A dissertation submitted in partial satisfaction of the requirements
for the degree of Doctor of Philosophy in Computer Science and Engineering
in the Graduate School of the University of Aizu

Machine Learning-Based Bug Detection and Debugging Support Model

by

Taku Matsumoto

September 2022

The thesis titled

*Machine Learning-Based Bug Detection
and Debugging Support Model*

by

Taku Matsumoto

is reviewed and approved by:

Main referee

Senior Associate Professor

Yutaka Watanobe — Yutaka Watanobe Aug.12.2022 渡部

Professor

Incheon Paik — Incheon Paik Aug. 15, 2022 [illegible]

Professor

Rentaro Yoshioka — [illegible] Aug. 15, 2022 吉岡

Associate Professor

Keita Nakamura — Keita Nakamura, August 9, 2022 中村

Professor Emeritus

Alexander Vazhenin — Alexander Vazhenin, Aug 15, 2022 [illegible]

THE UNIVERSITY OF AIZU

September 2022

External Publications

Major Journal (Refereed)

1. **Taku Matsumoto**, Yutaka Watanobe, and Keita Nakamura, “A Model with Iterative Trials for Correcting Logic Errors in Source Code,” *Applied Sciences*, vol. 11, no. 11, 4755, 2021. [Online]. Available: `https://www.mdpi.com/2076-3417/11/11/4755`. (**related to this dissertation**)

2. Md. Mostafizer Rahman, Yutaka Watanobe, **Taku Matsumoto**, Rage Uday Kiran, and Keita Nakamura, “Educational Data Mining to Support Programming Learning Using Problem-Solving Data,” *IEEE Access*, vol. 10, pp. 26186-26202, 2022.

3. Yutaka Watanobe, Md. Mostafizer Rahman, **Taku Matsumoto**, Rage Uday Kiran, and Penugonda Ravikumar, “Online Judge Systems: Requirements, Architecture, and Experiences,” *International Journal of Software Engineering and Knowledge Engineering*, vol. 32, no.6, pp.917-946, 2022

Major Conference (Refereed)

1. **Taku Matsumoto** and Yutaka Watanobe, “Towards hybrid intelligence for logic error detection,” in Advancing Technology Industrialization Through Intelligent Software Methodologies, Tools and Techniques: Proceedings of the 18th International Conference on New Trends in Intelligent Software Methodologies, Tools and Techniques (SoMeT19), vol. 318, pp. 120-131, IOS Press, 2019. **(Approved as a major conference by GSAAC) (related to this dissertation)**

2. **Taku Matsumoto**, Yutaka Watanobe, Keita Nakamura, and Yunosuke Teshima, “Logic error detection algorithm based on RNN with threshold selection,” in Knowledge Innovation Through Intelligent Software Methodologies, Tools and Techniques: Proceedings of the 19th International Conference on New Trends in Intelligent Software Methodologies, Tools and Techniques (SoMeT20), vol. 327, pp. 76-87, IOS Press, 2020. **(Approved as a major conference by GSAAC) (related to this dissertation)**

Minor Journal (Refereed)

1. Yuki Funayama, Keita Nakamura, Kenta Tohashi, **Taku Matsumoto**, and et al., “Automatic analog meter reading for plant inspection using a deep neural network,” *Artificial Life and Robotics*, 26, 176–186, 2021.

Minor Conference (Refereed)

1. **Taku Matsumoto**, Keita Nakamura, and Keitaro Naruse, “Modeling of driving force generated by rod wheel with single rod on loose soil,” The 22nd Asia Pacific Symposium on Intelligent and Evolutionary Systems (IES2018), December 22, 2018.

2. Yuki Funayama, Keita Nakamura, Kenta Tohashi, **Taku Matsumoto**, and et al., “Automatic analog meter reading for plant inspection,” Proceedings of the 25th International Symposium on Artificial Life and Robotics (ISAROB), 2020.

Contents

List of Figures

List of Tables

List of Abbreviations

AC Accepted.

AOJ Aizu Online Judge.

API application programming interface.

BERT bidirectional encoder representations from transformers.

CE Compile Error.

DL deep learning.

GLUE general language understanding evaluation benchmark.

GPT3 generative pre-trained transformer 3.

GUI graphical user interface.

IDE integrated development environment.

ITP1 Introduction to Programming 1.

LM language model.

LSTM long short-term memory.

LSTM-LM long short-term memory language model.

MLB machine learning-based.

MLE Memory Limited Exceeded.

MLM masked language model.

NLP natural language processing.

NSP next sentence prediction.

OJ online judge.

PE Presentation Error.

PTM pre-trained model.

RE Run-time Error.

RNN	recurrent neural network.
SBFL	spectrum-based fault localization.
Seq2Seq	sequence to sequence.
SPED	structure pattern and error degree.
SVM	support vector machine.
TLE	Time Limited Exceeded.
WA	Wrong Answer.

