Can Automated Impact Analysis Techniques Help Predict Decaying Modules?

Natthawute Sae-Lim, Shinpei Hayashi, and Motoshi Saeki
School of Computing, Tokyo Institute of Technology, Tokyo 152–8552, Japan
Email: {natthawute, hayashi, saeki}@se.cs.titech.ac.jp

***Abstract*—A decaying module refers to a module whose quality is getting worse and is likely to become smelly in the future. The concept has been proposed to mitigate the problem that developers cannot track the progression of code smells and prevent them from occurring. To support developers in proactive refactoring process to prevent code smells, a prediction approach has been proposed to detect modules that are likely to become decaying modules in the next milestone. Our prior study has shown that modules that developers will modify as an estimation of developers' context can be used to improve the performance of the prediction model significantly. Nevertheless, it requires the developer who has perfect knowledge of locations of changes to manually specify such information to the system. To this end, in this study, we explore the use of automated impact analysis techniques to estimate the developers' context. Such techniques will enable developers to improve the performance of the decaying module prediction model without the need of perfect knowledge or manual input to the system. Furthermore, we conduct a study on the relationship between the accuracy of an impact analysis technique and its effect on improving decaying module prediction, as well as the future direction that should be explored.**

I. Introduction

Code smell is often used to represent an indicator of a design flaw or a problem in source code [1]. For instance, a class that has God Class code smell is considered as having too many functionalities and overly complex. Such problems are found to be related to software maintenance problems [2]. As a consequence, many approaches and tools have been proposed to detect code smells using different kinds of information such as source code metrics [3], [4] or historical information [5]. Nevertheless, one drawback of most tools is that they focus on a detect-and-remove strategy, which means that developers can only solve code smells after their source code become smelly. In other words, developers can neither track the progression of code smells or prevent them from occurring. To this end, we previously proposed the concept of a decaying module, which is a module whose quality is getting worse and is likely to become smelly in the future [6]. This technique allows developers to prevent code smells from occurring.

In order to support developers' refactoring planning strategy, we also proposed a machine learning approach to predict modules that will decay in the future [6]. The baseline uses source code quality as predictor variables and whether the given module will decay in the next release as a response variable. In addition to the baseline model, we also conducted a preliminary study on using developers' context as an additional predictor variable and found that such information can significantly improve the performance of the prediction model. We used a set of modules that developers are going to modify as a proxy to express developers' context. However, the preliminary study was conducted under the assumption that the developer who uses the system must have perfect knowledge of the locations of the changes. In addition, the developer has to specify such information into the system manually.

To bridge this gap, in this study, we explore the use of an alternative approach that can estimate the context of developers instead of relying on knowledge of developers. We leverage automated impact analysis techniques which refer to techniques that automatically identify a full set of modules that are likely to be affected by a particular change [7]. Such techniques use the information commonly available in a software development project, such as change descriptions, to predict modules that developers are going to modify. We conduct a study to verify whether the predicted modules can be used as an estimation of developers' context, i.e., they can also be used to improve the prediction model performance of decaying modules. Such an approach will enable developers to improve the performance of the prediction model without the need for perfect knowledge of change locations nor the need for manual input to the system.

Moreover, while automated impact analysis techniques have been studied extensively in the literature [8], the approach is still far from being perfect. Although many attempts have been made to improve the accuracy of the approach, the relationship between the accuracy of an impact analysis technique and the performance of decaying module prediction remains unclear. Therefore, in this paper, we study the relationship by artificially modifying the accuracy of impact analysis. The result can be used as empirical evidence to show how we can further improve the impact analysis technique in the future.

The main contributions of this study are twofold. First, we show that developers' context estimated by impact analysis techniques can also improve decaying module prediction performance. Second, we present an investigation on how we can improve the impact analysis technique to enhance the performance of decaying module prediction model further.

The remainder of this paper is organized as follows. The next section summarizes the idea of a decaying module and its prediction. Section III presents our empirical studies. Section IV presents threats to validity of this study. Section V concludes this paper.

