Published version: Hiroto Sugimori, Shinpei Hayashi, "Revisiting Method-Level Change Prediction: Comparative Evaluation at Different Granularities,", in Proceedings of the 32nd IEEE International Conference on Software Analysis, Evolution and Reengineering, pp. 604–614, 2025, https://doi.org/10.1109/SANER64311.2025.00063

Revisiting Method-Level Change Prediction: A Comparative Evaluation at Different Granularities

Hiroto Sugimori
School of Computing
Institute of Science Tokyo
Meguro-ku, Tokyo 152–8550, Japan
sugimori@se.comp.isct.ac.jp

Shinpei Hayashi
School of Computing
Institute of Science Tokyo
Meguro-ku, Tokyo 152–8550, Japan
hayashi@comp.isct.ac.jp

***Abstract*—To improve the efficiency of software maintenance, change prediction techniques have been proposed to predict frequently changing modules. Whereas existing techniques focus primarily on class-level prediction, method-level prediction allows for more direct identification of change locations. Method-level prediction can be useful, but it may also negatively affect prediction performance, leading to a trade-off. This makes it unclear which level of granularity users should select for their predictions. In this paper, we evaluated the performance of method-level change prediction compared with that of class-level prediction from three perspectives: direct comparison, method-level comparison, and maintenance effort-aware comparison. The results from 15 open source projects show that, although method-level prediction exhibited lower performance than class-level prediction in the direct comparison, method-level prediction outperformed class-level prediction when both were evaluated at method-level, leading to a median difference of 0.26 in accuracy. Furthermore, effort-aware comparison shows that method-level prediction performed significantly better when the acceptable maintenance effort is little.**

***Index Terms*—change prediction, software maintenance, machine learning**

I. Introduction

Software continuously evolves even after its release. The reasons for changes include changing requirements, refactoring, and bug fixes [1]. These changes are essential for improving the software quality. However, numerous changes can lead to increased software complexity. Therefore, developers need to implement changes while preventing software complexity.

Studies on software change prediction are being conducted to support software change activities [2]–[11]. The goal of this task is to improve the software quality by predicting change-prone modules that are prone to frequent changes in the next release. By identifying these change-prone modules, developers can refactor the relevant source code and allocate human resources according to the anticipated amount of change. Therefore, developers can implement changes smoothly while preventing the software from becoming more complex. In addition, the developer's task context [12] can be inferred based on the change prediction results. To reduce information overload in development, a development environment[1] has been proposed to display only the modules that are relevant to the tasks on which the developer is working, based on the development context [12], [13]. If change-prone modules can be accurately identified, developers can concentrate only on the modules relevant to their tasks. Additionally, based on this inferred context, they can also plan refactoring that will contribute to future code changes in development tasks [14], [15].

Existing change prediction approaches focus primarily on the class-level prediction [3]–[8], [10], [11]. However, the size of a class can range from hundreds to thousands of lines. Therefore, it is unclear where exactly changes will occur within a class that is predicted to be change-prone.

Method-level prediction, which offers a finer granularity, enables developers to identify where changes will occur more directly [9]. The usefulness of method-level prediction can be clarified when a developer performs method-level refactoring to facilitate future changes based on class-level predictions. The developer refactors several methods that belong to a class predicted to be change-prone. If changes occur in other methods within that class that were not refactored, the refactoring effort of the developer will prove ineffective. Accurate method-level prediction, i.e., yielding results with fewer noisy methods, allows us to refactor the methods where changes actually occur. Method-level prediction enables a more direct identification of change locations; however, it may also lead to a reduced prediction performance. A trade-off may exist between the granularity of predictions and their performance, leaving developers uncertain about which level to choose in practice.

This paper aims to clarify the effectiveness of method-level change prediction by comparing the results at class-level and method-level. We designed our prediction technique based on the method-level prediction study by Farah et al. [9]. To investigate the usefulness of method-level change prediction, we derived the following research questions (RQs):

- ***RQ*$_1$**: *How does performance vary between class-level prediction and method-level prediction when evaluated at their respective levels?*
- ***RQ*$_2$**: *How does performance vary between class-level prediction and method-level prediction when evaluated at method-level?*

[1]https://eclipse.dev/mylyn

TABLE I
COMPARISON OF RELATED WORK

	Class-level	Method-level	Comparison
Bug prediction	✓ [16], [17]	✓ [18], [19]	✓ [18]
Bug localization	✓ [20], [21]	✓ [22]–[25]	✓ [23]
Change prediction	✓ [2]–[11]	✓ [9]	No study.

- ***RQ***$_3$: *How does performance vary between class-level prediction and method-level prediction in effort-aware evaluation?*

The motivation for each RQ is explained in detail later. By answering these RQs, we clarify the usefulness of method-level change prediction.

The main contributions of this paper are shown below.

- We reproduced the method-level change prediction technique proposed by Farah et al.
- We re-evaluated method-level change prediction by comparing it with class-level prediction from three perspectives: direct, method-level, and effort-aware comparisons.
- We provided guidelines for the context in which method-level prediction methods should be used based on the comparison results.

The investigation showed that when class-level and method-level predictions were directly evaluated at their respective levels, method-level prediction showed lower accuracy. However, when both were evaluated at method-level, method-level prediction outperformed class-level prediction across multiple evaluation metrics. In addition, effort-aware comparison showed that method-level prediction performed significantly better when the acceptable maintenance effort was little.

The remainder of this paper is organized as follows. Section II provides an overview of existing studies related to this work. Section III presents the details of our change prediction technique. Section IV reports the study design and results of the empirical studies, and Section V provides insights from the experiments. Section VI examine the threats to the validity of this paper. Finally, we conclude this study and state our future work in Section VII.

II. RELATED WORK

Various change prediction techniques have been proposed to date [2]–[11]. Most of these techniques employ machine learning models and are formulated as a classification problem. They utilize product and process metrics as independent variables, with a binary label that indicates whether a module is change-prone, serving as the dependent variable.

In the related tasks, such as bug prediction and bug localization, researchers have developed class-level and method-level prediction techniques [16]–[25] and demonstrated effectiveness of method-level predictions by comparing them with class-level predictions [18], [23]. Table I presents examples of studies that have proposed techniques at each level, as well as studies that compare both levels. Researchers have used the effort-aware evaluation to compare the two levels. Although method-level prediction techniques have also been proposed for change prediction, comparisons with class-level predictions have yet to be conducted.

A. Class-Level Change Prediction

Lu et al. demonstrated the effectiveness of product metrics in change prediction through an empirical study involving 102 projects [8]. Elish et al. derived process metrics to be used as inputs for change prediction using the Goal-Question-Metric approach and demonstrated their effectiveness through empirical experiments [6]. Furthermore, they revealed that predictions using both product and process metrics showed higher performance than those using only one type of metric. Catolino et al. proposed a prediction technique that incorporates developer-related information, which is commonly used in bug prediction [7]. They constructed models using product metrics, process metrics, and developer information as independent variables and compared the outcomes. The model incorporating developer information performed better than the other models. They also found that these metrics complement one another in terms of prediction performance. In addition, they demonstrated that utilizing ensemble methods in change prediction markedly improves the prediction performance [5]. Furthermore, they added code smell-related information as the independent variable to improve change prediction performance [10].

