DEVELOPMENT AND EVALUATION OF A METHOD FOR IDENTIFYING LOGIC ERRORS USING MACHINE LEARNING WITH A FOCUS ON PROGRAM STRUCTURE

Yuta Harada
Department of Education,
Tokyo Gakugei University,
4-1-1, Nukuikitamachi, Koganei-shi, Tokyo 184-8501, Japan
m248120f@st.u-gakugei.ac.jp

Soichiro Sato
Department of Education,
Tokyo Gakugei University,
4-1-1, Nukuikitamachi, Koganei-shi, Tokyo 184-8501, Japan
m218113m@st.u-gakugei.ac.jp

Shoichi Nakamura
Department of Computer Science and Mathematics,
Fukushima University
1, Kanayagawa, Fukusima-shi, Fukushima 960-1296, Japan
nakamura@sss.fukushima-u.ac.jp

Youzou Miyadera
Division of Natural Science,
Tokyo Gakugei University,
4-1-1, Nukuikitamachi, Koganei-shi, Tokyo 184-8501, Japan
miyadera@u-gakugei.ac.jp

ABSTRACT

In programming exercise classes at universities, instructors must quickly determine the learner's situation and provide support. However, it is difficult to deal with learners making logic errors. This is because there is more than one way to solve a task that a learner is working on, and the method of program logic coding (hereinafter, “coding method”) differs for each student. Therefore, when instructors help learners solve logic errors, they must be supported by considering each learner's coding method. This study aims to develop a method for identifying logic errors that learners are currently making, considering each learner's coding method. In this study, we assume that the learner’s coding method is reflected in the program structure. We then attempt to analyze the program focusing on its structure. Specifically, using clustering, we tried to categorize source code by the learner’s coding method and identify logic errors in it. Results of evaluating our method using the Hold-out Validation show it can identify logic errors with 70% accuracy. This result indicates a new approach for instructors to help learners when they are stuck during programming practice.

KEYWORDS

Programming Learning Support, Logic Error, Identifying Stumbling, Source Code Editing History, Machine Learning

1. INTRODUCTION

In recent years, many educational institutions such as universities have been widely implementing programming exercise classes, and the need for learning support has increased. To provide accurate guidance in programming exercises, the learning status of the learners needs to be grasped. However, in university programming exercises, providing individual learning support to each learner is challenging due to a small number of instructors or teaching assistants needing to support many learners. In programming exercises, learners face two types of errors: grammar errors, which are expressed as compilation errors, and logic errors, which are not. Grammar errors can be expected to be solved by learners through trial and error. On the other hand, logic errors are difficult to solve because learners must find and correct the causes by themselves. In addition, the method of program logic coding (hereinafter, "coding method") varies from learner to learner. Therefore, when instructors help learners solve logic errors, they must be supported by considering each learner's coding method.

Several methods have been proposed to support learners in programming exercises. Two methods (Ichimura et al., 2013; Alammary et al., 2012) have been proposed for grasping the learning status in programming exercises. Another method (Urakami et al., 2020) has been proposed for the automatically detecting impasses faced by learners. Also, the research (Singh et al., 2013) developed an automated feedback system for programming assignments, identifying errors and guiding students to correct them. Additionally, a method (Kawasaki et al., 2023) has been proposed for identifying impasses and their locations related to logic errors. However, the learner's coding method is not considered in these methods.

Furthermore, studies (Ishiwada et al., 2017; Kawaguchhi et al., 2020) have identified the learning status by analyzing the learner's source code editing process. This approach involves creating a machine learning model that outputs the corresponding learning status when provided with the source code editing history. Nevertheless, this model is developed based on token rewriting information. For this reason, the learner's coding method is not considered in these studies.

To summarize previous research, the issue remains that the learner's coding method is not sufficiently considered to support logic errors

Therefore, this study aims to develop a method for identifying the logic error caused by the source code currently being worked on for learners making logic errors, taking into account the coding method of the logic error. In this study, we particularly focused on the program structure in which the coding method is thought to be expressed and attempted to achieve the purpose by performing clustering based on the similarity of the source code. After that, we decided to evaluate the proposed method using the hold-out method. This shows the possibility of providing new support for instructors to grasp impasses during programming exercises.