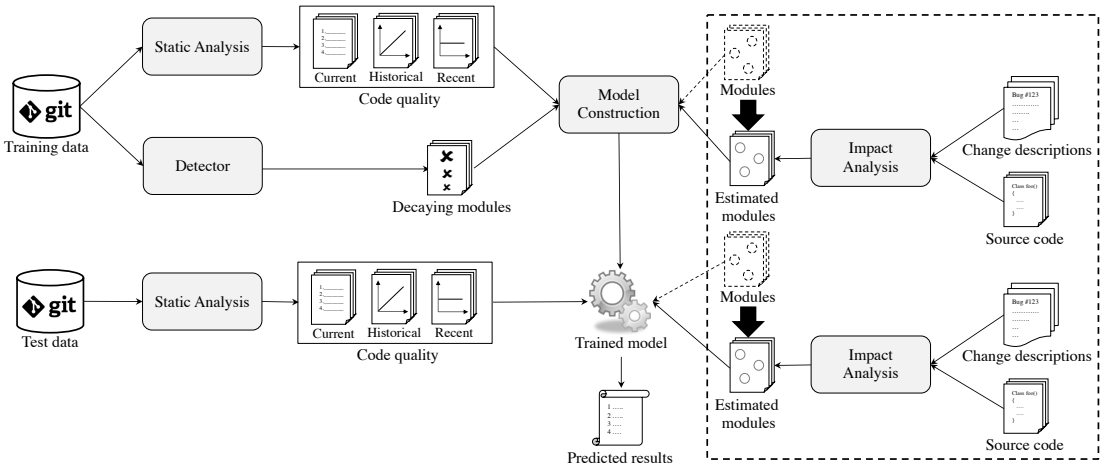


Fig. 1. Decaying modules prediction approach.

II. Decaying Module and Its Prediction

A. Decaying Module

The concept of a decaying module was proposed to mitigate the gap of most code smell detectors that warn developers about the quality problems only after they occur in the system [6], which did not allow developers to *prevent* the problems before. A decaying module was defined as a module that is getting closer to becoming smelly during a specified period. This would allow developers to handle modules before they become smelly.

In order to detect decaying modules, the Module Decay Index (MDI) was also proposed as an indicator of how close a module is to becoming code smell [6]. The MDI can be used in a situation of metric-based code smell detection that use multiple conditions to detect a code smell. Each condition is determined by whether a particular metric exceeds their corresponding thresholds. In this case, we can calculate MDI by averaging the ratio of a metric to its threshold of each condition. The MDI is range from 0 to 1. A higher MDI means that the module is closer to become a code smell. A module is considered a decaying module if its MDI has been increased from the previous release.

B. Decaying Modules Prediction

The idea of this approach is to use the information of the current release to predict modules that are going to decay in the next release. Figure 1 shows an overview of the baseline approach of our prior work and the modification we make in this study. The part outside the dashed box represents an overview of the baseline approach. The data is divided into training data and test data. Assuming that we want to predict if a given module will decay in release $n+1$, the training data will be the source code of a set of releases $M = \{1, 2, \ldots, n-1\}$ and the test data will be the source code of release n. First, for each release $m \in M$, we calculate three types of code quality of the source codes in training data to construct the classifier model. The three types of code quality are current code quality, historical code quality, and recent code quality trend. Then, we apply the detector to the source code of release $m + 1$ to identify decaying modules, i.e., modules that are getting closer to become smelly. Based on this information from all releases in M, the predictor variables (code quality) and the response variable (whether the module gets decayed or not) are used to construct the classifier model. Then, we apply the same static analyzer to calculate the code quality of source code in release n and input to the model. Finally, the result of the model indicate modules that are likely to decay in release $n + 1$.