B. Method-Level Bug Prediction

In a similar task of bug prediction, several method-level prediction techniques have already been proposed, and their usefulness has been demonstrated. Giger et al. proposed a method-level bug prediction technique using product metrics and process metrics [19]. They employed product metrics that are definable at method-level in CK [26], a product metric computation tool. In addition, they used ChangeDistiller [27] to gather method-level change histories to calculate the process metrics. Hata et al. proposed a method-level bug prediction technique that uses process metrics [18]. They utilized a fine-grained repository [28] to calculate method-level process metrics. A fine-grained repository generated by their technique [28] tracks software change histories with finer granularity. They employed an evaluation approach that accounts for an effort to compare class-level and method-level predictions. The findings indicated that method-level prediction is more efficient than class-level prediction in identifying bugs.

C. Method-Level Change Prediction

Similar to bug prediction, a method-level prediction technique has been proposed for change prediction. Farah et al. extended class-level change prediction technique using product and process metrics to method-level [9]. Like Giger et al., they used ChangeDistiller [27] to collect method-level process metrics. In addition, they analyzed the metrics and machine learning models that are effective for change prediction. The results of their empirical experiments demonstrated that prediction models, which incorporate both types of metrics, perform better than those that rely on a single type of metric.

TABLE II
INDEPENDENT VARIABLES

	Granularity	Product Metrics	Process Metrics
Farah et al.	Method-Level	CK [26] and Understand: 83 types	Elish et al. [6]: 17 types
Our technique	Class-Level	CK [26]: 49 types	Elish et al. [6]: 17 types
	Method-Level	CK [26]: 31 types	Elish et al. [6]: 17 types

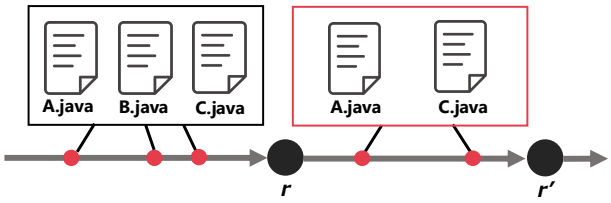


Fig. 1. Changes between releases.

To the best of our knowledge, this is the only paper that proposes a method-level change prediction technique.

In the above studies, class-level and method-level change prediction have not been compared, and the effectiveness of method-level change prediction has not been sufficiently demonstrated. In this paper, we replicated the technique of Farah et al. as closely as possible and conducted prediction experiments at both class-level and method-level. We evaluated the usefulness of method-level change prediction by comparing the results.

III. TECHNIQUE

In this paper, we conduct experiments at both class-level and method-level based on the change prediction technique proposed by Farah et al. [9]. This technique was selected for two reasons. First, to the best of our knowledge, their technique is the only one that addresses method-level change prediction. Second, their machine learning approach, which uses product and process metrics, aligns with the basic framework of existing class-level prediction techniques [2].

The software release scenario assumed in this paper is illustrated in Fig. 1. In this scenario, we consider a project with multiple releases, where several changes occur in each release. Let r represent the release and r' be the next release of r. Our technique uses metrics that can be collected from the information prior to r to predict the modules that will change between r and r'.

An overview of the prediction technique is shown in Fig. 2. The repository of the target project is analyzed to calculate metric values for each module. A dataset containing metric values $x_1, x_2, \ldots, x_m$ for the project modules serves as the independent variables alongside a binary (0/1) label that identifies whether or not a module is change-prone, which acts as the dependent variable. This setup is used to learn the relationship between the independent and dependent variables. When the metric values of a specific target module are input into the trained model, it predicts whether the module is likely to be change-prone.

In the remainder of this section, we first describe the technique proposed by Farah et al. and then explain our technique and how it differs from theirs.

A. Technique by Farah et al.

Farah et al. investigated the most effective techniques for method-level change prediction by applying various machine learning models and data resampling techniques and comparing their performance. In this subsection, we briefly present the techniques they compared and their results.

1) Independent Variables: Farah et al. selected product and process metrics as independent variables for prediction based on their literature review. They compared the results of using each metric individually with those of combining both metrics. The number of types of metrics they used is listed in Table II. The comparison showed that the use of both product and process metrics led to the highest prediction performance. They used CK [26] and Understand[2] to extract product metrics. CK and Understand are static analysis tools used for calculating product metrics. For extracting process metrics, they used a tool they developed themselves.

2) Dependent Variable: They defined modules with a number of changes greater than the median number of changes across all methods between releases as change-prone. This definition was provided by Romano et al. [29]. They used ChangeDistiller [27] to extract method-level changes between releases. This tool implements a diff algorithm that generates and compares syntax trees between two versions of a project.

3) Data Preprocessing: Before creating the machine learning model, they performed data preprocessing. The values of the metrics in a numeric format were normalized between 0 and 1 using the min-max technique. They found imbalances in the distribution of dependent variables in the data. As data imbalances can adversely affect learning, they utilized the following six data resampling techniques to mitigate this issue: Synthetic Minority Oversampling Technique (SMOTE) [30], Random Over-Sampler (ROS), Adaptive Synthetic (ADA), Random Under-Sampling (RUS), Edited Nearest Neighbors (ENN), and TomekLinks (TL). As a result of comparing their performance, ROS and RUS exhibited the best performance. They used scikit-learn [31] to implement these resampling techniques.

4) Building Classifier: They compared the results using Decision Tree, Logistic Regression, Multi-Layer Perceptron, and Random Forest. These models have been widely used in existing research [2]. Among the aforementioned data

[2]https://scitools.com/

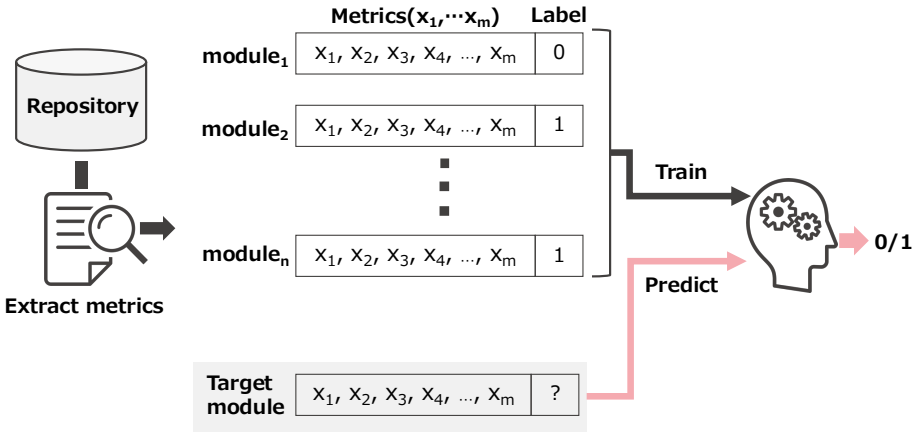


Fig. 2. Overview of prediction technique.

resampling techniques, the best result was achieved when combined with Random Forest and RUS.

