RevToken: A Token-Level Review Recommendation: How Far Are We?

Yasuhito Morikawa
NAIST
Nara, Japan
morikawa.yasuhito.mx2@is.naist.jp

Yutaro Kashiwa
NAIST
Nara, Japan
yutaro.kashiwa@is.naist.jp

Kenji Fujiwara
Nara Women's University
Nara, Japan
kenjif@ics.nara-wu.ac.jp

Hajimu Iida
NAIST
Nara, Japan
iida@itc.naist.jp

***Abstract*—Code review plays an important role in quality assurance, which improves readability, maintainability, etc. On the other hand, code review is notorious for being very time-consuming work because reviewers need to carefully inspect numerous lines for each change. To alleviate the efforts, many studies have proposed approaches to highlighting the code that reviewers need to review. However, even using the finest-grained approaches (*i.e.,* line-level), the granularity of the recommendations is coarse-grained so it is difficult for developers to figure out what needs to be reviewed. For example, lines in the QtBase projects have a median of 7 tokens and sometimes have more than 12 tokens (10% of the lines).**

This study proposes a token level approach to recommend where developers review, which is finer-grained than the previous studies. Specifically, we fine-tune CodeBERT so that it can return the tokens that are most likely to be commented on and revised. Our empirical evaluation using the OpenStack and QtBase datasets demonstrated that the proposed approach outperforms the state-of-the-art model to predict lines to be revised. Also, we find that 86% of the predicted tokens are accurately distinguished as either needing modification or not when the lines to be revised are correctly identified.

***Index Terms*—Software Quality Assurance, Modern Code review, CodeBERT**

I. Introduction

Code review plays an important role in quality assurance, which improves readability and maintainability [1]–[3], as well as helps to share knowledge [1]–[4] and find bugs [1], [3]–[6]. On the other hand, code review is notorious for being very time-consuming work because reviewers need to carefully inspect numerous lines for each change [1], [3], [5], [7]. As reviewers receive many review requests from the patch authors, most of the first feedback from reviewers is frequently provided after 15–64 hours [7].

To alleviate the efforts, many studies have proposed approaches to automate a part or all of the reviewing tasks. Hellendoorn *et al.* [8] have proposed a method for predicting hunks (*i.e.,* groups of lines) that reviewers need to point out. Furthermore, Hong *et al.* [7] have proposed the finer-grained method for predicting lines that need to be reviewed. These methods can help reviewers to identify which part of the code that needs to be inspected. Still, even using the line level approach, the granularity of the recommendations is coarse-grained. Figure 1 shows an example of lines that need to be revised. In the code (*i.e.,* the right side), there are multiple variables in Line 122. Even the line level recommendation highlights Line 122 to be revised, it is difficult for reviewers to figure out the potential issue. In fact, the median number of tokens per line is 7 in the QtBase project, and 10% of lines have more than 12 tokens.

In this study, we propose a token level approach that automatically suggests where developers need to review. Specifically, the proposed method takes the diff files in the change as input and splits the files into tokens. Then, we fine-tune a pre-trained model, CodeBERT with a set of the tokens in each line and a label showing whether the line will receive review comments and be revised in the next revision. When receiving the diff files to be predicted, the trained model infers which tokens will receive review comments and will be modified, using the weight calculated by the Attention and LIME mechanisms.

Replication package. To facilitate replication and further studies, the data used in this study are publicly available on GitHub repository.[1]

II. RevToken: A Token-Level Review Recommendation

This paper proposes `RevToken`, which can automatically suggest where reviewers need to review. This section describes the overview and each process of `RevToken`.

A. Overview

`RevToken` is designed to take diff files (*i.e.,* a commit) as input, split the modified lines into tokens, and return the lines and their tokens that need to be reviewed. To predict which tokens need to be reviewed, we fine-tune CodeBERT so that it takes diff files in the first change of a review and returns which tokens need to be revised in the next revision of the review process. CodeBERT is a pre-trained model trained on natural language and multiple programming languages (*e.g.,* Python, Java, JavaScript, PHP, Ruby, and Go). Many previous studies [9]–[12] reported outstanding performance on code-related tasks. Specifically, we first prepare lines split from the diff files and the corresponding labels showing if each line receives review comments and is revised.