Additionally, the part that we change from the setting of our prior study is shown in the dashed box of Fig. 1. In our prior work, we showed that adding developers' context information, which is expressed as a set of modules that developers are going to modify, as an additional predictor variable can improve the performance of the prediction model significantly. However, our prior study was conducted under the condition that the developer has perfect knowledge of modules that will be modified. Furthermore, the developer has to specify such information into the system manually. To alleviate these conditions, in this study, we explore the use of an automated approach that does not require perfect knowledge of developers. Specifically, we focus on the use of information retrieval (IR)-based impact analysis technique to estimate developers' context. The underlying reason is that it requires a minimum amount of information to perform: only change descriptions and source code. Such information is commonly available in a software development project. The IR-based impact analysis takes the change descriptions that developers are going to implement and the source code to predict the locations that are going to be modified. The details will be explained in the next section.

III. Empirical Study

A. Research Questions

In this study, we aim at answering the following research questions:

TABLE I
OPTIMIZED PARAMETERS OF IR-BASED IMPACT ANALYSIS TECHNIQUES

Project	Technique	Cut point
Accumulo	BM25	20
Ambari	BM25	40
Derby	VSM	30
Hive	BM25	10

RQ1: Can developers' context estimated by an IR-based impact analysis technique improve prediction performance?

RQ2: How can we further improve the performance of prediction model?

The details of each research question will be discussed later.

B. Experimental Setup

1) Data Collection: We use the same dataset as the baseline approach, which comprises of four open source projects: Accumulo, Ambari, Derby, and Hive. First, we obtain a list of issues of each project from their issue tracking systems[1]. Then, we extract summary, description, and the release that the issue was implemented. For each issue, we applied an impact analysis to generate locations that are likely to be changed to complete the issue. In this study, we focus on IR-based impact analysis because it requires minimum information to perform, i.e., it takes only the change descriptions and source code as inputs of the technique. The IR-based impact analysis works by calculating the textual similarity between an issue and source code. We consider the similarity score determined by the impact analysis technique as a probability that a particular module will be modified. Then, we calculate the Context Relevance Index (CRI), which was proposed in our prior work [9]. The CRI of a module can be calculated by the summation of the similarity score in all issues that contains the module. The CRI represents the relevance of each module to the context of developers. A higher value of CRI means higher relevance to the context, i.e., more likely to be modified. We then create a new variable representing the CRI value. In order to calculate the CRI value, we need to specify two parameters: the technique used by IR-based impact analysis and the cut point when calculating CRI. The technique used by IR-based impact analysis determines how the data is represented and how the similarity is calculated while the cut point determines the number of modules used when calculating CRI. In this study, we adopt four fundamental IR-based impact analysis techniques which are often studied in impact analysis research: Vector Space Model (VSM), Latent Semantic Indexing (LSI), Latent Dirichlet allocation (LDA), and Okapi BM25 (BM25). For the cut point, we use 10, 20, 30, and 40, which are usually used in the previous research [7], [9]. For each project, we try all combination of each impact analysis technique and cut point, e.g., VSM with 10 cut point or VSM with 20 cut point. We then finally select the best combination of each project to use in this study. The underlying reason is that we want to simulate the real world setting where parameters of the technique are optimized for each project before utilizing the technique. The best combination of each project can be found in Table I.

[1]https://issues.apache.org/jira/secure/Dashboard.jspa

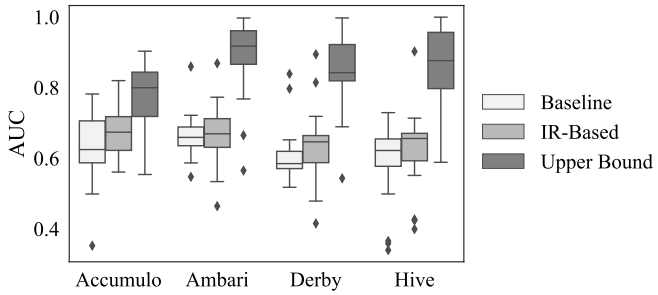


Fig. 2. AUC values of baseline, IR-based models and the upper bound models.