B. Our Technique

Based on the technique proposed by Farah et al., we designed prediction techniques at both the method-level and class-level. Although they have published part of the program necessary for the experiment, the artifact was not designed to execute the whole process. Therefore, we re-implemented the entire program to conduct experiments. The purpose of this paper is not to reproduce their technique perfectly but to compare results between different levels. Therefore, unlike their approach, we do not compare various prediction techniques but instead adopt the technique that demonstrated the best performance in their experiments.

1) Independent Variables: We employed product and process metrics as independent variables because Farah et al. demonstrated that using both metrics together yields better results than using either one alone. The number of types of metrics we used for class-level prediction and method-level prediction is also shown in Table II.

Product metrics are calculated based on the source code at release r. We used CK [26] to collect product metrics at both class-level and method-level. Although Farah et al. computed more metrics with Understand, our approach relies solely on CK because of its open-source nature and lower implementation cost. Since Understand is commercial software, it is not freely accessible to everyone.

Process metrics are calculated based on the commit history prior to r. We used the same types of metrics set as Farah et al. This set of process metrics was originally proposed by Elish et al. [6]. We developed our own program to compute these metrics because the implementation used by Farah et al. for measuring the process metrics was not publicly available. We used a method repository [28] that allows simpler retrieval of method-level changes. Method repository extracts code segments corresponding to the methods from each file in the repository and saves them as new files that carry over the method-level change history. This allows the acquisition of method-level change histories, similar to how file-level change histories are captured using Git. We used git-stein [32] to generate method repositories.

2) Dependent Variable: As to Farah et al., modules with a number of changes greater than the median number of changes across all modules between releases are defined as change-prone. While Farah et al. used ChangeDistiller to obtain method-level change histories, we did not adopt this approach in this paper. Instead of ChangeDistiller, we used a method repository [28] again here. Differences in the tools utilized may lead to discrepancies in tracking renamed modules, although these are not substantial concerns in the context of machine learning-based techniques.

3) Data Preprocessing: Based on the research by Farah et al., we preprocessed the data prior to training a machine learning model. The values of the metrics in a numeric format were normalized between 0 and 1 using the min-max technique. We employed RUS as a data resampling technique. According to Farah et al., RUS is the most effective data resampling technique when paired with the best machine learning model for change prediction, Random Forest.

4) Classifier Building: We employed Random Forest as our machine learning model because Farah et al. demonstrated that it is the most effective model for method-level prediction among traditional machine learning models.

IV. Empirical Study

We conducted experiments to answer the three research questions.

A. Dataset Preparation

In addition to the repositories selected by Farah et al., we used those selected by Catolino et al. [10] to confirm the applicability of our technique to other projects. Catolino et al. have proposed a class-level change prediction technique using machine learning, similar to the technique proposed by Farah et al.

The repositories used and their main characteristics are shown in Table III. These repositories are open-source software projects developed in Java and hosted on GitHub. In

TABLE III
DATASET OVERVIEW

Original paper	Repositories	# releases	# commits	# classes	# methods	CP class	CP method
Farah et al. [9]	apache/commons-bcel	3	29–52	416–418	3,652–3,701	0.024–0.20	0.022–0.11
	apache/pdfbox	10	43–6,180	434–1,159	3,360–10,081	0.09–0.46	0.03–0.50
	apache/commons-io	15	34–1,022	39–242	332–2,152	0.13–0.49	0.03–0.47
	easymock/easymock	10	36–219	88–102	522–574	0.02–0.42	0.004–0.27
	wro4j/wro4j	8	15–1,237	138–353	685–1,847	0.03–0.47	0.06–0.49
	igniterealtime/openfire	6	154–913	686–726	6,612–8,346	0.15–0.43	0.04–0.16
	(Total)	52	15–6,180	39–1,159	332–10,081	0.02–0.49	0.03–0.50
Catolino et al. [10]	apache/logging-log4j1	2	226–365	86–87	455–518	0.43–0.45	0.41–0.42
	apache/ant	3	1,885–3,556	281–573	1,992–5,436	0.43–0.5	0.20–0.31
	apache/ant-ivy	1	184	352	3,762	0.24	0.03
	apache/camel	3	498–1,142	153–447	876–3,144	0.31–0.39	0.14–0.45
	apache/forrest	1	874	29	165	0.38	0.17
	apache/lucene	2	533–599	185–226	1,658–2,136	0.38–0.50	0.26–0.3
	apache/poi	3	135–843	222–378	2,521–4,570	0.31–0.49	0.1–0.36
	apache/synapse	2	325–435	152–219	524–863	0.40–0.45	0.14–0.22
	apache/xerces2-j	2	239–453	262–172	2,164–2,289	0.36–0.48	0.12–0.14
	(Total)	19	84–3,556	29–573	165–5,436	0.24–0.50	0.03–0.45
Total		71	15–6,180	29–1,159	165–10,081	0.02–0.50	0.03–0.50

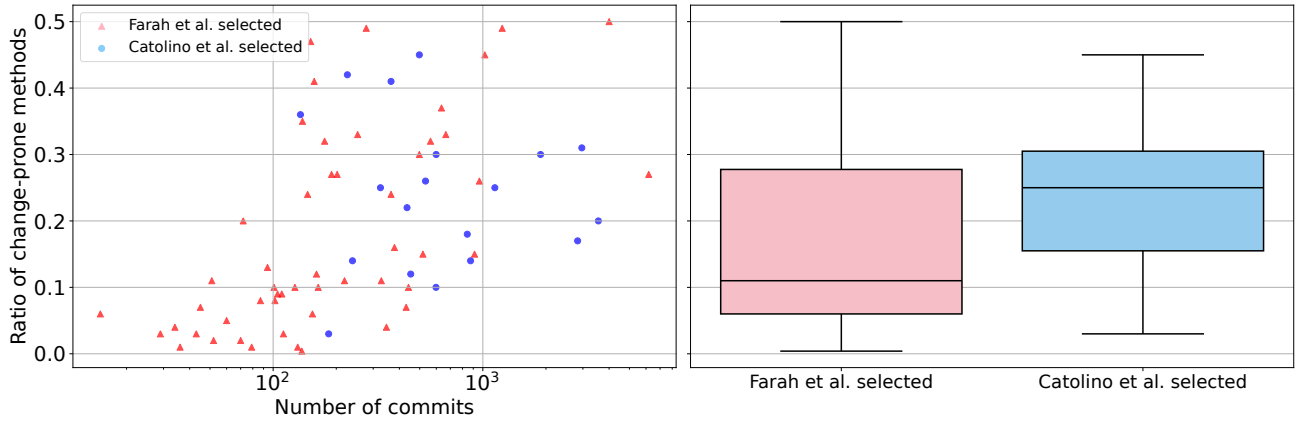


Fig. 3. Difference in the distribution of change-prone methods between the datasets by Farah et al. and Catolino et al.