List of Terms

correct code A compilable source code that meets the specification of a programming task.

incorrect code A compilable source code that does not meet the specification of a programming task.

logic error A bug that does not terminate an executable program abnormally and induces incorrect behavior.

run-time error A bug that terminate an executable program abnormally while the program is running.

syntax error An error that depends on the syntax of the programming language.

Abstract

A logic error is a bug in computer programs that do not terminate the computer programs abnormally and induces incorrect behavior. The conditions under which logic errors occur vary depending on the implementation details and the difference in programming languages. In addition, detecting and correcting logic errors in source code is one of the most difficult tasks because conventional compilers that can detect syntax errors have difficulty detecting logic errors. This problem is not only for novice programmers but also for advanced programmers because it requires programming knowledge, experience, and logical thinking ability to understand the specification of the software.

Software testing is capable of detecting the presence and location of logic errors. Software testing tests whether a program meets the specification based on test cases. A test case is a specification that includes inputs, experimental conditions, test procedures, and outputs obtained from the inputs to evaluate whether the program specification is correct. Logic errors and their locations are detected from the execution paths in each test case. However, these methods cannot provide information for debugging logic errors. Therefore, it is necessary to realize debugging support that detects logic errors and then proposes a fix for them.

Logic error detection methods detect logic errors using software repositories where a large amount of source code is stored to realize debugging of logic errors. The first method is comparison the structure of an incorrect code with a correct code from the repository that meets the specifications of the programming task based on static analysis. The second method is a machine learning model that learns internal parameters using source codes accumulated in a code repository. These methods have shown excellent results in detecting logic errors. However, these methods have some problems in terms of detection performance and the inability to detect multiple logic errors.

This dissertation clarifies problems in existing logic error detection methods and develops a debugging support model which can correct multiple logic errors based on machine learning.

The contributions of this dissertation are as follows:

- Analyzed static analysis and machine learning-based bug detection using the AOJ dataset to develop a hybrid intelligence that accurately detects and corrects logic errors
- Proposed a method that optimizes a threshold that regulates correction candidates detected by machine learning-based bug detection
- Developed a model that detects and corrects multiple logic errors by introducing iterative trials and an editing operation predictor

Chapter 1

Introduction

1.1 Background

Developing bug-free software is one of the most important aspects of software engineering for any application domain because bugs may cause fatal damage to the information society [1]. Programmers need to iterate software development processes such as coding, testing, and debugging to develop bug-free software. In coding, programmers create the source code with editors or integrated development environments (IDEs) using programming languages according to the application. In testing, programmers test whether the source code meets the specification of the software using testing methods. Testing establishes the existence of bugs in the software. Finally, in debugging, programmers identify and fix bugs in the source code based on results in testing until the source code meets the specification of the software [2].To develop bug-free software, programming knowledge, experience, and logical thinking ability are required to understand software specifications [3].

Programming education has become indispensable for fostering engineers who will be active in today's advanced information society. One of the ways to develop programming skills is through coding exercises with repetitive problem-solving. Therefore, many e-learning systems to support programming education have been developed [4–6]. However, because programming is a difficult task that requires programming knowledge, experience, and logical thinking skills, it is easy for learners to stumble, especially when debugging tasks. Among them, logic errors in compilable code are particularly troublesome. The logic error is a token in the source code that causes unexpected behaviors. So, because programming learners may have little programming knowledge and experience, it is not easy for them to correct logic errors in source code. This is a problem faced not only by novice programmers but also by advanced programmers and instructors. If they cannot identify the logic errors in the source code, they will spend more time debugging and may lose their motivation in programming. Therefore, we believe that identifying logic errors in the source code can support programmers.

Debugging is a particularly time-consuming task in software development processes. Programmers may create source codes with bugs such as run-time errors and logic errors. A run-time error is a bug that terminates an executable program abnormally while the program is running. A logic error is a bug that do not terminate an executable program abnormally but induces incorrect behavior. In contrast, a syntax error is an error that depends on the syntax of programming languages. Conventional compilers can detect syntax errors in source code written in programming languages that require compiling the source code. Because conventional compilers cannot detect run-time errors and logic errors, programmers need to identify and localize run-time errors and logic errors by executing programs.

To solve this problem, debugging is based on the results of software testing [7]. Software testing methods can establish the existence of a bug and can locate it. Software testing methods include black-box testing, white-box testing, and others. It tests whether the program is correct based on the test cases created. Many programming education systems use this black-box testing to evaluate program correctness [8, 9]. On the other hand, white-box testing tests a program focused on the internal structure and operation of the program. These methods help to identify bugs. However, they are unable to propose a fix for the location of the bug and how to fix it. Therefore, it is necessary to provide the location of the bug and its proposed fix.

To debug the software, programmers can also use debugging support tools such as debugger [10] and the visualization tool [11]. These tools can visualize the execution path of the program. However, they cannot present the location of bugs in the program and suggested fixes. Moreover, it is not easy for novice programmers to learn debugging support tools [12].

