In fact, Kanade et al. [4] discussed the embedding-based NLP model called CuBERT, which is BERT whose pre-training used program source codes. CuBERT outperformed the previous models in five classification tasks and one modification task for source codes. This implies that it is effective to use the data collected from the target domain in pre-training. However, the question remains whether this is always effective.

This paper attempts to discuss the applicability of NLP models for the task of predicting source code changes, i.e., BERT and CuBERT can be applied to predict the number of source code changes based on the information about the source codes directly. Here we consider the following research questions:

RQ1 How effective is the NLP-based embedding approach for the prediction task?

RQ2 What is the best approach for retraining NLP models?

RQ3 What is the appropriate model for predicting the number of source code changes?

In RQ1 we try to show how effective BERT and CuBERT are in predicting the number of source code changes compared to the conventional metrics-based regression approach. In addition, by comparing BERT and CuBERT, we discuss the effect of pre-training data on the accuracy of specified tasks.

RQ2 is related to the retraining strategy. In general, there are two phases of training in modern AI models: pre-training and retraining. The pre-training is to adjust the model parameters to learn the basic knowledge in the target domain with a large amount of training data. Retraining is to make small adjustments to the model parameters so that the pre-trained AI model fits the specified task with a small amount of data. Generally, there are two retraining strategies: fine-tuning and transfer learning. In RQ2, we try to answer the question which is better for fine-tuning and transfer learning to predict the number of source code changes.

In RQ3, we discuss the architecture of regression models. Apart from the neural network based models, the generalized linear regression model (GLM) is widely used for practical problems. In the context of GLM, the GLM models are generally classified based on the link functions and their associated loss functions. In the past literature, the regression task for the number of defects was performed with linear and Poisson regression models as specific models of GLM. Even in the NLP-based models, we can deal with both linear and Poisson regression-type models. RQ3 attempts to reveal the predictive power of these models.

II. Related Works

The statistical behavior of the number of source code changes is similar to the behavior of the number of defects. Predicting the number of defects is one of the most important tasks in software reliability engineering. As mentioned in the introduction, defect prediction is categorized into fault-prone module detection and defect number estimation, which are related to discrimination and regression in the statistical problems, respectively.

Fault-prone module detection is the process of identifying software modules that contain software defects in a software project. Once the defective modules can be identified, we reduce the testing effort by focusing software testing on the defective modules. Shen et al. [5] discussed module detection as an early paper. Since the 1990s, several researchers [6]–[8] have proposed methods based on software metrics such as CK metrics [9]. On the other hand, in predicting the number of defects, Khoshgoftaar [10] discussed Poisson regression with software metrics.

III. NLP Models

A. BERT

Natural language processing (NLP) has been one of the most attractive AI models in recent years. In fact, a well-known NLP model called BERT uses deep learning techniques to extract important features from raw text data. The architecture of BERT is based on the transformer [11]. Roughly speaking, BERT is an extension of recurrent neural networks (RNNs) that deal with time series data. The RNN reveals the dependence of the time series data based on one direction. BERT represents the bidirectional dependency of time series data with an attention mechanism, which is the most important feature of BERT. Since BERT uses a bidirectional structure, it essentially outputs a fixed-size vector sequence from a fixed-size word sequence, i.e., a sentence.

The BERT pre-training consists of the following two tasks:

- The Masked Language Model (MLM): First, some of the words in an input sentence are replaced by the special token [MASK]. BERT is trained to predict which are the original words for the masked ones from the context of the sentence. For example, consider the following sentence:
  ```
  This morning he had breakfast.
  ```
  The training data is generated by replacing randomly selected words with [MASK], e.g,
  ```
  This morning he [MASK] breakfast.
  ```
  BERT trains to guess the original words only from the words before and after the masked words. In the MLM task, a selected word is replaced by [MASK] with probability 0.8 and by another randomly selected word with probability 0.1. Otherwise, it remains with a probability of 0.1.
- The Next Sentence Prediction (NSP): Consider two sentences separated by the special token [SEP]. BERT predicts whether or not there is a semantic connection between the two sentences from the context of the two sentences. For example, consider the following sentences
  ```
  [CLS] I [MASK] to a bookstore.
  [SEP] There I bought three books.
  ```
  There is a semantic connection between the two sentences above. On the other hand, it is clear that there is no semantic connection between the following two sentences:
  ```
  [CLS] I [MASK] to a bookstore.
  [SEP] People are mammals.
  ```
  BERT learns to predict the semantic connection between them as a discriminant problem. The training data is organized in such a way that 50% are positive examples, i.e. two sentences have a semantic connection, and the remaining 50% are negative examples, i.e. there is no semantic connection. Also, [CLS] is the special token to express the beginning of the sentence.

B. CuBERT

CuBERT (code-understanding BERT) [4] is a natural language processing model based on BERT and oriented towards the processing of programming languages. As mentioned above, the pre-training tasks of BERT are MLM and NSP tasks with a large natural language corpus. On the other hand, CuBERT is trained on a large source code corpus. Although the difference between BERT and CuBERT is the training data, CuBERT outperformed other machine learning models, including BERT, on five programming language classification tasks and one program modification task [4].