Table III, the "# releases" column represents the number of targeted releases in the repository. The "# commits" column shows the range of commit counts for these releases. The "# classes" column represents the range of a number of classes, and the "# methods" column reflects the range of a number of methods. The "CP class" and "CP method" columns represent the ranges of the ratio of change-prone classes and change-prone methods, respectively. Releases were identified using tags in the Git repository, and the commit history between tags was retrieved. We found that the distribution trends of change-prone methods in the datasets used by Farah et al. and Catolino et al. differed. We believe that this difference is owing to the number of commits, which represents the scale of the releases. The scatter plot on the left in Fig. 3 shows the relationship between the proportion of change-prone methods and the number of commits for each release. The box plot on the right shows the distribution of the proportion of change-prone methods for each release. When the ratio of methods being changed is small, the median number of changes across all methods becomes zero. In this case, as already mentioned in Section III, any method that is modified even once is considered change-prone; however, if this ratio is small, the percentage of change-prone methods is also low.

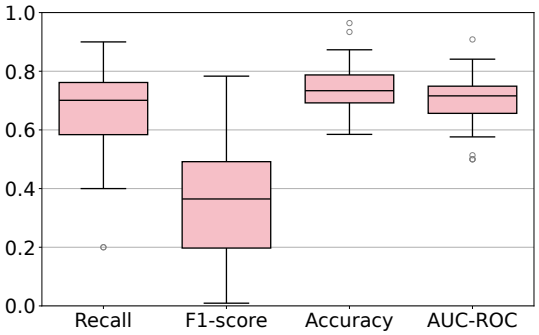
B. Evaluation Metrics

We employed evaluation metrics that are commonly used in change prediction, including the research by Farah et al. First, we computed precision, recall, and accuracy defined as follows:

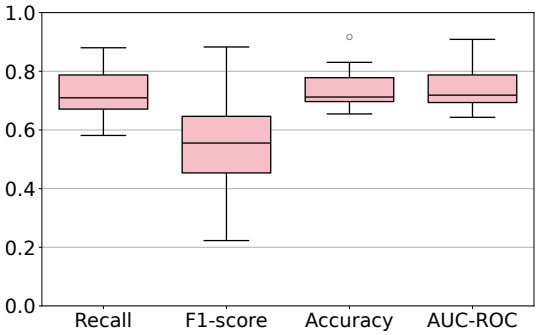
$$Precision = \frac{TP}{TP + FP},$$
$$Recall = \frac{TP}{TP + FN},$$
$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN}.$$

where, TP is the number of true positives, TN is the number of true negatives, FP is the number of false positives, and FN is the number of false negatives. F1-score is defined as the harmonic mean of precision and recall:

$$\text{F1-score} = 2 * \frac{Precision * Recall}{Precision + Recall}.$$



(a) Replication on repositories used by Farah et al.



(b) Replication on repositories used by Catolino et al.

Fig. 4. Results of replication experiments.

Moreover, we employed the Area Under the ROC Curve (AUC-ROC).

C. Replication

1) Motivation: As explained in Section III-B, our technique is not a complete reproduction of that proposed by Farah et al. [9]. We conducted replication experiments to verify whether the prediction results from our technique did not differ substantially from those of Farah et al.

2) Study Design: We conducted experiments on method-level change prediction using the technique described in Section III. The datasets were divided into those used by Farah et al. (upper half of Table III) and those used by Catolino et al. (lower half of Table III). We then compared these results described in the paper by Farah et al. This confirms that our technique does not substantially diverge from the approach by Farah et al. and can be replicated with other datasets. For evaluation, we use the metrics explained in Section IV-B, measured at method-level. However, since Farah et al. did not use precision for their evaluation, we calculated the values for the other evaluation metrics.

3) Results and Discussion: The values of each evaluation metric from the experiments conducted on the dataset by Farah et al. are depicted in the box plots in Fig. 4a. These results were compared with those reported by Farah et al. They have published project-level results in their replication package rather than providing results for all releases of the target projects. Therefore, we calculated our project-level results and compared them with theirs. According to Farah et al., in comparing the performance of each model, we used the AUC-ROC as the main evaluation metric. Among the repositories used by Farah et al., the largest AUC-ROC difference was 0.16 in the `igniterealtime/opefire` project, indicating that the values were close. However, there was no substantial difference in the overall trend, and we believe that the variations are within the range attributable to the different sets of metrics used in our prediction compared with theirs. The values of each evaluation metric in the experiments conducted on the dataset selected by Catolino et al. are also depicted in box plots in Fig. 4b. Comparing these results with those of our experiments on the dataset selected by Farah et al., the trends in the metrics were similar; however, the F1-score increased slightly. As mentioned in Section IV-A, the dataset selected by Farah et al. has a smaller ratio of change-prone methods than that selected by Catolino et al., resulting in greater data imbalance. We believe that this data imbalance degraded the performance of prediction on the dataset selected by Farah et al. Based on these results, we believe that our technique is applicable across different datasets.

> Our technique differs from that proposed by Farah et al. in terms of the employed metrics, leading to different outcomes. Among the repositories used by Farah et al., the largest AUC-ROC difference was 0.16. Nevertheless, as the objective of our study is to compare different granularities, this issue does not detract from the subsequent discussion.

D. RQ_1: How does performance vary between class-level prediction and method-level prediction when evaluated at their respective levels?

1) Motivation: Method-level prediction provides the location of changes more directly than class-level prediction. However, there may be differences in performance due to the number of modules targeted for prediction and the metrics used for prediction varying between class-level and method-level predictions. There could be a trade-off between the granularity of the predictions and their performance, leaving developers uncertain about which level to select in practice. Therefore, we compare the performance of method-level prediction with class-level prediction.

2) Study Design: Following the technique described in Section III, we performed predictions at both class-level and method-level and compared their results. As in Section IV-C, the average value of each evaluation metric was calculated using 10-fold cross-validation. We compared the results across the two levels using a two-sided Wilcoxon signed-rank test with a significance level of $\alpha = 0.05$. We also reported Cliff's delta (d) to measure the magnitude of the performance difference. Cliff's delta is interpreted based on the threshold proposed by Romano et al. [33]: *negligible* for $|d| < 0.147$, *small* for $0.147 \leq |d| < 0.33$, *medium* for $0.33 \leq |d| < 0.474$, and *large* for $0.474 \leq |d|$.

We provide the definitions of *TP*, *TN*, *FP*, and *FN* for class-level and method-level. Let the sets of all classes and

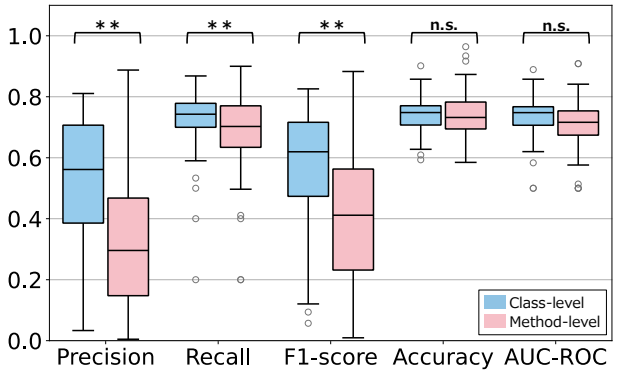


Fig. 5. Results for RQ_1: direct comparison.

methods in the target project be $E_c = \{\ldots, e_c, \ldots\}$ and $E_m = \{\ldots, e_m, \ldots\}$, respectively. In class-level prediction, we define the ground truth of change-prone classes $G_c \subseteq E_c$ and $\overline{G_c} = E_c \setminus G_c$ as sets of classes that should be predicted to be change-prone and non-change-prone, respectively. In the same way, in method-level prediction, we define the ground truth of change-prone methods $G_m \subseteq E_m$ and non-change-prone methods $\overline{G_m} = E_m \setminus G_m$.