Logic error detection methods have been researched to support debugging logic errors in source code that are difficult to debug using conventional debugging methods. Yoshizawa et al. proposed the structure pattern and error degree (SPED), which can detect logic errors in the source code based on the comparison results if there are correct codes in the software repository that match the structure of the source code to be corrected [13]. On the other hand, Teshima et al. proposed a machine learning-based (MLB) bug detection, which trains the structure of the correct code in the software repository using long short-term memory language model (LSTM-

LM), and presents correction candidates (tokens to be corrected, their locations, and suggested corrections) [14]. This experimental result show that LSTM-LM can detects logic errors. However, since MLB bug detection detects correction candidates probabilistically, the correction candidates may contain not only logic errors but also other tokens. The difference in detection performance between them has not investigated because SPED and LSTM-LM have evaluated them using different datasets and experimental conditions. However, since these methods showed positive results in each experiment, it is necessary to realize logic error detection with hybrid intelligence that makes the most of these methods.

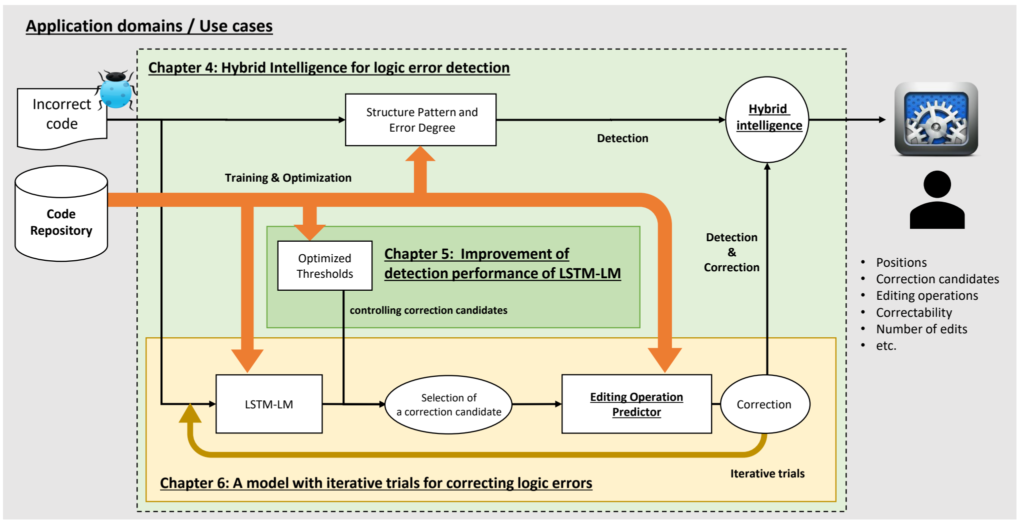
In addition, although MLB logic error detection methods showed their effectiveness in detecting logic errors, they have not been verified to apply to the actual correction of logic errors. In addition, debugging support models have not been developed for correcting logic errors using the correction candidates obtained by the logic error detection method based on LSTM-LM. Therefore, it is necessary to realize a debugging support model based on MLB logic error detection and to clarify the correction performance of these models for correcting logic errors.

1.2 Objectives

The objectives of this dissertation are to clarify the challenges of the hybrid method combining SPED and LSTM-LM and to develop a debugging support model that can detect and correct logic errors using MLB logic error detection. When users use the debugging support model, they themselves need to judge whether the information provided by the debug support model is correct or not. Therefore, we believe that we can reduce the possibility of losing the opportunity for users to develop their logical thinking by enabling users to identify logic errors in the source code they have created from among the correction candidate obtained from the debugging support model. Moreover, it can efficiently support the debugging work of people involved in software development and programming education and prevent the prolonged debugging process and the loss of motivation for programming.

The main objectives of this dissertation are summarized as follows:

- To clarify problems for logic error detection by analyzing the detection performance of existing logic error detection methods focused on the evaluation values based on only detection, no detection, and misdetection.
- To improve the detection performance of MLB logic error detection based on LSTM-LM.



Application domains / Use cases
Chapter 4: Hybrid Intelligence for logic error detection
Incorrect code
Code Repository
Structure Pattern and Error Degree
Detection
Hybrid intelligence
Training & Optimization
Optimized Thresholds
Chapter 5: Improvement of detection performance of LSTM-LM
controlling correction candidates
Detection & Correction
Positions
Correction candidates
Editing operations
Correctability
Number of edits
etc.
LSTM-LM
Selection of a correction candidate
Editing Operation Predictor
Correction
Iterative trials
Chapter 6: A model with iterative trials for correcting logic errors

Figure 1.1: An summary of contributions of this dissertation.

- To develop debugging support model for correcting logic errors using the correction candidates detected by MLB logic error detection based on LSTM-LM.

1.3 Summary of Contributions

The contributions of this dissertation for debugging support in software engineering and programming education (Fig. 1.1), are summarized as follows.

- Analyzed a trade-off between the detection performance and the reliability by comparing the static analysis and machine learning-based model, and that a combination of the methods can be the basis for hybrid intelligence to improve the trade-off.