IV. Experiment I

To provide answers to RQ1 through RQ3, we conduct the experience of predicting the number of changes for software projects. We collect Java programs from GitHub as retraining data. In the style of Java language, one file contains one Java class and each class has several methods. The collected data includes 74 classes and a total of 200 methods. Since the input of BERT and CuBERT is one method, we divide 200 methods into 150 methods as training data and 50 methods as validation data. The number of source code changes for each method is also collected from GitHub.

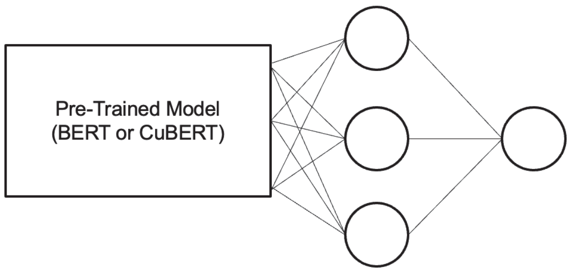

Fig. 1. The architecture of model.

To accomplish the prediction task, we use the network architecture shown in Figure 1. The BERT and CuBERT work for encoding the source codes into the vector sequences. The following part corresponds to the regression model to predict the number of source code changes from the vector sequences as inputs. Furthermore, since the number of source code changes may depend on the lines of code of the method itself, the prediction task is to estimate the number of source code changes per token, i.e., the number of source code changes is divided by the number of tokens of the method.

We also consider two regression models as the last part of the network. Basically, the regression model consists of two linear and dense layers. The output of the last layer is a scalar value. In the case of linear regression, the output represents the number of source code changes per token as it is. The loss function of linear regression is a squared error between the output and the data. On the other hand, in the case of Poisson regression, the number of source code changes is given by the logarithm of the output. The loss function of Poisson regression is the negative log-likelihood function of the Poisson distribution. Based on this architecture, fine-tuning adjusts all layers consisting of BERT or CuBERT and the regression model to minimize the defined loss function. Transfer learning adjusts only the parameters of the regression model. In both cases, learning (optimization) is performed by Adam with 10 and 20 epochs.

The table I shows the results of the prediction task. In the table, the columns of NLP and Regression indicate the models used for NLP and regression, respectively. Also, the Retraining and Epoch columns show the retraining strategies and the number of iterations for updating the parameters with mini-batch data. The MPSE column shows the mean squared prediction error between the prediction of the number of source code changes per token and the validation data. Table II shows the results of the prediction task for the metrics-based model. In this experiment, we collect the number of tokens, the number of if-statements, and the number of loops as metrics. The regression model, i.e. the last part of the model, is used to obtain the number of source code changes per token from these metrics.

RQ1: How effective is the NLP-based embedding approach for the prediction task?
Comparing tables I and II, we see that the best (smallest) MPSE of the NLP-based approach is 0.044, while the best MPSE of the metrics-based model is 0.921. Also, in many cases, there is a tendency for the MPSEs of the NLP-based approach to be smaller than those of the metrics-based model. In particular, CuBERT outperforms BERT in terms of MPSE. The reason why CuBERT is better than the others may be that CuBERT was trained from source codes in the pre-training phase. Since the architectures of BERT and CuBERT are the same, this implies that it is important that the pre-training data should be collected from the domain related to the subsequent tasks.

RQ2: What is the best retraining approach in NLP models?
In this experiment, we applied two retraining strategies; fine-tuning and transfer learning. In the case of BERT, the minimum MPSEs of fine-tuning and transfer learning are 2.063 and 0.749, respectively. Similarly, in the case of CuBERT, the MPSE of fine-tuning is 0.044 and that of transfer learning is 0.907. That is, transfer learning is superior to fine-tuning in the case of BERT, we get the opposite result in the case of CuBERT. This may be due to the fact that BERT is not suitable for understanding source codes, since it was trained with the natural language corpus. As mentioned before, the fine tuning adjusts all model parameters, including BERT. Since BERT is not fully trained in the pre-training stage, the parameters of BERT are also significantly changed in the fine-tuning stage. However, the data size of fine tuning is not sufficient to tune BERT. On the other hand, CuBERT can be fully trained with the corpus using source codes in the pre-training stage. Then, the parameters of CuBERT are not changed in the fine-tuning stage, and the fine-tuning focuses on updating the model parameter of the regression model so that it fits a given specific task.

TABLE I
RESULTS OF PREDICTION TASK WITH NLP MODELS.