[1]https://github.com/yattinda/RevToken

Fig. 1: Motivating example that shows a line containing many tokens (https://codereview.qt-project.org/c/qt/qtbase/+/522806/5..6)

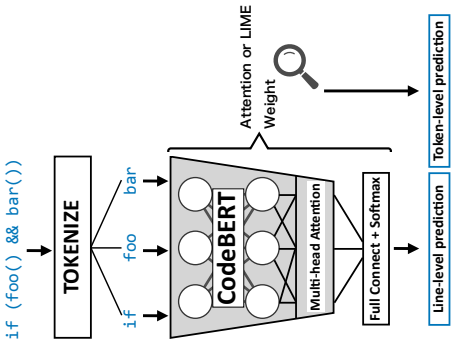

Fig. 2: Overview of the model structure

With the split lines, we then fine-tune CodeBERT, and the attention mechanism of CodeBERT assigns weights to each token based on its relevance to predicting the output (*i.e.,* whether it will be revised or not), capturing tokens that will receive review comments and be revised. Utilizing these weights, `RevToken` determines which tokens are more likely to be revised. It is worth noting that `RevToken` does not require token level labels, `RevToken` internally splits the lines into tokens and learns which tokens induce review comments and changes.

B. Preprocessing

First of all, `RevToken` takes only the lines that are added or modified in a change (i.e., "++" in the diff file) in the same manner as the previous study [7] that found that the deleted lines rarely received comments from reviewers. We then replace numeric literals with a special token "NUM". Also, we remove non-alphanumeric characters such as semicolons and operators from the lines. This is because numeric literals and characters are often meaningless, which leads to performance degradation in the model. Finally, we prepare the corresponding label for each line, which is a binary label (*i.e.,* true or false) indicating whether the line received comments and was revised in the next revision in a review.

C. Fine-tuning

We describe how `RevToken` learns fine-tuned tasks. Figure 2 shows the overview structure of the model. `RevToken` receives the changed lines, splits the input source code into tokens by the tokenizer of CodeBERT, and extracts the features as sequence data. Then, the extracted features are fed into a fully connected layer and a sigmoid function to obtain the prediction result. These processes perform a binary classification on whether the line will receive comments and be revised.

Loss function: The number of lines that receive review comments is expected to be significantly less than that of lines without review comments, which results in imbalanced datasets. Therefore, we employ a loss function that imposes a large penalty when the minority data is misclassified during learning. Specifically, a combination of the sigmoid function and Binary Cross Entropy (BCE) is used as the loss function.

D. Inferencing

To perform the token level prediction, `RevToken` conducts two-fold predictions, line and token level. We describe each step to predict tokens that need to be reviewed as follows.

1) Line-level prediction: As the first step, `RevToken` predicts the lines that will receive review comments and be revised in future revisions during a review. To do so, `RevToken` takes lines that are added in the changes as input and performs preprocessing to the lines in the same manner as Section II-B. `RevToken` returns a binary value that shows whether the line will receive review comments and be revised. The binary value is calculated from the probability distribution obtained by CodeBERT. However, as this probability distribution ranges from 0 to 1, the continuous values need to be transformed into binary. Therefore, if the probability distribution is greater than 0.5, `RevToken` returns true (*i.e.,* the line needs to be reviewed). If not, `RevToken` answers false (*i.e.,* the line does not need to be reviewed).

2) Token-level prediction: In order to measure to what extent each token needs to be reviewed, we prepare two different approaches, the Multi-head Attention and LIME. `RevToken` recommends the tokens that have the top-N weights calculated by the following approaches (the N is any integer value).

Multi-head Attention: The first approach is an approach that utilizes the weight of each token provided by the Multi-head Attention (we used the final layer of the multiple Multi-head attention in CodeBERT). The weight shows the contribution to the prediction of each token when it is predicted that the source code will receive comments and be revised (*i.e.,* the possibility of change).

LIME: To identify tokens that are more likely to be reviewed, we utilize a Local Interpretable Model-Agnostic Explanations (LIME) [13]. LIME is a widely-used model-agnostic explainable algorithm for deep learning models. It computes the importance of scores for each token in a given prediction (*i.e.,* which tokens contribute most to the prediction), by randomly deleting tokens from the original data.

III. Case Study Design

To evaluate the proposed model, we perform an experiment encompassing two different granularities of evaluations, line level and token level. The following subsections describe the dataset, evaluation, and environment for the experiment.