- Proposed a method that optimizes a threshold value to regulates the number of correction candidates detected by LSTM-LM. The evaluation values used to determine the thresholds are important in order to maximize the detection performance of the LSTM-LM.

- Developed a debugging support model that introduced an editing operation predictor that predicts an editing operation for a correction position. Moreover, this model can provide indirect and non-immediate debugging support that consider the programming skills of both novice and advanced programmers.

1.4 Dissertation Organization

The dissertation is organized as follows.

Chapter 2 introduces our views on related works and approaches of other researchers. This dissertation is concerned with the development of methods for detecting logic errors and debugging support models based on deep learning models in the programming field. First, in Section 2.1, debugging support techniques and the latest research on AI-based bug detection has been introduced. In Section 2.2, we introduce the latest trend in deep learning models in the field of natural languages: language models. In Section 2.3, we introduce the architecture and applications of the latest language models in the programming field. Finally, Section 2.4 introduces AI for debugging based on machine and deep learning-based model.

Chapter 3 introduces an overview of the Aizu Online Judge and the datasets. Section 3.1 describes an overview of AOJ and the importance of AOJ datasets. Section 3.2 describes how to use the source code and metadata stored in the AOJ and how to extract the datasets used for the construction and verification of the proposed method. Section 3.3 outlines the specifications of the programming tasks covered in this paper. Section 3.4 describes the classification of logic errors in each programming task. Finally, Section 3.5 summarizes this Chapter 3.

Chapter 4 describes the detection performance of SPED [13] and LSTM-LM [14] for developing the hybrid intelligence combining the methods. Section 4.1 introduces an overview of the detection by SPED and LSTM-LM. Section 4.2 describes an experiment to analyze the detection performance of each method using the AOJ data set. Section 4.3 shows experimental results that show the strengths and weaknesses of SPED and LSTM-LM. Moreover, in the section, the appropriate basis for developing the hybrid intelligence is also discussed. Finally, Section 4.4 summarizes Chapter 4.

Chapter 5 presents a method that improve the detection performance of LSTM-LM. Section 5.1 presents a method that optimizes thresholds that control correction candidates detected by LSTM-LM. Section 5.2 describes an experiment to evaluate the detection performance by each threshold value. Section 5.3 describes the experimental results that indicate the thresholds can control correction candidates detected by LSTM-LM. Finally, Section 5.4 summarizes Chapter 5.

Chapter 6 proposes a model that detect and correct logic errors iteratively. Section 6.1 introduces an overview of the proposed model. Section 6.2 describes an experiment to evaluate the detection and correction performance of the proposed model. Section 6.3 describes the experimental results that indicate the proposed model can correct multiple logic errors. Finally, Section 6.4 summarizes Chapter 6.

Chapter 7 discusses the integrated debugging support model by combining the proposed approaches. Section 7.1 introduces an integrated debugging support model that combines the proposed approaches. Section 7.2 describes the applications of the integrated debugging support model in education and software engineering. Section 7.3 describes the limitation of the model. Finally, Section 7.4 summarizes Chapter 7.

Chapter 8 concludes the dissertation. It summarizes the contributions and shows the direction of future works.

Chapter 2

Related Work

In this chapter, we present our views on related works and approaches of other researchers. This dissertation is concerned with developing methods for detecting logic errors and debugging support models based on deep learning models in the programming field. First, in Section 2.1, debugging techniques and the latest research on AI-based bug detection have been introduced. Section 2.2 describes the latest trends in deep learning models in the field of natural languages. Section 2 describes the architecture and applications of the latest language models in the programming field. Section 2.4 introduces AI for debugging based on machine and deep learning-based models. Finally, Section 2.5 summarizes Chapter 2.

2.1 Debugging Support

Once again, debugging is to correct bugs from a program revealed when the program is tested [2]. Testing and debugging a program are different processes. Testing establishes the existence of bugs while debugging is concerned with locating and fixing these bugs. When debugging, it is necessary to form hypotheses about the observable behavior of the program and then test these hypotheses to find the defects that caused more than the output. Interactive debugging tools that trace intermediate values of program variables and execution paths are used to support in debugging process.

Gnu GDB [10] and Java Debugger (JDB) [15] have been developed as debuggers that trace the execution path and variables. These debuggers allow the user to check the internal operation of a program sequentially. These debuggers have functions such as breakpoints that pause execution at specified lines, allowing debugging based on such information. However, these debuggers are not provided as a graphical user interface (GUI), making them relatively difficult to use.

We need to see the output results of executing compilable correct source code to detect logic errors. There are tools such as Jeliot [16] and tango [11] that can visualize these behaviors. While these tools are useful for checking the execution status, they require knowledge and experience to use them. Therefore, programmers gain knowledge to use these tools separately, and debugging may take a lot of time.

There are two methods for debugging logic errors: static analysis and dynamic analysis. Static analysis is a method that automatically analyzes the behavior of the source code without executing the source code [17]. Dynamic analysis, on the other hand, is a method of locating bugs by tracing the behavior of the program when the program is running. One such method is Fault localization [18]. These methods specialize in locating bugs, so they cannot show how to fix the identified bugs.