2. RESEARCH POLICY

2.1 Overview of Method for Identifying Logic Errors

In this study, we develop a method to identify logic errors in the current source code caused by learners during an exercise, considering each learner's coding method. We assume that the learner's coding method is reflected in the program structure and attempt to analyze the program focusing on its structure. Using clustering, we tried to categorize source code by the learner's coding method and identify logic errors in it. By analyzing the source code containing logic errors from past learners and preemptively classifying it by coding method and logic error type, the logic errors can be identified based on the structure of new learners' programs.

Specifically, we extract abstract syntax trees (ASTs) from the source code containing logic errors from past learners. Next, we calculate the similarity between source codes using the Kast1 Spectrum Kernel proposed by Torres et al. (Torres at al., 2018), based on the ASTs. Subsequently, we attempt to classify the source codes by coding method and further categorize the logic errors using clustering based on the calculated similarities. Based on these results, we create datasets (sets of logic errors and their representative source codes) for each solution and logic error.

Finally, we calculate the similarity between the datasets and the current source code containing logic errors caused by learners during an exercise. We then identify that the learner is causing the logic error associated with the dataset that has the highest similarity and present the results to the instructor. We adopt the Hold-out Validation to validate the effectiveness of the proposed method.

2.2 Research Procedure

This study will proceed as follows:

- Calculation of similarity between source codes (Chapter3)
- Creation of dataset (Chapter 4)
- Implementation of logic error identification method (Chapter 5)
- Evaluation and discussion (Chapter 6)

In Chapter 3, we extract ASTs from the source code containing logic errors from past learners and calculate the similarity between source codes using the Kast1 Spectrum Kernel proposed by Torres et al. (Torres at al., 2018), the basis of the ASTs. This method enables analysis focused on the structure of the programs.

In Chapter 4, we use clustering based on the calculated similarity to classify the source codes by coding method and further categorize the logic errors. Based on these results, we create datasets for each coding method and logic error, consisting of one representative source code and its corresponding logic errors.

In Chapter 5, we divide the source code containing logic errors from past learners into training data and validation data in a 9:1 ratio. From the training data, we create datasets as described in Chapter 4. We then treat the validation data as the source code currently being worked on by the learners and then develop and implement the proposed method.

In Chapter 6, we evaluate and discuss the proposed method.

3. CALCULATION OF SIMILARITY BETWEEN SOURCE CODES

3.1 Overview

In this chapter, we describe the calculation of source code similarity, which is a necessary step in developing the proposed method discussed in the previous chapter. To create a dataset of logic errors, it is necessary to calculate the similarity between source codes containing logic errors caused by past learners and use clustering based on the calculated similarities. When comparing source codes, variable names and function names may have minor differences depending on the learner, and meaningless code may be mixed into the program. Therefore, the notation needs to be somewhat standardized. By converting the code into ASTs generated by the compiler, unnecessary parts for code generation are removed, and semantically similar elements are represented as common nodes. In this study, we use a low level virtual machine (LLVM) to generate the ASTs (Section 3.2). Next, we generate a token sequence consisting of literals and weights (Section 3.3). After that, we calculate the similarity between source codes using the Kast1 Spectrum Kernel proposed by Torres et al. (Torres at al., 2018) (Section 3.4).

3.2 Generation of Abstract Syntax Trees

In this section, we define the details of generating ASTs from the source code. In this study, we used LLVM to generate the ASTs. Although ASTs contain various types of information beyond nodes, for this research, we retained only the nodes and tree structures used by the Kast1 Spectrum Kernel and removed all other information.

3.3 Conversion to Token Sequences

Next, we want to apply clustering while maintaining the structural information of the program. Therefore, the ASTs need to be converted into numerical information. In this study, we follow the algorithm proposed by Torres et al. (Torres at al., 2018) to convert the ASTs into token sequences. During this process, we also compress the token sequences to make them easier to handle.

Simply listing the nodes that appear in the ASTs would lose the tree structure, so we introduce a new token "LEVEL_UP." This token is added with increased weight as we move up from the bottom while reading the ASTs, with the aim of maintaining the tree structure while treating it as a token sequence.

The generated token sequence is often redundant, so compression is carried out. The compression algorithm consists of four steps:

1. Tokens with consecutive literals are merged, and their weights are summed.

2. Cast expressions, etc. are removed, and their weights are added to the subsequent token.
3. Declarations and "LEVEL_UP" between the tokens are removed, and their weights are added to the previous token.
4. Token pairs with the same literals are merged, and their weights are summed.