A. Dataset

This study extends the dataset released by Hong *et al.* [7]. This dataset includes review records for QtBase and OpenStack Nova. Both systems use Gerrit code reviewing systems and are often used in previous studies [7], [14]–[16]. Their dataset contains added lines in changes, and a label for each line showing whether or not the line receives inline comments (if the line receives comments, the label is true. Otherwise, the label is false).

As described earlier, there are many lines that receive review comments but will not be changed (*i.e.,* review comments are not addressed). Therefore, in this study, we modify the labels of the dataset by investigating whether each line is modified in the next revision in the review. Thus, the true label shows that the line receives review comments and is modified in the next revision; the false label shows that the line does not receive comments, or receives comments but is not modified.

To evaluate the token level prediction, we need to identify which tokens are revised. In most cases, it is certain *which lines* received comments, but it is not clear *which tokens* the review comments point out in the datasets. To address this challenge, we identify which tokens in the line that received review comments will be modified in the next revision. First, we extract only the lines that receive review comments and are modified. We then separate each line into tokens. However, there are no tools that support both C++ (*i.e.,* for QtBase) and Python (*i.e.,* for OpenStack). Hence we use different tokenizers, srcDiff [17] for QtBase, CodeBERT's tokenizer and `diff-match-patch`[2] for OpenStack. While srcDiff can parse the lines into tokens and detect the modified tokens, CodeBERT's tokenizer identifies tokens only, so we employ `diff-match-patch`. This tool compares tokens before and after a change and then detects the changed tokens. We assign the tokens that have been changed with true labels, and the other tokens with false labels to evaluate token level predictions.

B. Evaluation

This study evaluates the performance of the models at both the line level and the token level predictions with the dataset described in Section III-A. We fine-tune the model, using 90% of the data (*i.e.,* training dataset) that are randomly selected from the dataset. With the remaining 10% of the data (*i.e.,* testing dataset), we evaluate the performance of the models to answer each research question, which are denoted as follows.

RQ 1: Does `RevToken` outperform the state-of-the-art model to predict lines to be revised? We first evaluate the performance at the line level prediction, comparing `RevToken` with the previous study (`REVSPOT`) [7]. `REVSPOT` is developed using the RandomForest algorithm with LIME to predict the lines. To measure the performance, we use the following three metrics that were employed in the previous studies [7]: *Recall*, False Alarm Rate (*FAR*), and distance to heaven (*d2h*).

1) *Recall* represents the ratio of the number of predicted lines that are expected to receive comments and be revised, out of the actual number of lines that will receive comments and be revised. This metric ranges from 0 to 1 and a higher value shows better performance.
2) *FAR* represents a proportion of the number of clean lines (that do not receive comments or are not revised) but that are incorrectly predicted as non-clean lines out of the number of clean lines. Specifically, this metric is formulated as (*i.e.,* FP / (TN + FP)). This metric ranges from 0 to 1 and a lower value shows better performance.
3) *d2h* represents the root mean square of the *Recall* and *FAR* values. This metric ranges from 0 to 1 and a lower value shows better performance. If *d2h* is 0, it means that `RevToken` has achieved perfect prediction (*i.e.,* when *Recall* is 1, *FAR* is 0).

RQ 2: How accurate is `RevToken` in predicting tokens to be revised? RQ2 evaluates the performance at the token-level predictions when the line-level predictions correctly identify the lines. Specifically, we examine to what extent the predicted tokens are revised in the next revision. First, we sort the values showing how likely the tokens are to be revised, which is produced by the Attention or LIME, and take the top-N tokens. In this study, we evaluate the performance when using $N = [1, 3, 5]$. Finally, with the top-N tokens, we measure *Precision*, *Recall* and *F1_Score*.

1) *Precision* represents what percentage of tokens are revised in the next revision out of the predicted tokens (*i.e.,* top-N tokens).
2) *Recall* represents what percentage of revised tokens are included in the predicted tokens.
3) *F1_Score* represents the harmonic mean of *Precision* and *Recall*.

C. Environments

To fine-tune the model, we used a server equipped with four Intel Xeon Gold 6230R CPUs, 250GB of memory, and two NVIDIA A100. To prevent over-learning, we terminated the fine-tuning process (*i.e.,* early stops) if the loss function value did not update the minimum value for 5 consecutive epochs. The number of epochs for OpenStack is 18, and for QtBase is 19. In addition, the batch size is 256. The fine-tuning of CodeBERT with QtBase or OpenStack eventually took about 2.5 hours.