In order to help learners who cannot debug the source code using compilers and IDEs completely, many studies have been conducted to enhance the error messages output by them [19, 20]. These studies reported that by analyzing the source code created by the learner and the error messages produced by the compiler, they were able to reduce the number of syntax errors encountered by the learner. However, since logic errors depend on the specifications of the programming task, they do not have the same rules as the syntax of a programming language. Therefore, debugging logic errors considering the specifications of programming tasks is a major issue.

As one of the debugging techniques, program slicing [21–23] is used to analyze the pattern of a specific bug by checking the dependencies in a program. Program slicing is a method of identifying the location of bugs in source code by subdividing the source code into the smallest units. Program-slicing-based methods are useful for locating bugs, but they cannot provide information on how to fix them.

There is an approach called spectrum-based fault localization (SBFL) that identifies faults in the software based on the execution path of each test case [18, 24]. From the execution path information for each test case, a suspicion level is calculated that indicates the likelihood of being the source of the fault [25]. The higher the suspicion level, the more likely it is that the line of software corresponding to the suspicion level is the fault location. However, while these methods can predict the location of the error, they do not provide any instructions on how to fix the error.

Our Observations

Debugging tools are very good tools for identifying bugs in software. Using debugging tools requires knowledge of how to use debugging tools in addition to the knowledge required to develop the program to be developed. For experienced programmers, debugging tools are routine. On the other hand, it is assumed that it is difficult for novice programmers to acquire the knowledge necessary for software development and the knowledge of debugging tools at the same time. It has been reported that it is difficult for programming novices to handle debugging techniques [12]. This means that learners who aim to learn programming have a hard time acquiring knowledge about debugging tools. Debugging knowledge is acquired through problem-solving iterations and software development experience in programming education. However, knowledge of debugging tools is important for the development of large-scale, high-quality software.

2.2 Deep Learning Models for Natural Language Processing

Until the advent of DL model, statistical-based language model (LM) research had been conducted in the field of natural language processing (NLP). LMs are models for computing probability distributions over a sequence of language tokens. These models take into account the distributed representation of words, and include neural network language model (NNLM) such as the n-gram model [26].

Bengio et al. proposed a probabilistic feed-forward neural network language model (FFNNLM), which can solve the Curse of dimensionality in traditional statistical language model and greatly improve the n-gram model. Mikolov et al. proposed Word2vec, which can generate distributed representations of words using the skip-gram model and the continuous back-of-words (CBOW) model [27].

The RNN has been developed as a deep learning model that can handle time-series data [28]. This model can learn the dependency of time series data by feeding back the past hidden layer state. However, RNN has a problem that the gradient disappears due to long-term time dependence. To solve this problem, long short-term memory (LSTM) blocks were introduced in place of neurons in the hidden layer of RNNs [29].

Sutskever et al. developed sequence to sequence (Seq2Seq), which can transform one input data into another [30]. This model consists of an Encoder and a Decoder, where the Encoder learns the structure of the series data corresponding to the input data, and the Decoder learns another series data corresponding to that series data. For example, Seq2seq can convert an input English sentence into a French sentence.

Vaswani et al. [31] proposed Transformer that can significantly improve learning speed over traditional recurrent layer and convolutional layer based architectures in translation tasks by improving the encoder and decoder in seq2seq. Transformer introduces attention mechanism [32] and positional encoding instead of a recurrent layer. The transformer is able to learn not only the time series dependency of the input data but also the local dependency of the time series data.

The deep learning models introduced so far are trained using data related to a specific task in natural language processing, and thus have the problem that they can only be applied to a specific task. Therefore, pre-trained models (PTMs) are attracting attention as generalized language models that can be applied not only to specific tasks but also to other tasks [33]. Pre-trained models pre-train their own parameters by using unsupervised tasks such as masked language model (MLM) and next sentence prediction (NSP) on large data sets. The trained model is then fine-tuned using a dataset relevant to a particular task to re-modify the model to be able to solve the particular task. These models are used to evaluate the generalization performance of natural language processing comprehension tasks using the general language understanding evaluation benchmark (GLUE) [34], which shows the high performance of the pre-trained models.

Google proposed bidirectional encoder representations from transformers (BERT), which uses the same architecture as the Encoder used in the Transformer [35]. BERT is pre-trained unsupervised using a large amount of text. After the pre-training is done, supervised learning for the target task is done to build a generalized model that can be applied to any task in the field of natural language processing.

Pre-trained models have been widely used to solve NLP tasks [33]. OpenAI proposed generative pre-trained transformer 3 (GPT3) as state-of-the-art MLB in 2020 [36]. The model is capable of generating sentences that are comparable to those written by humans. GPT-3 was realized by using a huge number of parameters of the model and a large amount of training data. It has been suggested that GPT-3 may generate offensive text such as hate speech. However, we are working to improve these problems by developing InstructGPT, which incorporates human feedback into existing machine learning methods [37].