C. RQ1: Can developers' context estimated by an IR-based impact analysis technique improve prediction performance?

1) Motivation: As discussed earlier, we showed that developers' context (i.e., the modules that developers are going to modify) could be used to improve decaying module prediction performance. However, with the purpose of obtaining the upper bound performance of the model, the study was conducted based on the assumption of the perfect knowledge of developers. In addition, the developer has to input such information to the system manually. To this end, in this study, we propose an alternative approach that does not rely on perfect knowledge of developers. By using the results of automated impact analysis technique to represent developers' context, we suspect that it can also improve the performance of the model without the need of perfect knowledge from developers. However, since such techniques do not have perfect accuracy, it is sensible that such improvement is smaller than using the perfect knowledge of developers.

2) Study Design: We compare the accuracy of the prediction model between the baseline and the model with CRI value. We conduct the Wilcoxon signed-rank test with the following null hypothesis to confirm if the results are statistically significant.

H_0: Developers' context estimated using IR-based impact analysis technique does *not improve* the performance of prediction model.

Therefore, an alternative hypothesis can be defined as:

H_a: Developers' context estimated using IR-based impact analysis technique *improves* the performance of prediction model.

In addition, we calculate Cliff's delta (d) as a measure of the magnitude of the improvement. The Cliff's delta is interpreted based on the threshold by Romano et al. [10]: negligible for $|d| < 0.147$, small for $|d| < 0.33$, medium for $|d| < 0.474$, and large for $|d| \geq 0.474$.

TABLE II
RESULTS OF WILCOXON SIGNED-RANK TESTS AND THE CLIFF'S DELTA EFFECT SIZE TESTS

Project	p value	Cliff's delta
Accumulo	<0.001	0.293 (small)
Ambari	0.237	0.067 (negligible)
Derby	0.007	0.382 (medium)
Hive	0.002	0.218 (small)

3) Results and Discussion: Figure 2 shows the accuracy of the prediction model between the baseline model, the model using IR-based impact analysis techniques to estimate developers' context and the upper bound models. When comparing the accuracy between the baseline and IR-based models, we can observe that the IR-based model tends to have higher performance than the baseline model. Specifically, the median of the AUC values have been improved from 0.62 to 0.67 for Accumulo, from 0.66 to 0.67 for Ambari, from 0.58 to 0.65 for Derby, and from 0.62 to 0.66 for Hive.

Table II shows the results of Wilcoxon sign-rank tests and Cliff's delta effect size tests. The cells with p values less than 0.05 and Cliff's delta higher than 0.147 (not negligible) are highlighted in gray. The results of the Wilcoxon signed-rank test shows that the results are statistically significant at $\alpha = 0.05$ except for Ambari project. Therefore, for the projects other than Ambari, we can reject the null hypothesis and conclude that developers' context estimated by IR-based impact analysis technique can improve the performance of the prediction model. Furthermore, Cliff's delta values show that the result has a small effect for Accumulo and Hive, medium effect for Derby, and negligible effect for Ambari. So, we can see that the improvements are not negligible for all projects except for Ambari.

However, when we compare the improvement of the performance of the prediction model between IR-based and upper bound models, we can see that the improvements of IR-based models are much lower than the ones of the upper bound model. The result is as expected because the improvement of the upper bound model assumes perfect knowledge of developers, but the impact analysis technique relies on change descriptions to estimate the context. We can conclude that the IR-based impact analysis technique has a potential of representing developers' context, although using only textual change descriptions and source code as inputs.