IV. Experimental Results

This section reports the performance of the line level and token level predictions.

[2]https://github.com/google/diff-match-patch

TABLE I: Results of line-level evaluation

	OpenStack		QtBase	
	REVSPOT	RevToken	REVSPOT	RevToken
Recall	**0.67**	0.61	0.50	**0.68**
FAR	0.56	**0.28**	0.43	**0.30**
d2h	0.52	**0.34**	0.55	**0.31**

As for FAR and d2h, a lower values show a better performance.

A. Line-level evaluation

Table I shows the prediction of lines (*Recall*, *FAR*, and *d2h*) that are modified by both previous studies, (REVSPOT) [7] and RevToken. In the line-level evaluation, RevToken outperforms the previous study in most of the performance metrics except for the *Recall* in OpenStack. Particularly, in terms of the *d2h*, RevToken shows 35% and 44% of the improvement in OpenStack and QtBase, respectively. In addition, we observed 50% and 30% improvement of the *FAR* for OpenStack and QtBase, respectively. This suggests that our model is designed for token-level prediction but can also be used for line-level prediction.

RQ$_1$. RevToken *outperforms the state-of-the-art model in terms of all the performance measures (35% and 44% of the improvement in d2h in OpenStack and QtBase projects, respectively).*

B. Token-level evaluation

We performed the token-level predictions on 98 lines in OpenStack and 78 lines in QtBase which were correctly predicted in RQ1. Table II shows the performances of RevToken with Attention and LIME at the token-level prediction.

When comparing Attention-based and LIME-based approaches, LIME-based approach outperforms Attention-based approach in all the performance measures. Specifically, as for *precision* in OpenStack, the *precisions* for top-1, 3, and 5 are 0.33, 0.31, and 0.30, respectively. In QtBase, the *precisions* are 0.27, 0.26, and 0.26, respectively. These imply that only 30% of the predicted tokens are revised in reality.

Specifically, regarding recall in OpenStack, the *recalls* for top-1, 3, and 5 are 0.20, 0.55, and 0.77, respectively. In QtBase, the *recalls* are 0.21, 0.56, and 0.86, respectively. In other words, with the top-5 recommendations, RevToken can highlight most of the tokens that will be revised.

Figure 3 shows a successful example of the token-level. In this example, two tokens are replaced with a token after a reviewer points out the inappropriate implementation, mentioning *"I guess, you could use 'el.isOdd()' instead?"*. On the other hand, RevToken recommended "const", "value", and "1" as that will be more likely to be revised in the next revision. Two out of three of the predictions turned out to be correct: "value" and "1". Even though this line has 9 tokens, RevToken suggested three tokens and two tokens should be revised, which would guide reviewers to point out and prevent them from missing issues.

TABLE II: Results of token-level evaluation

			Precision	Recall	F1_Score
Open Stack	top-1	Attention	0.32	**0.20**	**0.25**
		LIME	**0.33**	**0.20**	**0.25**
	top-3	Attention	**0.31**	0.54	0.39
		LIME	**0.31**	**0.55**	**0.40**
	top-5	Attention	0.29	0.75	0.42
		LIME	**0.30**	**0.77**	**0.43**
QtBase	top-1	Attention	0.23	0.18	0.20
		LIME	**0.27**	**0.21**	**0.24**
	top-3	Attention	0.22	0.48	0.30
		LIME	**0.26**	**0.56**	**0.36**
	top-5	Attention	0.23	0.76	0.35
		LIME	**0.26**	**0.86**	**0.40**

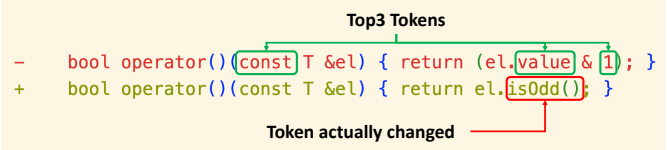

Fig. 3: Examples of correctly predicted

RQ$_2$. *LIME-based approach performs better than Attention-based approach, achieving at 0.77 and 0.86 of recall in OpenStack and QtBase with top-5 recommendations.*

V. Future Directions

A. Can RevToken predict the revised tokens when there are many tokens in a line?

Our evaluation demonstrates that the token-level prediction performs well in the dataset that we used. However, the dataset contains many lines with several tokens. Thus, it is not clear to what extent RevToken can predict the tokens to be revised when there are many tokens in a line. In this section, we discuss the performance to predict the tokens that will be revised when there are many tokens.