Our Observation

LM based on deep learning in the field of NLP have evolved in an innovative way. These models are very good and can make predictions probabilistically based on training data. However, the correctness of the information predicted by these models is determined by the users of these models. Therefore, we believe that it is important to know how to evaluate whether the information predicted by these models is correct or not.

2.3 Deep Learning Models for Programming

This section describes applications of deep learning models in the field of programming. In recent years, machine learning techniques have been utilized to solve complex programming-related problems [38]. This section presents examples of problem solving in programming.

First, we should emphasize the recent evolution of IDE extensions. Functions such as error highlighting, completion, and refactoring are essential for software development. Such support can be realized by machine learning approaches with data analysis. For example, the latest technology in Visual Studio IntelliCode has had a major impact on software engineering [39]. In this approach, artificial intelligence uses huge GitHub repositories for learning and provides intelligent completion capabilities that take into account not only listing variables and functions but also other situations.

Microsoft proposed CodeBert based on bidirectional encoder representations from transformers (BERT) [40]. CodeBert can perform tasks such as natural language code search, which converts natural language to code, and code document generation.

CodeT5 is a pre-trained model for code understanding and code generation based on T5 [41]. This model outperformed previous researches on understanding tasks such as code defect detection and clone detection.

2.3.1 Application

Particular approaches for identifying and predicting bugs in source codes have already been proposed by various organizations. Among several machine learning approaches, bug detection based on a large corpus of programs [42] and a support vector machine for code clone detection [43] are noteworthy. A comprehensive survey of machine learning for big code has also been conducted [44]. However, [42] demonstrated that bug finding based on machine learning is inferior to static program analysis. On the other hand, other evidence suggests that some machine learning approaches with an n-gram MLB and neural-network-based regression are superior to static program analysis [45, 46]. Therefore, the strengths and weaknesses of these approaches for providing appropriate coding support appear to be context-specific, depending on experimental processes and data volume. How future vectors are created from source codes as well as learning models with appropriate parameters should also be considered.

Chen et. al. proposed GitHub Copilot [47] that generate the source code from the text written in the natural language based on the pre-trained model Codex based on GPT-3 [48]. A large amount of source code from many public GitHub repositories is used to build CodeX.

DeepMind proposed a new attempt, AlphaCode, a code generation model capable of solving competition programming problems [49]. AlphaCode uses a pre-training model based on Transformer, first using source code from a public GitHub repository as pre-training data, and then using CodeForces and AOJ, a website that runs competitive programming contests, as fine-tuning data. AlphaCode achieved an average top ranking of 54.3 percent in a competition with over 5,000 entrants.

2.3.2 Dataset

A number of datasets have been proposed to advance the study of deep learning models for codes. POJ-104 [50] is one of the most pioneering coding datasets collected by the OJ, containing 52,000 codes for 104 programming problems. The POJ-104 dataset also has its limitations, such as lack of useful metadata, overlapping problems, and limited use of programming languages (C and C++) [50]. Another public dataset is related to Google Code Jam (GCJ) which cover a variety of programming languages [51]. However, these datasets are also not sufficient due to data size, number of languages, lack of annotations, metadata, and code pairs. IBM has also released a dataset called CodeNet, which advocates ”AI for code” and contains source code and metadata accumulated in Aizu Online Judge and AtCoder, systems designed for programming learning [52]. CodeNet also provides its special tokenizer to facilitate the use of these data in deep learning (DL) models. Microsoft has also released CodeXGlue, a benchmark dataset for code intelligence [50]. Moreover, CodeXGlue publishes the deep learning models that have been tested and their accuracy, so you can see which model is suitable for each task. CodeXGlue has publicly available the deep learning models that have been tested so far and their accuracy, so you can check which model is suitable for each task.

Our Observation

In this section, we introduce deep learning models and their applications in the programming field. These models have enabled the evolution of language models in the field of natural language processing to solve problems in the field of programming education. In addition, with the development of hardware, the number of systems with large software repositories, such as GitHub, is increasing. Using data stored in these software repositories, AI technology in the programming field is evolving. These models are built using large amounts of training data and a huge number of parameters, making it possible to generate more natural programs. However, they are problematic in that they are very expensive in terms of training costs and time. Finally, the emergence of these models will shorten the time-consuming software engineering process by allowing programs to be generated from natural language.

2.4 AI for Debugging Support

In this section, we introduce a method for debugging support using AI. The AI-based debugging support method is able to propose fixes and correct source code, which is difficult to achieve with conventional debugging support methods.

Many automatic bug fixing methods have been proposed for quickly fixing software bugs [53–55]. Pu et al. proposed sk_p, a program modification method for MOOCs [54]. sk_p is a model based on seq2seq and shows that it can correct 29% of programming tasks using Python. Drain et al. realizes to detect and fix bugs using the standard Transformer [31] as a MLB based on source code extracted from GitHub repositories [56]. Ueda et al. have proposed the fix method by mining the editing histories in GitHub [57]. These methods make it possible to provide debugging support by using source codes, bug reports, edit histories, and so on.