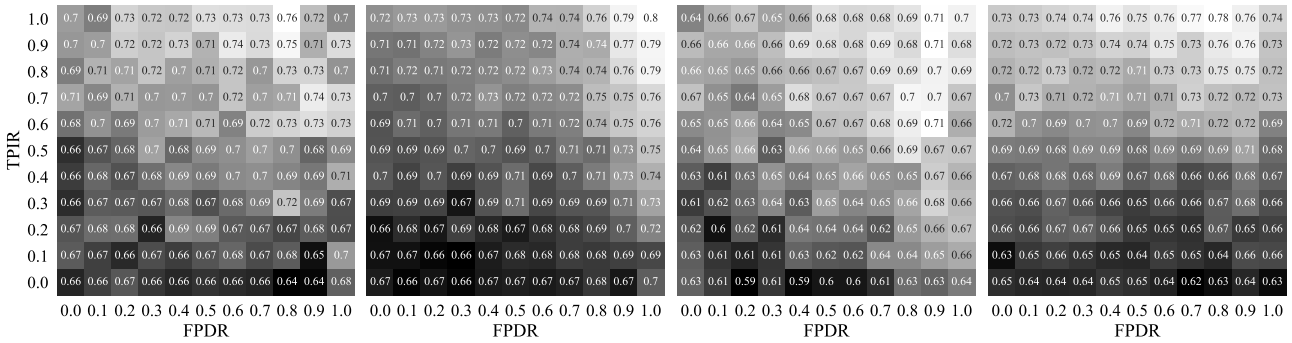
In conclusion, developers' context estimated by IR-based impact analysis technique can improve the performance of the decaying module prediction model.

D. RQ2: How can we further improve the performance of prediction model?

1) Motivation: As discussed in earlier sections, while the context estimated by existing IR-based impact analysis techniques can help improve the performance of prediction model, the improvements are still low comparing to the situation of perfect knowledge of developers. One reason for such small improvement may be the low accuracy of the IR-based impact analysis, i.e., the high number of false positives and the low number of true positives. If the impact analysis technique results in many false positives, they will become noises that may obstruct the prediction model instead of helping them identify decaying modules. Furthermore, if the number of true positives is low, the impact analysis techniques can predict only a part of the correct answers and fail to detect the rest and, therefore, provide insufficient information to the prediction model. To this end, many approaches have been proposed to improve the accuracy of impact analysis techniques such as combining IR-based approach with extra information [8]. Nevertheless, even the state-of-the-art approach is still far from being perfect. Although the research community has been working on improving the accuracy of impact analysis techniques, it is still unclear whether high accuracy impact analysis techniques can improve the decaying module prediction model. Thus, to obtain empirical evidence, we artificially tune the accuracy of impact analysis techniques and observe the relationship between the accuracy of an impact analysis technique and decaying module prediction model. We expect the result to be useful for the future direction of impact analysis research.

2) Study Design: We conducted an analysis under the assumption that mitigating the problems of high false positives and low true positives can improve the performance of the prediction model. We artificially modified the result of IR-based impact analysis technique in two steps, which are inspired by the task input generation approach of a feature location study [11]: First, we decrease the number of false positives by randomly removing false positives from the result. Second, we increase the number of true positives by randomly adding false negatives to the result. We refer to the ratio that we decrease the number of false positives as False Positive Decrement Ratio (FPDR) and to the ratio that we increase the number of true positives as True Positive Increment Ratio (TPIR). Both of the ratios are from 0.0 to 1.0 with the step of 0.1. After performing the modification, we recalculate the CRI and use it as an exploratory variable of the prediction model. Finally, we calculate the performance of each model for comparison.

3) Results and Discussion: Figure 3 represents the heat map of the AUC of the prediction model. The vertical axis represents the values of TPIR, while the horizontal axis represents the values of FPDR. Each cell represents the AUC value of each setting. The brighter color shows a higher AUC, while the darker color shows the lower AUC values. In general, we can observe that AUC values tend to have a higher value in the top right corner of the heat map (e.g., in Ambari project). This result suggests that the more we decrease the number of false positives, and the more we increase the number of true positives, the higher AUC values become. Moreover, when we observe the value with low TPIR, we can see that increasing FPDR does not significantly improve the AUC values. This may indicate that increasing the number of true positives should be given higher priority than decreasing the number



(a) Accumulo. (b) Ambari. (c) Derby. (d) Hive.

Fig. 3. Comparison of the prediction model performance at different TPIR and FPDR.

of false positives.