At this point, the token sequence (consisting of literals and weights) is complete.

3.4 Calculation of Similarity

Finally, we will calculate the similarity between source codes using the Kast1 Spectrum Kernel proposed by Torres et al. (Torres at al., 2018). The basic procedure is as follows: we compare two token sequences and find the longest common subsequence of tokens along with the sum of their weights. This value becomes the first dimension of our calculation. Then, we find the next longest common subsequence of tokens from the parts of the token sequences that were not used in the first-dimension calculation and use the sum of their weights as the second-dimension value. Similarly, we calculate up to the third dimension. Since a 3D vector is generated for each token sequence, we can calculate the similarity by dividing the dot product of these two vectors by the product of the sum of the weights of the two token sequences.

4. CREATION OF DATASET

4.1 Overview

In this chapter, we describe the creation of a dataset based on the similarity calculated in Chapter 3. Having calculated the similarities of the source codes, we proceed with clustering based on these similarities. Next, based on the clustering results, we classify the source codes by coding method, further categorize them by logic errors, and create datasets for each logic error. The datasets consist of each logic error and one representative source code containing that logic error. The representative source code is selected arbitrarily. The following details the process of creating.

4.2 Classification of Logic Errors using Clustering

Having calculated the similarities of the source codes in Chapter 3, we proceed with clustering based on these similarities, using SciPy's linkage function (SciPy, 2022). In this study, we use dendrograms to visualize the data through clustering. The dendrogram groups the source codes based on their similarities and represents the degree of similarity among these clusters in a tree structure.

As an example of the dendrogram showing the clustering results, Table 1 presents the content of Task 35, and Figure 1 shows the dendrogram. The vertical axis of the dendrogram indicates the threshold representing the distance between clusters.

As can be seen from the dendrogram, the number of clusters varies depending on the threshold. For example, in the dendrogram of Figure 1, there are 2 clusters when the threshold is set to 60 and 3 clusters when the threshold is set to 30.

Because the optimal threshold for classifying logic errors needs to be determined, we conducted evaluations with various thresholds. By setting the threshold lower, finer clusters are formed.

Table 1. Contents of Task 35

Task No.	Contents
35	Create a program that takes two integers (m, n) as input and uses a recursive function sum to calculate and output the sum from m to n. It is assumed that m < n.

4.3 Validation of Clustering Results

In this section, we validate whether the clustering results obtained in the previous sections are reasonable. As an example of validation, we explain using Task 35, which was also used in Figure 1.

This task was selected because it is expected to have multiple coding methods and various logic errors. As a validation method, we start the analysis with a large threshold to check if the classifications are done by the coding method. Then, with a smaller threshold, we verify whether the classifications can be made by logic errors.

First, when the threshold is set to 20, the clusters are divided into four. The first cluster groups solutions that use a recurrence relation like "sum from m to n-1 + 1" to calculate the total sum from m to n. The second cluster groups solutions that calculate the total sum from m to n using the difference between two sums, such as "sum(n) - sum(m-1)". The third cluster is similar to the second but groups solutions with a different number of created functions. The fourth cluster groups solutions where the processing within the recursive function is generally incorrect. This shows that with a large threshold, coding methods are grouped into clusters.

Moreover, when the threshold is reduced to 12, the first cluster at threshold 20 is further divided into two. The first sub-cluster groups source codes where the condition part of the if statement for terminating the recursion is incorrect, while the second sub-cluster groups source codes where the structure of the if statement for terminating the recursion is incorrect. This shows that they are roughly grouped into clusters. Detailed discussions are provided in Chapter 6.

From these results, the classification of logic errors by a coding method using this clustering can be considered to be reasonably valid.