We first measured the number of tokens contained in each line and the number of changed tokens for each line, which are shown in Figure 4.[3] In both OpenStack and QtBase, the median numbers of tokens contained in changed lines are 7.0. Additionally, the median number of changed tokens is 2.0 in OpenStack and 1.0 in QtBase. This suggests that the line-level prediction recommends up to 7.0 times more tokens than the tokens that are actually revised. Also, this means that top-5 tokens recommendation may predict all the tokens in a line to be revised.

We thus decided to calculate the performance measure with the lines that have more than the median number of tokens (*i.e.,* 7 or more tokens). This filtering resulted in eliminating 49 lines and 28 lines in OpenStack and QtBase, respectively. And finally, 49 lines and 50 lines remained, respectively. After filtering, the median of tokens becomes 10 from 7 and the median of the revised tokens becomes 2 from 1 in QtBase.

[3]Due to the limited space, we only show the figure for QtBase but we observe a similar distribution in OpenStack

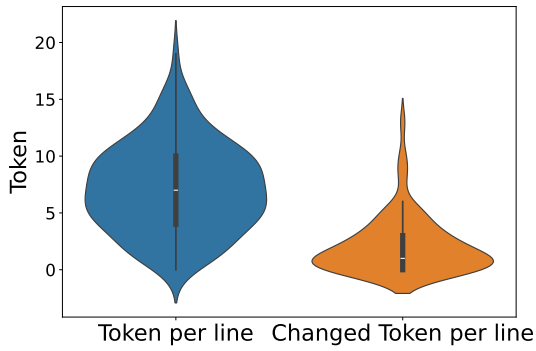

Fig. 4: Number of tokens in line (QtBase)

TABLE III: Token-level evaluation performance (using 7 or more tokens). The numbers in the brackets show the differences (%pt) from the original results

			Precision	Recall	F1_Score
Open Stack	top-1	Attention	0.12 **(-0.20)**	0.12 (-0.08)	0.12 (-0.13)
		LIME	0.14 **(-0.19)**	0.14 (-0.06)	0.14 (-0.11)
	top-3	Attention	0.14 **(-0.17)**	0.37 (-0.17)	0.20 (-0.19)
		LIME	0.14 **(-0.17)**	0.39 (-0.16)	0.21 (-0.19)
	top-5	Attention	0.16 **(-0.13)**	0.63 (-0.12)	0.25 (-0.17)
		LIME	0.16 **(-0.14)**	**0.66** (-0.11)	0.26 (-0.17)
QtBase	top-1	Attention	0.16 (-0.07)	0.16 (-0.02)	0.16 (-0.04)
		LIME	0.18 (-0.09)	0.18 (-0.03)	0.18 (-0.06)
	top-3	Attention	0.14 (-0.08)	0.38 (-0.10)	0.20 (-0.10)
		LIME	0.18 (-0.08)	0.50 (-0.06)	0.27 (-0.09)
	top-5	Attention	0.15 (-0.08)	0.66 (-0.10)	0.25 (-0.10)
		LIME	0.18 (-0.08)	**0.77** (-0.09)	0.29 (-0.11)

Table III summarizes the prediction results when lines have 7 or more tokens. As the table shows, the precision is significantly lowered (30%-63%). The degradation of recall is better than precision but still, there are 10% to 43% decreases. However, looking into the top-5 recommendations using LIME, the recall remains 0.66 and 0.77 in OpenStack and QtBase, respectively. This implies that `RevToken` would help developers identify the tokens to be revised. Our future work needs to enhance the performance when there are many tokens as well as improve the performance of identifying the lines to be revised.