NLP	Retraining	Regression	Epoch	MPSE
BERT	Fine-Tuning	Linear	10 epochs	2.063
			20 epochs	2.175
		Poisson	10 epochs	2.428
			20 epochs	2.247
	Transfer Learning	Linear	10 epochs	0.918
			20 epochs	**0.749**
		Poisson	10 epochs	1.643
			20 epochs	0.873
CuBERT	Fine-Tuning	Linear	10 epochs	2.833
			20 epochs	0.617
		Poisson	10 epochs	0.611
			20 epochs	**0.044**
	Transfer Learning	Linear	10 epochs	0.907
			20 epochs	0.888
		Poisson	10 epochs	0.866
			20 epochs	0.821

TABLE II
RESULTS OF PREDICTION TASK WITH METRICS-BASED MODEL.

Regression	Epoch	MPSE
Linear	10 epochs	0.958
	20 epochs	1.159
Poisson	10 epochs	**0.921**
	20 epochs	1.124

RQ3: What is the appropriate model to predict the number of source code changes?
Comparing the MPSEs of the linear and Poisson regression models in tables I and II, the linear regression model is better than the others in the case of BERT. Also in the case of CuBERT, the Poisson regression model outperforms the linear model. This is also due to the fact that BERT is not suitable for dealing with source code.

We summarize the answers to the research questions. In the experiment, the best approach is the NLP-based approach with CuBERT, fine-tuning, and Poisson regression model. In this case, MPSE takes the lowest value of 0.044.

V. CONCLUSION

In this paper, we have discussed the NLP-based approach to predict the number of source code changes. In particular, we have dealt with two NLP models; BERT and CuBERT, and have compared them in the experiment. As a result, CuBERT outperformed BERT because CuBERT was trained with the corpus of source codes as pre-training. This implies that the importance of the pre-training and the training data of the pre-training should be appropriate for the subsequent specific tasks. We have also compared the retraining strategies; fine-tuning and transfer learning. In the context of NLP models, fine-tuning is effective to adjust the model parameters when the NLP model is fully trained in the pre-training stage. On the other hand, transfer learning works when the NLP model is not fully trained. The same tendency can be found in the difference between linear and Poisson regression models. As a result, the NLP model with CuBERT is effective under the fine tuning and Poisson regression model.

In this paper, we have used CuBERT as an NLP model for understanding source code. On the other hand, there have been several NLP models dealing with source code in recent research [12]. Therefore, we will conduct an experiment using such recent NLP models. Furthermore, in this paper, we use the number of source code changes as static values, but they change with the elapsed time. In other words, we should consider that the model deals with the time series data.

REFERENCES

[1] M. Xie, *Software Reliability Modelling*. Singapore: World Scientific, 1991.

[2] M. R. Lyu, Ed., *Handbook of Software Reliability Engineering*. New York: McGraw-Hill, 1996.

[3] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," in *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. Minneapolis, Minnesota: Association for Computational Linguistics, Jun. 2019, pp. 4171–4186.

[4] A. Kanade, P. Maniatis, G. Balakrishnan, and K. Shi, "Learning and evaluating contextual embedding of source code," in *Proceedings of the 37th International Conference on Machine Learning*, ser. ICML'20. JMLR.org, 2020.

[5] V. Shen, T. jie Yu, S. Thebaut, and L. Paulsen, "Identifying error-prone software—an empirical study," *IEEE Transactions on Software Engineering*, vol. SE-11, no. 4, pp. 317–324, 1985.

[6] V. Basili, L. Briand, and W. Melo, "A validation of object-oriented design metrics as quality indicators," *IEEE Transactions on Software Engineering*, vol. 22, no. 10, pp. 751–761, 1996.

[7] L. C. Briand, J. Wüst, J. W. Daly, and D. Victor Porter, "Exploring the relationships between design measures and software quality in object-oriented systems," *Journal of Systems and Software*, vol. 51, no. 3, pp. 245–273, 2000.

[8] Y. Zhou and H. Leung, "Empirical analysis of object-oriented design metrics for predicting high and low severity faults," *IEEE Transactions on Software Engineering*, vol. 32, no. 10, pp. 771–789, 2006.

[9] S. Chidamber and C. Kemerer, "A metrics suite for object oriented design," *IEEE Transactions on Software Engineering*, vol. 20, no. 6, pp. 476–493, 1994.

[10] T. Khoshgoftaar, K. Gao, and R. Szabo, "An application of zero-inflated poisson regression for software fault prediction," in *Proceedings 12th International Symposium on Software Reliability Engineering*, 2001, pp. 66–73.

[11] A. Vaswani, S. Bengio, E. Brevdo, F. Chollet, A. Gomez, S. Gouws, L. Jones, Ł. Kaiser, N. Kalchbrenner, N. Parmar, R. Sepassi, N. Shazeer, and J. Uszkoreit, "Tensor2Tensor for neural machine translation," in *Proceedings of the 13th Conference of the Association for Machine Translation in the Americas (Volume 1: Research Track)*. Boston, MA: Association for Machine Translation in the Americas, Mar. 2018, pp. 193–199.

[12] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. Liu, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu, M. Tufano, S. K. Deng, C. B. Clement, D. Drain, N. Sundaresan, J. Yin, D. Jiang, and M. Zhou, "Graphcodebert: Pre-training code representations with data flow," in *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net, 2021.