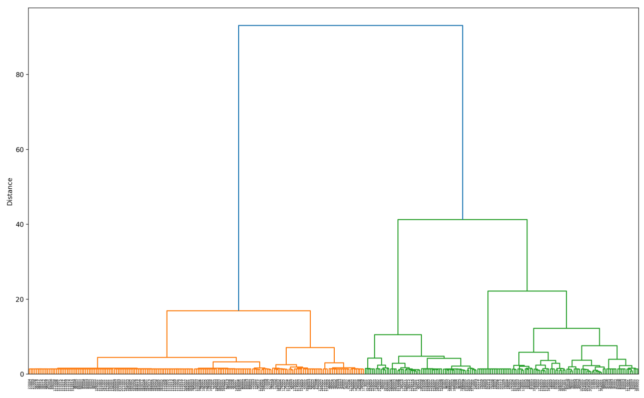

Figure 1. The dendrogram of Task 35

4.4 Creation of the Dataset

Using the clustering results validated in the previous section, we describe the creation of a dataset for identifying logic errors. The required dataset consists of sets of logic errors based on coding methods and source codes representing those logic errors. Based on the classification results of logic errors conducted in Section 4.3, one representative source code was selected for each logic error.

5. IMPLEMENTATION OF A METHOD FOR IDENTIFYING LOGIC ERRORS

5.1 Overview

In this chapter, we develop and evaluate the method for identifying logic errors using the datasets created in the previous chapters. In the previous chapters, we calculated the similarity between the datasets and the current source code containing logic errors caused by learners during an exercise. We then identified that the learner is causing the logic error associated with the dataset that has the highest similarity and present the results to the instructor.

In this study, we classify the source code containing logic errors of learners by logic errors based on the solution and create datasets. Since there are basically more than three types of datasets, the method for identifying logic errors to be developed will be a multi-class classification. Considering that the target method for identifying logic error is multi-class classification and there are many samples of source code containing logic errors from past learners, we decided to use the Hold-out Validation for evaluation to avoid overfitting. Below, we describe the development and evaluation of the proposed method.

5.2 Development of a Method for Identifying Logic Errors

In the programming practice courses conducted at our university, we selected 3 of the 111 tasks that have multiple coding methods and where various types of logic errors are anticipated based on the method. These selected tasks are shown in Table 2.

Furthermore, as can be seen from the dendrogram shown in Figure 1, the number of clusters varies depending on the threshold setting, which is expected to significantly affect the identification accuracy. Therefore, we investigated the appropriate threshold values for each task.

The sample data used for developing the method for identifying logic error were collected from the source codes of 162 students who took courses from 2019 to 2022 at our university, including those with logic errors. The results are shown in Table 3.

Table 2. Examples of selected tasks and their content

Task No.	Contents
35	Create a program that takes two integers (m, n) as input and uses a recursive function sum to calculate and output the sum from m to n. It is assumed that m < n.
43	Create a program that takes an integer input and outputs each digit of that value in the following manner using the recursive function reverse2: Input: 12345 Output: 5432112345
73	Create a program that takes an integer value as input, converts it to a binary number as a decimal number, and outputs the result. Use loop syntax and arrays to do this.

Table 3. The number of data used

Task No.	Number of students	Number of source codes
35	94	357
43	112	654
73	108	934

6. EVALUATION AND DISCUSSION

6.1 Overview

Based on the content of the previous sections, a logic error identification experiment was conducted. Initially, by using the data samples from the three selected tasks in the previous chapter, the training and validation data for dataset creation were divided into a 9:1 ratio. Subsequently, by comparing the learners' logic errors with those identified by the proposed method, precision, recall, and F1-score were calculated. Furthermore, since this is a multi-class classification, the macro average was considered for these metrics. The following sections will describe the logic error identification method developed for each task, in the order of clustering result validation, representative source code selection, and logic error identification results.

6.2 Identification of Logic Errors in Task 35

Figure 1 presents the dendrogram for Task 35. Verification was conducted to determine the appropriate threshold for Task 35.

First, when the threshold was set to 20, the clusters were divided into four. In the first cluster, coding methods were grouped that used a recursive formula such as "the sum from 'n' to 'm-1' + 1" to calculate the sum from 'm' to 'n'. The coding methods in this cluster were also found to be in line with the strategy assumed by the instructor. Next, in the second cluster, coding methods were grouped that took the difference between each sum, such as "sum(n) - sum(m-1)", to calculate the sum from 'm' to 'n'. In the third cluster, coding methods were like those in the second cluster, but the number of created functions was different. In the fourth cluster, source codes with errors in the processing of recursive functions were grouped. From the second cluster onwards, logic errors due to unexpected coding methods by the instructor were identified. Among them, the third cluster had coding methods that contradicted the part "based on the recursive function sum2 that calculates the sum from 'm' to 'n'" in the problem statement in Table 2. Therefore, these coding methods were clearly unsuitable for solving the current task. In this way, logic errors that deviated significantly from the instructor's expected coding methods were also extracted.