Many methods have been proposed to fix source code using the correction candidates predicted by machine learning models. To correct multiple syntax errors in a source code, Gupta et al. proposed Deepfix [58], which iteratively corrects the errors in the source code. In this method, the source code can be modified line by line using one correction candidate predicted by the Sequence to Sequence (Seq2Seq) attention network. Hajipour et al. [59] proposed Samplefix that iteratively corrects errors using a Seq2Seq model. Each time the source code is modified, these methods use a compiler to verify whether an error still exists. It was shown that the accuracy of correcting syntax errors in the source code can be improved by making repeated corrections. However, although source code verification using a compiler can correct syntax errors, it is difficult to find and correct logic errors. Huang et al. apply Seq2Seq as a model for code correction [60].

Gupta et al. proposed a method for predicting the places of logic errors using the tree convolutional neural network and Abstract Syntax Tree (AST) [61]. The experimental results demonstrated that the accuracy is less than 30 percent if the number of candidate lines is one, and 80 percent for 10 candidate lines. However, the increase of the number of candidate lines makes it difficult to find true logic errors. We should consider the trade-off between misdetection and overdetection.

Vasic et al. proposed a program correction method leveraging a joint model using an LSTM network and an attention mechanism to solve the variable misuse problem [62]. Publicly available data were used to verify the accuracy of identifying and correcting variable misuse points. However, although variable misuse can be considered as one type of logic error, it cannot be used to identify other types of logic errors.

Berabi et al. proposed a transformer-based Tfix [63], which is pre-trained using natural language and fine-tuned using large, high-quality data extracted from GitHub commits to generate code fixes. The model simultaneously fine-tunes for several error types reported by the static analyzer. They evaluated it on a large dataset written in Javascript and found that it was able to produce code that fixed the errors in about 67 percent of the cases, demonstrating the effectiveness of Tfix.

To support the debugging of logic errors by novice programmers in educational scenes, many studies have been conducted using source code groups created to meet the specifications of a certain task. Yoshizawa et al. proposed a static analysis method that considers the structure of the source code [13, 64]. For this, the correct code group is converted into Abstract Syntax Trees (ASTs). The source code to be debugged is also converted to AST. By comparing the structure of the converted AST and the prepared ASTs, the position of the logic error and the type of the logic error can be obtained with high accuracy. In MLB approaches, Teshima et al. proposed a MLB based on LSTM-LM and correct codes [14]. LSTM-LM can indicate the position of logic errors in a given incorrect code and suggest possible words by learning the structure of the correct code. Rahman et al. improved this model by applying the Attention Mechanism with LSTM (LSTM-AttM) [65, 66]. They also employed Bidirectional LSTM to detect logic errors and to present suggestions for corrections [67]. The models can also be applied to code completion [68]. Although the machine learning model used for error detection is different, LED and correction are realized by learning the structure of the correct code.

Our Observation

AI-based bug detection methods can not only locate logic errors in the source code but also suggest fixes. This has been shown by experimental results in the studies presented so far. However, when programmers use these methods, they need to determine whether the information obtained from these methods is correct. Even if the source code can be modified using the predicted results, the modified source code may contain logic errors. Therefore, these techniques do not guarantee that the programmer will be able to correct all logic errors in the source code.

Detecting and fixing bugs in the source code is a very difficult task, and Deepfix and Samplefix can remove more bugs by iteratively detecting bugs in the source code and parsing them by the compiler. However, since these models fix bugs in source codes, they may lose the opportunity to develop learners' logical thinking. Learners and instructors in programming education need to have the opportunity to consider step-by-step what debugging should be done depending on their proficiency in programming skills.

2.5 Chapter Summary

This chapter has introduced deep learning models for natural and programming languages and debugging support methods based on existing bug detection methods. In the software engineering field, there is a need for immediate and direct debugging support, and the emphasis is on how to reduce the time required for debugging. However, in programming education, it is necessary to provide immediate and non-direct support to learners. Direct support may deprive learners of the opportunity to develop logical thinking to solve problems based on the support provided, so debugging support for learners should be provided with caution. If there is too little information used for debugging support, learners may not know how to modify the source code. On the other hand, if there is too much information, the learner may modify the source code without thinking deeply. Therefore, considering the educational effect in programming, the information used for debugging support must be presented carefully.