Let $P_c \subseteq E_c$ and $P_m \subseteq E_m$ be the sets of the classes and methods predicted as change-prone, and $\overline{P_c} = E_c \setminus P_c$ and $\overline{P_m} = E_m \setminus P_m$ be those predicted as non-change-prone, respectively. *TP*, *TN*, *FP*, and *FN* at each level can be defined as follows:

$$TP_c = |P_c \cap G_c|, \qquad TP_m = |P_m \cap G_m|,$$
$$TN_c = |\overline{P_c} \cap \overline{G_c}|, \qquad TN_m = |\overline{P_m} \cap \overline{G_m}|,$$
$$FP_c = |P_c \cap \overline{G_c}|, \qquad FP_m = |P_m \cap \overline{G_m}|,$$
$$FN_c = |\overline{P_c} \cap G_c|, \qquad FN_m = |\overline{P_m} \cap G_m|.$$

3) Results and Discussion: Figure 5 shows the values for each evaluation metric at method-level and class-level, respectively. Table IV presents the median value of each metric at both levels, the p-value from the Wilcoxon signed-rank test, as well as the values and results of Cliff's delta. In Fig. 5 and Table IV, the statistical significance is denoted as follows: $p < 0.05$ is marked with $*$, $p < 0.01$ with $**$, and non-significant results are indicated by n.s. This notation is used consistently in subsequent figures and tables throughout this paper. The results suggested that class-level predictions outperformed method-level predictions in performance, and their differences were statistically significant for all metrics except for accuracy and AUC-ROC. The median F1-score for class-level prediction was 0.61, while for method-level prediction, it was 0.2 points lower, at 0.41. The effect sizes for precision and F1-score were large, while that for recall was small.

> Method-level prediction demonstrated lower performance compared to class-level prediction. In terms of the F1-score, the median of the method-level prediction result was 0.20 lower than that of the class-level prediction results, and its effect size was large.

TABLE IV
RESULTS FOR RQ_1: DIRECT COMPARISON

	Class-level	Method-level	p-value	Cliff's delta
Precision	0.56	0.29	1.0×10^{-13}**	−0.47 (large)
Recall	0.74	0.70	7.5×10^{-4}**	−0.22 (small)
F1-score	0.61	0.41	5.2×10^{-13}**	−0.48 (large)
Accuracy	0.74	0.73	0.24 n.s.	−0.095 (negligible)
AUC-ROC	0.74	0.71	0.11 n.s.	−0.20 (small)

E. RQ_2: How does performance vary between class-level prediction and method-level prediction when evaluated at method-level?

1) Motivation: In a change-prone class, there may be methods that are not change-prone (and vice versa). The evaluation in answering RQ_1 ignores such methods. Therefore, we investigated the performance when considering class-level predictions at method-level.

2) Study Design: We re-evaluated the class-level predictions in RQ_1 based on method-level ground truth and compared them with the method-level prediction results in RQ_1. In order to assess class-level predictions at method-level, we define *TP*, *TN*, *FP*, and *FN* based on method-level ground truth.

We regard all the methods in all classes predicted as change-prone as change-prone methods:

$$P_{cm} (\subseteq E_m) = \bigcup_{c \in P_c} methods(c)$$

where $methods(c)$ denotes the set of methods belonging to the class c. Using the set of ground-truths, these numbers could be computed:

$$TP_{cm} = |P_{cm} \cap G_m|, \qquad TN_{cm} = |\overline{P_{cm}} \cap \overline{G_m}|,$$
$$FP_{cm} = |P_{cm} \cap \overline{G_m}|, \qquad FN_{cm} = |\overline{P_{cm}} \cap G_m|.$$

Here, $\overline{P_{cm}} = E_m \setminus P_{cm}$ represents the set of all methods included in the classes predicted as non-change-prone.

Using these values, we calculate precision, accuracy, recall, and F1-score for re-evaluating class-level predictions based on method-level ground truth. To determine whether there was a difference in the median values of each evaluation metric between class-level predictions and method-level predictions, we applied the two-sided Wilcoxon signed-rank test at a significance level of 0.05.

3) Results and Discussion: Figure 6 shows the values of each evaluation metric at method-level and class-level, respectively. Table V shows the median values of each metric at each level, the p-values from the Wilcoxon signed-rank test, and the values and effect sizes of Cliff's delta. These results exhibit a trend different from those of RQ_1. Method-level predictions significantly outperformed in terms of precision and accuracy with medium and large effect sizes, respectively. However, there was a decrease in the recall and F1-score

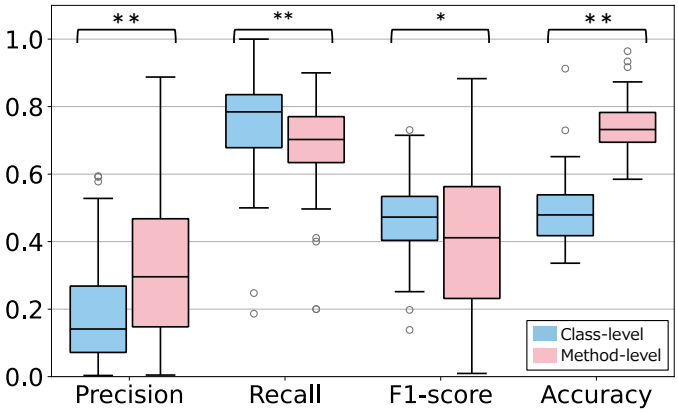


Fig. 6. Results for RQ_2: method-level evaluation.

TABLE V
RESULTS FOR RQ_2: METHOD-LEVEL EVALUATION

	Class-level	Method-level	p-value	Cliff's delta
Precision	0.14	0.29	8.3×10^{-7}**	0.36 (medium)
Recall	0.78	0.70	0.0020**	−0.33 (medium)
F1-score	0.47	0.41	0.013*	−0.2 (small)
Accuracy	0.47	0.73	2.3×10^{-17}**	0.94 (large)

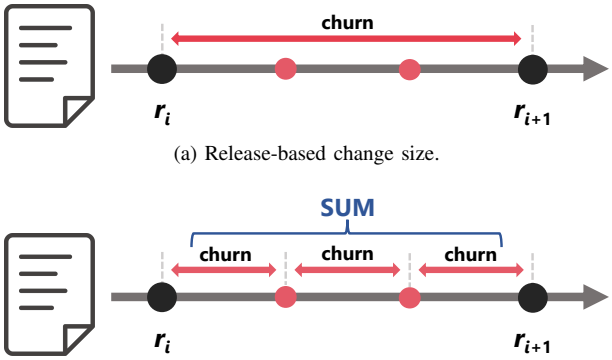
with a small effect size. These facts indicate that method-level predictions increase the number of false negative methods but reduce the number of false positive methods compared with class-level predictions.