From this, it was found that with a threshold of 20, a certain level of classification was achieved for each coding method. However, this did not lead to identifying logic errors in detail. Therefore, we decided to further reduce the threshold to conduct a more detailed analysis.

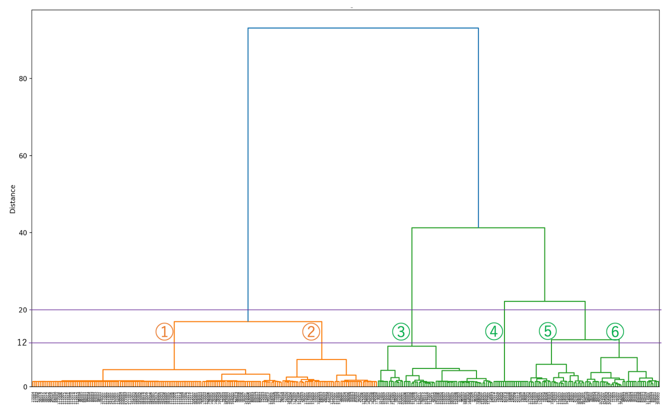

Figure 1. The dendrogram of Task 35

Next, the threshold was set to 12. At this threshold, the clusters were divided into six. It was found that the first cluster at the threshold of 20 was further divided into two. The first group consisted of source codes where the condition part of the if statement for recursive termination was incorrect, and the second group consisted of source codes where the structure of the if statement for recursive termination itself was incorrect. Additionally, it was observed that the fourth cluster at the threshold of 20 was further divided into two. The first group included source codes where the recursive formulaic processing that should have been done in the recursive function was in the main function, indicating that the recursion was not working correctly. The second group consisted of source codes where the processing inside the recursive function was incorrect overall.

Although reducing the threshold below 12 was considered, we were judged to have already classified the necessary logic errors somewhat at the threshold of 12. Therefore, the appropriate threshold for Task 35 was set to 12.

6.3 Identification of Logic Errors in Task 43

Figure 3 presents the dendrogram for Task 43. By similar considerations, the appropriate threshold for Task 43 was set to 20.

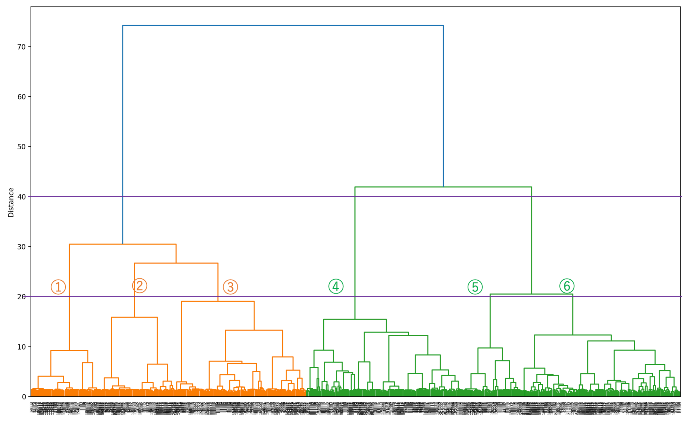

Figure. 3. The dendrogram of Task 43

6.4 Identification of Logic Errors in Task73

Figure 4 presents the dendrogram for Task 73. Task 73 had significantly more source codes than the other two tasks, and multiple source codes were collected for each person. Therefore, even at the threshold of 20, we observed cases where one person formed one cluster.