B. How can `RevToken` identify the tokens that will be added after the review?

In this study, we proposed a token-level approach to recommend where developers review and show the potential to predict tokens that will be revised. However, the performance is still far from perfect. To improve the performance, we need to overcome the most serious challenge. That is *"How to predict tokens that will be added in the revisions?"* `RevToken` can identify tokens that will be revised from the tokens that exist in the line. In other words, `RevToken` selects the tokens that will be revised. Hence, if developers need to add some tokens in the line, `RevToken` cannot suggest them. To improve the performance, we need to develop a new approach for this purpose. One of the possible approaches is to use Masked Language Modeling (MLM) that can predict hidden tokens in sentences. In our future work, we are planning to predict such hidden tokens, using CodeBERT or other generative models.

VI. Related work

In recent years, many approaches have been proposed to mitigate the effort for code reviewing.

Recommending code to be revised. Many studies proposed approaches to guide developers to the code that needs to be revised. Olewicki *et al.* [18] revealed that file re-ordering tools for reviewing improve the review process. Fan *et al.* [19] have proposed a method to predict which code changes will be merged after code reviews. Kamei *et al.* [20] proposed an approach to identify the commits that need to be reviewed, based on defect-proneness. Hellendoorn *et al.* [8] proposed a method to predict hunks that need review comments, using a Transformer model. Hong *et al.* [7] proposed `REVSPOT` to recommend which lines of code or files need review comments, using Random Forest with LIME [13]. While these studies proposed approaches to identify the code to be improved, the granularities of the suggested code are commits, hunks, and lines, which are more coarse-grained.

Generating revised code: In recent years, a lot of studies have employed generative models for code review. Tufano *et al.* [21] proposed a method to translate submitted code to reviewed code, employing a Neural Machine Translation (NMT) model. Patanamon *et al.* [22] also automate code review activities, using a Byte-Pair Encoding and a Transformer-based NMT architecture. In addition, Lin *et al.* [23] employed GraphCodeBERT [24] and CodeT5 [25] to generate the revised code, taking into account the structure of source code. Watanabe *et al.* [26] investigated the use of ChatGPT for code review and found that many developers use ChatGPT to outsource the reviewing tasks (*i.e.,* generating code). However, the generative tasks often receive negative reactions because the generated code does not bring extra benefits. Although these studies directly eliminate the reviewing tasks instead of reviewers, our approach aims at accelerating reviewers' inspections.

VII. Conclusion

This study proposed a token level approach to recommend where developers review, which is finer-grained than the previous studies. Specifically, we fine-tuned CodeBERT and utilized either the Attention function or LIME approach to return the tokens that are most likely to be revised. Our empirical evaluation using the OpenStack and QtBase dataset demonstrated that the proposed approach outperforms the state-of-the-art model to predict lines to be revised. Also, we found that 86% of the tokens that will be revised in the next revision are accurately identified when the lines to be revised are correctly identified.

Acknowledgment

We gratefully acknowledge the financial support of JSPS for the KAKENHI grants (JP21H03416, JP24K02921, JP24K02923), Bilateral Program grant JPJSBP120239929, and JST for the PRESTO grant JPMJPR22P3.

REFERENCES

[1] A. Bosu, J. C. Carver, C. Bird, J. Orbeck, and C. Chockley, "Process aspects and social dynamics of contemporary code review: Insights from open source development and industrial practice at microsoft," *IEEE Transactions on Software Engineering*, vol. 43, no. 1, pp. 56–75, 2017.

[2] M. Bernhart and T. Grechenig, "On the understanding of programs with continuous code reviews," in *Proceedings of the 21st International Conference on Program Comprehension (ICPC)*, 2013, pp. 192–198.

[3] T. Baum, O. Liskin, K. Niklas, and K. Schneider, "Factors influencing code review processes in industry," in *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2016, pp. 85–96.

[4] A. Bacchelli and C. Bird, "Expectations, outcomes, and challenges of modern code review," in *Proceedings of 35th International Conference on Software Engineering (ICSE)*, 2013, pp. 712–721.

[5] L. MacLeod, M. Greiler, M.-A. Storey, C. Bird, and J. Czerwonka, "Code reviewing in the trenches: Challenges and best practices," *IEEE Software*, vol. 35, no. 4, pp. 34–42, 2018.

[6] G. Bavota and B. Russo, "Four eyes are better than two: On the impact of code reviews on software quality," in *Proceedings of the 2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2015, pp. 81–90.

[7] Y. Hong, C. Tantithamthavorn, and P. Thongtanunam, "Where should i look at? recommending lines that reviewers should pay attention to," in *Proceedings of 2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2022, pp. 1034–1045.