> Method-level prediction performed significantly better in terms of accuracy and precision, and their effect size was large and medium, respectively. The difference in median accuracy was 0.26. However, in terms of recall and F1-score, class-level prediction performed better. These facts indicate that method-level prediction can reduce false positives; however, the false negatives are larger than those for class-level prediction.

F. RQ_3: How does performance vary between class-level prediction and method-level prediction in effort-aware evaluation?

1) Motivation: Developers aim to utilize their limited efforts as efficiently as possible when maintaining software. They often prioritize refactoring areas where future changes are anticipated over areas with severe code smells [15]. An effort-aware evaluation was employed to compare the effectiveness of class-level prediction and method-level prediction in supporting efficient maintenance activities.

2) Study Design: We introduced an effort-aware evaluation to change prediction. This evaluation technique allows for comparisons between different levels in terms of effort efficiency. We adopt the approach of evaluating effort using Lines of Code (LOC), which is a common metric in the field of bug prediction [18]. We assessed the effectiveness of class-level and method-level prediction using this approach when developers perform maintenance tasks on a given number of lines of source code based on change prediction results. Unlike existing evaluation approaches that consider prediction results as a binary indicator of whether or not the modules are change-prone, we employed a probability-based ranking. We then assess the usefulness of the ranking by considering how developers utilize the top-k lines of modules in the ranking. Developers prefer to refactor code snippets that are likely to be changed in the future rather than those with severe code smells [15], [34]. Therefore, we argue that predictions for modules with more changes are useful.



(a) Release-based change size.

(b) Commit-based change size.

Fig. 7. Change size metrics.

For the purpose of quantifying change size, we define two types of metrics: *release-based change size* and *commit-based change size*, to measure the change size between releases from different perspectives. Figure 7 illustrates their concepts. A possible way is to compute the size of the differences between two snapshots of the releases. *Release-based change size* represents the sum of the *code churn* [35], i.e., the total number of added and deleted lines, when comparing the beginning and ending snapshots of the release milestone for each module. However, the changes implemented between releases may overlap, and the same code location may be modified multiple times. Therefore, the difference between the snapshots may underestimate the repeated efforts to the same code location. To mitigate this issue, we also computed the *commit-based change size*, which uses the sum of the churn of the changes of each commit in the given release milestone.

More precisely, let C be a set of commits and E be a set of modules. First, we define a function $\mathrm{churn}(e, c, c')$ that represents the churn of a module $e \in E$ in a pair of commits $c, c' \in C$. Given two releases r and r', the sequence of n commits between them is denoted by $c_1, c_2, \ldots, c_n$, where $c_r = c_1$ and $c_{r'} = c_n$ are the commits corresponding to the releases r and r', respectively. *Release-based change size* and *commit-based change size* of module e are denoted by $\Delta_e^{\mathrm{release}}$ and $\Delta_e^{\mathrm{commit}}$, respectively, and computed as follows:

$$\Delta_e^{\mathrm{release}} = \mathrm{churn}(e, c_r, c_{r'}),$$

$$\Delta_e^{\mathrm{commit}} = \sum_{i=1}^{n-1} \mathrm{churn}(e, c_i, c_{i+1}).$$

TABLE VI
MEDIAN OF TOP-k RELEASE-BASED CHANGE RATIO AND TEST RESULTS

k	Class-level	Method-level	p-value	Cliff's delta
100	0.19	0.66	1.1×10^{-6}**	0.37 (medium)
500	0.24	0.60	1.1×10^{-6}**	0.28 (small)
1,000	0.26	0.46	1.3×10^{-5}**	0.22 (small)
5,000	0.23	0.30	0.0019**	0.07 (negligible)
10,000	0.20	0.23	0.45 n.s.	0.032 (negligible)

TABLE VII
MEDIAN OF TOP-k COMMIT-BASED CHANGE RATIO AND TEST RESULTS

k	Class-level	Method-level	p-value	Cliff's delta
100	0.25	0.94	7.7×10^{-5}**	0.34 (medium)
500	0.33	0.76	3.0×10^{-4}**	0.24 (small)
1,000	0.32	0.70	2.5×10^{-5}**	0.21 (small)
5,000	0.33	0.41	0.058 n.s.	0.064 (negligible)
10,000	0.29	0.31	0.62 n.s.	0.032 (negligible)

Assuming that the effort to maintain k lines of code is available, we measure these two metrics in the modules within the top-k LOC and then sum the values for each module.

Let $R = \langle e_1, e_2, \dots \rangle$ be the ranking of modules in the descending order of the change-proneness that the change-prediction model calculated. Therefore, the number of top results l can be the number that satisfies the following condition:

$$\sum_{i=1}^{l} \mathrm{LOC}(e_i) < k \leq \sum_{i=1}^{l+1} \mathrm{LOC}(e_i)$$

where $\mathrm{LOC}(e)$ counts the LOC of module e at release r. In our hypothetical situation, only the top-l ranking results are utilized because acceptable development effort is limited.

Using the top l items in the ranking $R_l = \langle e_1, e_2, \dots, e_l \rangle$, two change-size metrics can be computed follows:

$$\textit{top-k release-based change ratio} = \frac{\sum_{e \in R_l} \Delta_e^{\text{release}}}{\sum_{e \in R_l} \mathrm{LOC}(e)},$$

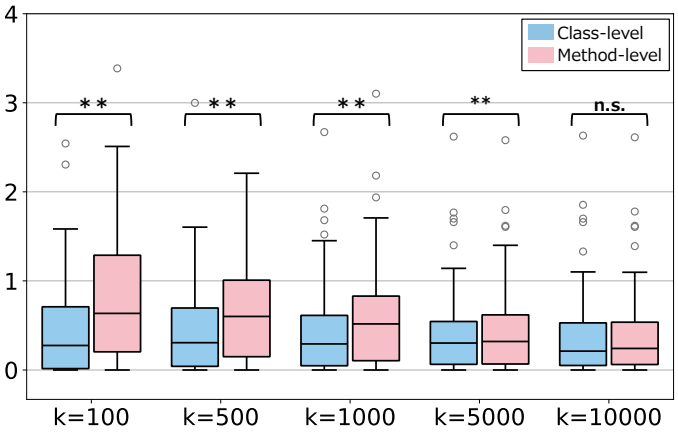
$$\textit{top-k commit-based change ratio} = \frac{\sum_{e \in R_l} \Delta_e^{\text{commit}}}{\sum_{e \in R_l} \mathrm{LOC}(e)}.$$

The assumption behind these metrics is that the larger these values are, the more useful the ranking of the top-k lines will be in a realistic context, where the acceptable development effort is limited. We use 100, 500, 1,000, 5,000, and 10,000 for the value of k for the evaluation to simulate different contexts in terms of the acceptable development effort.