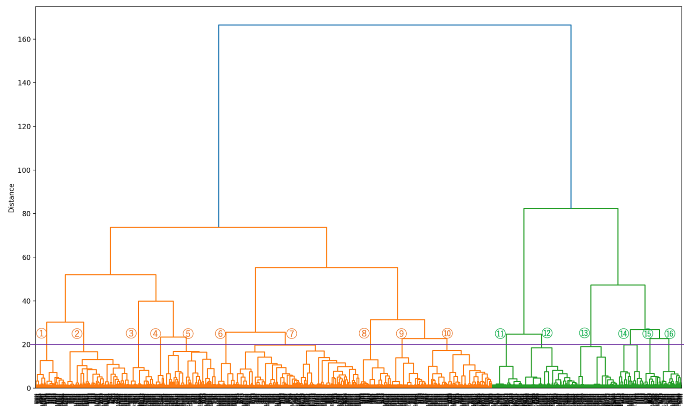

Figure. 4. The dendrogram of Task 73

6.5 Evaluation and Discussion

The evaluation experiment yielded the results shown in Table 4. Due to the inability to create a dataset for Task 73, the evaluation experiment could not be conducted for this task.

Table 4. Indication result of logic errors

Task No.	Conformance rate	Reproducibility rate	F value
35	0.67	0.72	0.69
43	0.92	0.95	0.93
73	-	-	-

The evaluation results suggested that a method capable of identifying logic errors for each coding method could be developed with an accuracy rate of about 70%. To consider how feasible this identification rate is, let us analyze the results. Excluding Task 73, which had the worst evaluation results, Task 35 had an F-score of 0.69. The F-score indicates how correct the logic error identification is. Therefore, it can be understood that even for tasks with poor evaluation results, an identification rate of around 70% can be achieved. Consequently, it is believed that about 70% of the relevant logic errors can be identified when inputting source code containing new learners' logic errors.

While the proposed method is useful in scenarios such as programming exercises, it cannot be applied to new programming tasks. This study's limitation lies in the prerequisite of pre-creating the dataset. Additionally, the method relies on having all coding methods and logic errors available from past source code data. Therefore, to improve the identification accuracy in the future, it is hoped that a method incorporating analysis considering the program editing process will be developed based on this approach.

7. CONCLUSION

In this study, we aimed to identify logic errors for each coding method with multiple coding methods and present the results to instructors. We assumed that the method of program logic coding is reflected in its structure and attempted to analyze programs focusing on their structure. To achieve this, we clustered source codes based on its structure to classify them by coding methods and then tried to classify logic errors within each coding method.

We conducted evaluation experiments in accordance with existing verification methods for the proposed method. As a result, we were able to identify the corresponding logic errors in new students' source code with an accuracy of approximately 70%. Therefore, this result suggests a new possibility for assisting instructors in identifying students' struggles during programming exercises.

In the future, to improve identification accuracy, a method should be developed that considers the editing process of the program in addition to the current method. Additionally, we hope to operate the developed method in practice based on our proposal.

REFERENCES

Ichimura,S., Kafinami, K., Hirano, H. (2013). A Trial to Support Understanding a Programming Exercise Class. *IPSJ Annual Conference*, 54(12), 2518-2527.

Alammary, A., Carbone, A., Sheard, J. (2012). Implementation of a smart lab for teachers of novice programmers. *Proc. ACE 2012*, 121-130.

Singh, R., Gulwani, S., Solar-Lezama, A. (2013). Automated Feedback Generation for Introductory Programming Assignments. Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation, 15-26.

Urakami, S., Nagashima, K., Namiki, M., Knemune, S., Cho, S. (2020). Automatic Detection of Programming Learners' Difficulties. *IPSJ SIG Technical Report*, 2020-CE-154(4), 1-8.

Kawasaki, M., Onami, S. Onuma, R., Nakayama, H., Kaminaga, H., Miyadera, Y., Nakamura, S. (2023). Methods of Analyzing the Exploratory Traces During Programming Exercise for Grasping the Impasses in Construction of Program Logic. *IEICE Technical Report*, 123(184), 23-28.

Ishiwada, K., Morimoto, Y., Nakamura, S., Nakayama, H., Miyadera, Y. (2017). Development of a Method for Analyzing Source Code Editing Processes to Estimate Students' Learning Situations. *IEICE Technical Report*, 116(438), 75-80.

Kawaguchi, S., Sato, Y., Onuma, R., Nakayama, H., Nakamura, S., Miyadera, Y. (2020). An Automatic Learning Situations Estimation System Using AI Technology in Programming Exercise. *IEICE Technical Report*, 119(468), 141-146.

Torres, R., Ludwig, T., Kunkel, J., Dolz, M. (2018). Comparison of Clang Abstract Syntax Trees Using String Kernels.

SciPy. https://scipy.org/ (Access 2022.12.24).