Technically, decreasing the number of false positives may be accomplished by complimenting IR-based approach with other approaches such as dynamic analysis approach. For example, we can use execution trace to filter irrelevant modules from the result of IR-based approach, which may result in a lower number of false positives [7]. On the other hand, increasing the number of true positives can be done by combining the IR-based approach with a technique such as mining software repositories (MSR). For instance, MSR approach can adopt association mining rules to detect modules that were often modified together in the past and use that information to detect the modules that may not be found by only IR-based approach [7]. While combining both techniques together have been shown to improve the accuracy of impact analysis techniques [7], one downside is that it requires extra information which may or may not be available depending on the projects.

To sum up, whereas improving the accuracy of the impact analysis techniques by decreasing the number of false positives and increasing the number of true positives can improve decaying modules prediction, we should give higher priority to increasing the number of true positives.

IV. Threats to Validity

One significant threat to validity when using change descriptions to estimate developers' context lies in software repositories. For example, developers may make some modifications unrelated to any issue in the issue tracking system. In this situation, impact analysis techniques will fail to include such modification in the estimated list. In addition, as we rely on commit messages to identify the true positives of each issue, if developers do not put the details of the issue to the commit messages, our technique will fail to identify the true positives. We mitigated this threat by filtering the projects that have a high ratio of commits that include issue ID (higher than 80%) based on the list by Miura et al. [12]. This can ensure that most of the changes were related to the issues in the issue tracking system.

V. Conclusion

In this paper, we explored the possibility of improving decaying modules prediction performance by using developers' context estimated by automated impact analysis techniques. We found that the context estimated by IR-based impact analysis techniques can improve the performance of the prediction model. We also discussed that the performance could be further improved by improving the accuracy of impact analysis technique such as decreasing the number of false positives and increasing the number of true positives of the results of impact analysis techniques.

Acknowledgments

This work was partly supported by JSPS Grants-in-Aid for Scientific Research Number JP18K11238.

References

[1] M. Fowler, *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[2] A. Yamashita and L. Moonen, "Do code smells reflect important maintainability aspects?" in *Proc. ICSM*, 2012, pp. 306–315.

[3] M. Lanza and R. Marinescu, *Object-Oriented Metrics in Practice*. Springer, 2006.

[4] F. A. Fontana, V. Ferme, M. Zanoni, and R. Roveda, "Towards a prioritization of code debt: A code smell intensity index," in *Proc. MTD*, 2015, pp. 16–24.

[5] F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, A. De Lucia, and D. Poshyvanyk, "Detecting bad smells in source code using change history information," in *Proc. ASE*, 2013, pp. 268–278.

[6] N. Sae-Lim, S. Hayashi, and M. Saeki, "Toward proactive refactoring: An exploratory study on decaying modules," in *Proc. IWoR*, 2019, pp. 1–10.

[7] M. Gethers, B. Dit, H. Kagdi, and D. Poshyvanyk, "Integrated impact analysis for managing software changes," in *Proc. ICSE*, 2012, pp. 430–440.

[8] B. Dit, M. Revelle, M. Gethers, and D. Poshyvanyk, "Feature location in source code: A taxonomy and survey," *J. Softw. Evol. Proc.*, vol. 25, no. 1, pp. 53–95, 2013.

[9] N. Sae-Lim, S. Hayashi, and M. Saeki, "Context-based approach to prioritize code smells for prefactoring," *J. Softw. Evol. Proc.*, vol. 30, no. 6:e1886, pp. 1–24, 2018.

[10] J. Romano, J. D. 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 *FAIR*, 2006, pp. 1–33.

[11] T. Ishio, S. Hayashi, H. Kazato, and T. Oshima, "On the effectiveness of accuracy of automated feature location technique," in *Proc. WCRE*, 2013, pp. 381–390.

[12] K. Miura, S. McIntosh, Y. Kamei, A. E. Hassan, and N. Ubayashi, "The impact of task granularity on co-evolution analyses," in *Proc. EMSE*, 2016, pp. 1–10.