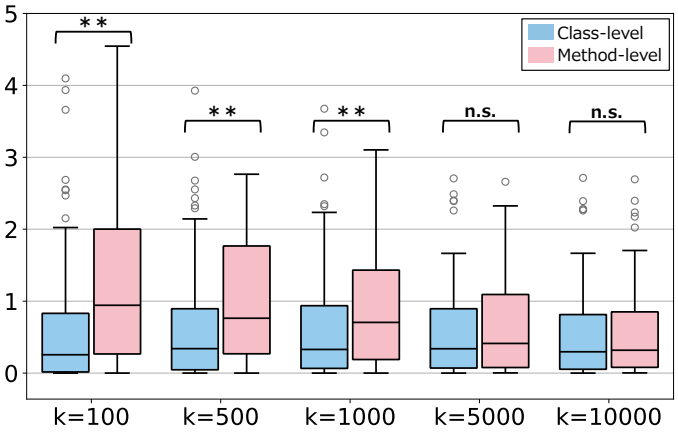
We applied the two-sided Wilcoxon signed-rank test at a significance level of 0.05 To determine whether there were differences in the median values of each evaluation metric.

3) Results and Discussion: Figure 8 depicts the values of top-k release-based and commit-based change ratios at both method-level and class-level, respectively. Tables VI and VII present the median values of each metric at each level, the p-values from the Wilcoxon signed-rank test, and the values and effect sizes of Cliff's delta.

In both change sizes, smaller values of k were associated with higher performance in method-level prediction. As k increased, the discrepancy between method-level and class-level predictions decreased. The most significant difference was observed for $k = 100$. Method-level prediction performed significantly better with medium effect size in top-k release-based change ratio and small in top-k commit-based change ratio. At $k = 10{,}000$, neither top-k release-based nor commit-based change ratios showed any significant differences, and its effect size was negligible. A possible interpretation for these results is that, as k increases, the modules included within k lines encompass lower-ranking positions, thereby diluting the association with predictive performance. These results indicate that method-level predictions become more effective when development resources are scarce.



(a) Top-k release-based change ratio.



(b) Top-k commit-based change ratio.

Fig. 8. Results for RQ_3: effort-aware evaluation.

The smaller the value of k, the larger the difference between the method-level prediction and class-level prediction. For $k = 100$, method-level prediction significantly outperformed class-level prediction in both top-k release-based and commit-based change ratios, resulting in a median difference of 0.47 and 0.69, respectively. When development resources are limited, the efficiency of maintenance activities can be enhanced by utilizing method-level prediction.

V. Discussion

Following the outcomes of RQ_1 to RQ_3, we discuss the effectiveness of method-level change prediction compared with class-level change prediction. Method-level change prediction is considered effective because it can predict the specific code snippets within the source code where changes occur more directly compared to class-level prediction. However, there was a concern that the performance of predictions might decrease as the prediction target shifts from the entire project's classes to methods. According to the results of RQ_1, as anticipated, method-level prediction performance was lower than that of class-level prediction.

However, the results of RQ_2 suggest that method-level prediction may reduce the detection of false positive methods compared with adapting class-level predictions to corresponding methods. This implies that when developers utilize change prediction results for method-level maintenance, it might be beneficial to focus maintenance efforts on areas where changes are most anticipated while accepting some oversights.

In RQ_3, we considered a realistic setting in which maintenance effort was limited. Our findings suggest that method-level change prediction enables more efficient maintenance when development resources are scarce.

Based on these findings, we conclude that method-level change prediction is more effective than class-level change prediction for prioritizing efficient use of limited maintenance resources rather than aiming for comprehensive maintenance. Method-level prediction can be used effectively when the developers want to maintain concentrated change locations while allowing some tolerance for missed change locations, or when development resources are limited and large-scale maintenance is not feasible.

VI. Threats to Validity

A. Internal Validity

In this paper, we used CK [26] to collect product metrics and implemented a tool to collect process metrics. Although CK has been widely used in other change prediction studies and there is no standard tool for calculating process metrics to the best of our knowledge, the results may vary depending on the implementation.

We used a method repository to collect the change history of methods. This approach enabled the acquisition of method-level change histories using Git. Farah et al. used ChangeDistiller for the same purpose. The code differences generated by ChangeDistiller ignore changes to whitespace and comments, whereas those generated by Git do not. Moreover, there are differences in the approach to detect renaming. In addition, ChangeDistiller detects renaming by comparing Abstract Syntax Tree (AST) nodes between different versions of the code, identifying instances where nodes with similar structures but different names indicate a rename [27]. In contrast, Historage detects renaming based on the similarity of file contents when no matching file name is found [28]. It compares the content and names of files across versions, detecting renaming through similarity between the original and modified files. Consequently, there are slight discrepancies in the changes considered by each approach. However, these facts do not detract from the comparison results between class-level prediction and method-level prediction, which is the focus of this paper.

B. Construct Validity

The types of independent variables used for change prediction are considered threats to construct validity. In this paper, product and process metrics were used as independent variables because the approach that employs these two types of metrics has shown the best performance in the paper by Farah et al. and is widely used in other studies [2]. However, other independent variables have been explored in class-level prediction [7], [10]. If we introduce these metrics to the independent variables, it might lead to different outcomes.

C. External Validity

We did not use multiple machine learning models, sets of metrics, or data resampling techniques; we selected the most effective ones, as identified by Farah et al. Employing different approaches from those presented in this study could yield different results. However, the primary goal of this paper was to compare class-level predictions with method-level predictions. These facts do not undermine our discussion from the perspective of comparisons between different prediction levels.

The projects targeted in this paper are all open-source software developed in Java. This poses a threat to the generalizability of the results of this study to other software programs.

VII. Conclusion and Future Work

In this paper, we conducted change prediction experiments at both class-level and method-level using existing change prediction techniques and compared the results. Regarding RQ_1, we found that method-level change prediction underperformed compared to class-level prediction. For RQ_2, we evaluated class-level change prediction at method-level and compared the results with that obtained from method-level change prediction. The findings indicated that, although method-level predictions resulted in a higher number of false negatives, they reduced the number of false positives. To address RQ_3, we introduced an effort-aware evaluation. The results showed that method-level prediction is more effective when maintenance resources are limited.

In future research, we are planning to analyze the impact of release size on predictions. The size of a release affects the proportion of change-prone modules. This may change the difficulty of making predictions.

All the datasets of the experiments conducted in this paper are publicly available [36].

Acknowledgments

This work was partly supported by JSPS Grants-in-Aid for Scientific Research (JP24H00692, JP23K24823, JP21H04877, JP21K18302, and JP21KK0179).

REFERENCES

[1] M. M. Lehman and L. A. Belady, *Program evolution: Processes of software change*. USA: Academic Press Professional, Inc., 1985.

[2] R. Malhotra and M. Khanna, "Software change prediction: A systematic review and future guidelines," *e-Informatica Software Engineering Journal*, vol. 13, no. 1, pp. 227–259, 2019.

[3] G. Catolino and F. Ferrucci, "Ensemble techniques for software change prediction: A preliminary investigation," in *Proc. 2nd IEEE Workshop on Machine Learning Techniques for Software Quality Evaluation (MaLTeSQuE 2018)*, 2018, pp. 25–30.