[8] V. J. Hellendoorn, J. Tsay, M. Mukherjee, and M. Hirzel, "Towards automating code review at scale," in *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (FSE)*, 2021, pp. 1479–1482.

[9] S. Arshad, S. Abid, and S. Shamail, "Codebert for code clone detection: A replication study," in *Proceedings of 2022 IEEE 16th International Workshop on Software Clones (IWSC)*, 2022, pp. 39–45.

[10] X. Zhou, D. Han, and D. Lo, "Assessing generalizability of codebert," in *Proceedings of 2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2021, pp. 425–436.

[11] E. Mashhadi and H. Hemmati, "Applying codebert for automated program repair of java simple bugs," in *Proceedings of the 18th IEEE/ACM International Conference on Mining Software Repositories (MSR)*, 2021, pp. 505–509.

[12] Y. Cai, A. Yadavally, A. Mishra, G. Montejo, and T. N. Nguyen, "Programming assistant for exception handling with codebert," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (ICSE)*, 2024, pp. 1146–1158.

[13] M. T. Ribeiro, S. Singh, and C. Guestrin, ""why should i trust you?": Explaining the predictions of any classifier," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, 2016, pp. 1135–1144.

[14] K. Hamasaki, R. G. Kula, N. Yoshida, A. E. C. Cruz, K. Fujiwara, and H. Iida, "Who does what during a code review? datasets of oss peer review repositories," in *Proceedings of the 10th Working Conference on Mining Software Repositories (MSR)*, 2013, pp. 49–52.

[15] J. Yu, L. Fu, P. Liang, A. Tahir, and M. Shahin, "Security defect detection via code review: A study of the openstack and qt communities," in *Proceedings of the 2023 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2023, pp. 1–12.

[16] C. Hannebauer, M. Patalas, S. Stünkelt, and V. Gruhn, "Automatically recommending code reviewers based on their expertise: An empirical comparison," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2016, pp. 99–110.

[17] M. J. Decker, M. L. Collard, L. G. Volkert, and J. I. Maletic, "srcdiff: A syntactic differencing approach to improve the understandability of deltas," *Journal of Software: Evolution and Process*, vol. 32, no. 4, p. e2226, 2020.

[18] D. Olewicki, S. Habchi, and B. Adams, "An empirical study on code review activity prediction and its impact in practice," *Proceedings of the ACM International Conference on the Foundations of Software Engineering (FSE)*, vol. 1, pp. 2238–2260, 2024.

[19] Y. Fan, X. Xia, D. Lo, and S. Li, "Early prediction of merged code changes to prioritize reviewing tasks," *Empirical Software Engineering*, vol. 23, 12 2018.

[20] Y. Kamei, E. Shihab, B. Adams, A. E. Hassan, A. Mockus, A. Sinha, and N. Ubayashi, "A large-scale empirical study of just-in-time quality assurance," *IEEE Transactions on Software Engineering*, vol. 39, no. 6, pp. 757–773, 2013.

[21] R. Tufano, L. Pascarella, M. Tufano, D. Poshyvanyk, and G. Bavota, "Towards automating code review activities," in *Proceedings of the 43rd IEEE/ACM International Conference on Software Engineering (ICSE)*, 2021, pp. 163–174.

[22] P. Thongtanunam, C. Pornprasit, and C. Tantithamthavorn, "Autotransform: Automated code transformation to support modern code review process," in *Proceedings of the 44th IEEE/ACM International Conference on Software Engineering (ICSE)*, 2022, pp. 237–248.

[23] H. Y. Lin and P. Thongtanunam, "Towards automated code reviews: Does learning code structure help?" in *Proceedings of the 30th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2023, pp. 703–707.

[24] 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. Clement, D. Drain, N. Sundaresan, J. Yin, D. Jiang, and M. Zhou, "Graphcodebert: Pre-training code representations with data flow," arXiv:2009.08366v4, 2021.

[25] Y. Wang, W. Wang, S. Joty, and S. C. H. Hoi, "Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation," arXiv:2109.00859, 2021.

[26] M. Watanabe, Y. Kashiwa, B. Lin, T. Hirao, K. Yamaguchi, and H. Iida, "On the use of chatgpt for code review: Do developers like reviews by chatgpt?" in *Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering (EASE)*, 2024, pp. 375–380.