[4] R. D. C. Silva, P. R. Farah, and S. R. Vergilio, "Machine learning for change-prone class prediction: A history-based approach," in *Proc. 36th Brazilian Symposium on Software Engineering (SBES 2022)*, 2022, pp. 289–298.

[5] G. Catolino and F. Ferrucci, "An extensive evaluation of ensemble techniques for software change prediction," *Journal of Software: Evolution and Process*, vol. 31, no. 9, pp. e2156:1–15, 2019.

[6] M. Elish and M. Al-Rahman AlKhiaty, "A suite of metrics for quantifying historical changes to predict future change-prone classes in object-oriented software." *Jounal of Software: Evolution and Process*, vol. 25, no. 5, pp. 407–437, 2012.

[7] G. Catolino, F. Palomba, A. D. Lucia, F. Ferrucci, and A. Zaidman, "Developer-related factors in change prediction: An empirical assessment," in *Proc. 25th International Conference on Program Comprehension (ICPC 2017)*, 2017, pp. 186–195.

[8] H. Lu, Y. Zhou, B. Xu, H. Leung, and L. Chen, "The ability of object-oriented metrics to predict change-proneness: a meta-analysis," *Empirical Software Engineering*, vol. 17, no. 3, pp. 200–242, 2012.

[9] P. R. Farah, R. de Carvalho Silva, and S. R. Vergilio, "An assessment of machine learning algorithms and models for prediction of change-prone Java methods," in *Proc. 37th Brazilian Symposium on Software Engineering (SBES 2023)*, 2023, pp. 322–331.

[10] G. Catolino, F. Palomba, F. A. Fontana, A. D. Lucia, A. Zaidman, and F. Ferrucci, "Improving change prediction models with code smell-related information," *Empirical Software Engineering*, vol. 25, no. 3, pp. 49–95, 2020.

[11] M. Alkhiaty, R. Abdel-Aal, and M. Elish, "Abductive network ensembles for improved prediction of future change-prone classes in object-oriented software," *International Arab Journal of Information Technology*, vol. 14, no. 6, pp. 803–811, 2017.

[12] M. Kersten and G. C. Murphy, "Using task context to improve programmer productivity," in *Proc. 14th International Symposium on Foundations of Software Engineering (FSE 2006)*, 2006, pp. 1–11.

[13] K. Mik and M. G. C., "Mylar: A degree-of-interest model for IDEs," in *Proc. 4th International Conference on Aspect-Oriented Software Development (AOSD 2005)*, 2005, pp. 159–168.

[14] N. Sae-Lim, S. Hayashi, and M. Saeki, "Revisiting context-based code smells prioritization: On supporting referred context," in *Proc. XP 2017 Scientific Workshops*, 2017, pp. 1–5.

[15] ——, "Context-based approach to prioritize code smells for prefactoring," *Journal of Software: Evolution and Process*, vol. 30, no. 6, pp. e1886:1–24, 2018.

[16] 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.

[17] N. Ohlsson and H. Alberg, "Predicting fault-prone software modules in telephone switches," *IEEE Transactions on Software Engineering*, vol. 22, no. 12, pp. 886–894, 1996.

[18] H. Hata, O. Mizuno, and T. Kikuno, "Bug prediction based on fine-grained module histories," in *Proc. 34th International Conference on Software Engineering (ICSE 2012)*, 2012, pp. 200–210.

[19] E. Giger, M. D'Ambros, M. Pinzger, and H. C. Gall, "Method-level bug prediction," in *Proc. 6th International Symposium on Empirical Software Engineering and Measurement (ESEM 2012)*, 2012, pp. 171–180.

[20] R. Almhana, W. Mkaouer, M. Kessentini, and A. Ouni, "Recommending relevant classes for bug reports using multi-objective search," in *Proc. of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE 2016)*, 2016, p. 286–295.

[21] J. Zhou, H. Zhang, and D. Lo, "Where should the bugs be fixed? More accurate information retrieval-based bug localization based on bug reports," in *Proc. 34th International Conference on Software Engineering (ICSE 2012)*, 2012, pp. 14–24.

[22] W. Zhang, Z. Li, Q. Wang, and J. Li, "FineLocator: A novel approach to method-level fine-grained bug localization by query expansion," *Information and Software Technology*, vol. 110, pp. 121–135, 2019.

[23] S. Tsumita, S. Hayashi, and S. Amasaki, "Large-scale evaluation of method-level bug localization with FinerBench4BL," in *Proc. 30th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER 2023)*, 2023, pp. 815–824.

[24] K. C. Youm, J. Ahn, and E. Lee, "Improved bug localization based on code change histories and bug reports," *Information and Software Technology*, vol. 82, pp. 177–192, 2017.

[25] S. Wang and D. Lo, "Version history, similar report, and structure: Putting them together for improved bug localization," in *Proc. 22nd International Conference on Program Comprehension (ICPC 2014)*, 2014, pp. 53–63.

[26] M. Aniche, *Java code metrics calculator (CK)*, 2015, available in https://github.com/mauricioaniche/ck/.

[27] B. Fluri, M. Wursch, M. Pinzger, and H. C. Gall, "Change distilling: Tree differencing for fine-grained source code change extraction," *IEEE Transactions on Software Engineering*, vol. 33, no. 11, pp. 725–743, 2007.

[28] H. Hata, O. Mizuno, and T. Kikuno, "Historage: Fine-grained version control system for Java," in *Proc. 12th International Workshop on Principles of Software Evolution and the 7th Annual ERCIM Workshop on Software Evolution (IWPSE-EVOL 2011)*, 2011, pp. 96–100.

[29] R. Daniele and P. Martin, "Using source code metrics to predict change-prone Java interfaces," in *Proc. 27th IEEE International Conference on Software Maintenance (ICSM 2011)*, 2011, pp. 303–312.

[30] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer, "SMOTE: Synthetic minority over-sampling technique," *Journal of Artificial Intelligence Research*, vol. 16, no. 1, pp. 321–357, 2002.

[31] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, no. 85, pp. 2825–2830, 2011.

[32] S. Shiba and S. Hayashi, "Historinc: A repository transformation tool for fine-grained history tracking," *Computer Software*, vol. 39, no. 4, pp. 75–85, 2022.

[33] J. Romano, J. Kromrey, J. Coraggio, and J. Skowronek, "Appropriate statistics for ordinal level data: Should we really be using t-test and cohen's d for evaluating group differences on the NSSE and other surveys?" in *Proc. 2006 Annual Meeting of the Florida Association of Institutional Research (FAIR 2006)*, 2006, pp. 1–33.

[34] N. Sae-Lim, S. Hayashi, and M. Saeki, "An investigative study on how developers filter and prioritize code smells," *IEICE Transactions on Information and Systems*, vol. E101-D, no. 7, pp. 1733–1742, 2018.

[35] J. Munson and S. Elbaum, "Code churn: A measure for estimating the impact of code change," in *Proc. 14th International Conference on Software Maintenance (ICSM 1998)*, 1998, pp. 24–31.

[36] H. Sugimori and S. Hayashi, "Artifacts of Revisiting method-level change prediction: A comparative evaluation at different granularities," figshare, https://doi.org/10.6084/m9.figshare.27678486, 2025.