Chapter 3

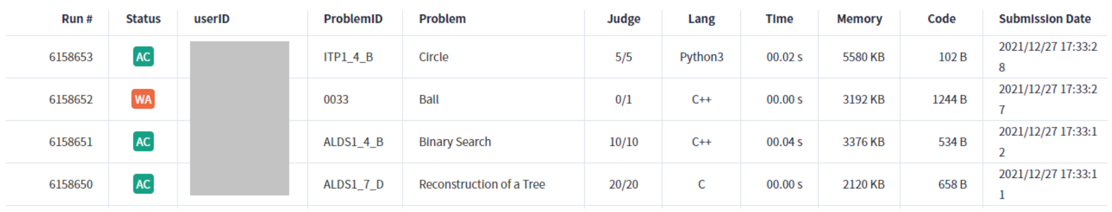
Data Preparation

This chapter introduces an overview of the Aizu Online Judge and the datasets. Section 3.1 describes an overview of AOJ and the importance of AOJ datasets. Section 2 describes how to use the source code and metadata stored in the AOJ and how to extract the datasets used to construct and verify the proposed method. Section 3.3 outlines the specifications of the programming tasks covered in this paper. Section 3.4 describes the classification of logic errors in each programming task. Finally, Section 3.5 summarizes this Chapter 3.

3.1 Aizu Online Judge

AOJ is one of the OJ systems that can automatically judge the source code submitted by learners [8, 69]. AOJ is a system that can be used by a wide range of users, from beginners to experts, and currently has more than 100,000 registered users. This system supports 15 programming languages such as C/C++, Java, and Python. Users can freely choose from about 2,500 problems to challenge themselves. When users submit their solution codes to AOJ, AOJ automatically judges their solution codes. Therefore, users can know whether their source code meets the specifications of the programming task or not.

AOJ rigorously evaluates the submitted source code in the AOJ's Judge Server, which is equipped with an execution environment [70]. The AOJ has test cases for each input and output to verify whether the source code meets the specifications of a programming task. If the source code passes all these test cases, it is evaluated as correct code. On the other hand, if the source code cannot pass even one of these test cases, it is evaluated as incorrect code. In addition, If programming tasks require the use of algorithms such as sorting and graphs, test cases also



Run #	Status	userID	ProblemID	Problem	Judge	Lang	Time	Memory	Code	Submission Date
6158653	AC	[redacted]	ITP1_4_B	Circle	5/5	Python3	00.02 s	5580 KB	102 B	2021/12/27 17:33:28
6158652	WA	[redacted]	0033	Ball	0/1	C++	00.00 s	3192 KB	1244 B	2021/12/27 17:33:27
6158651	AC	[redacted]	ALDS1_4_B	Binary Search	10/10	C++	00.04 s	3376 KB	534 B	2021/12/27 17:33:12
6158650	AC	[redacted]	ALDS1_7_D	Reconstruction of a Tree	20/20	C	00.00 s	2120 KB	658 B	2021/12/27 17:33:11

Figure 3.1: An example of judge results

have constraints such as computation time and memory usage. These strict grading procedures of AOJ make it possible to provide rigorous grading results to users. After the AOJ graded the source code, it accumulated the source code and the metadata, such as the corresponding user information and the judgment result, into a database.

Judge status is strictly evaluated based on the test environment in AOJ as Compile Error (CE), Run-time Error (RE), Wrong Answer (WA), Time Limited Exceeded (TLE), Memory Limited Exceeded (MLE), Presentation Error (PE), and Accepted (AC). CE is determined when the syntax of the source code is incorrect. RE is determined when the source code terminates abnormally. WA is determined when the output to the test case provided by AOJ is different. TLE is determined when the source code does not finish executing within the time limit set in the test environment. MLE is determined when the memory usage exceeds the set amount. PE is determined when the source code meets the specification of the programming task itself, but the output format is incorrect. Finally, AC is determined when the source code meets all the specifications of the test cases for each problem.

AOJ dataset is a necessity for programming education and software engineering problem-solving. The AOJ is widely used by users, from beginners to experts, with different programming knowledge and experience. Since AOJ stores all the source codes created by users, it is possible to observe the process of correcting an incorrect code to a correct code. For these reasons, we believe that the AOJ dataset will be useful for discovering new knowledge in software engineering and programming education by analyzing the relationship between users' abilities and the corresponding source codes.

The AOJ can judge the submitted source code, but cannot provide the learning support such as the debugging. Therefore, AOJ is aiming to develop a new ecosystem by using the data stored in AOJ [71]. This is a new attempt for learners, instructors, and researchers by reusing the accumulated data. Currently, we have been researching fill-in-the-blank problem generation [72], code completion [66,68], bug detection [13,14,67,73], bug clustering [74,75], source code classification [76], problem recommendation [77], and academic performance analysis [78–80] using these accumulated data.

3.2 Extraction of Dataset

AOJ provides the application programming interface (API) to obtain source codes and metadata stored in AOJ [81]. Metadata, which is associated with source code, can be used to extract the source code according to the application. In this dissertation, we use the incorrect and correct codes corresponding to each programming task to construct and evaluate the proposed methods.

We introduce how to extract incorrect and correct codes for each programming task. Figure 3.2 To extract these source codes, the problemID, userID, judge status, and judgeID included in the metadata are used. Firstly, to classify the metadata by each programming task, metadata whose problemID matches the target problemID is extracted. Next, the metadata classified by each programming is categorized by users. If a user has attempted each programming task multiple times, there may be multiple judge data and source codes. Among them, the last source code whose status is AC is extracted as the correct code. On the other hand, the second-to-last source code whose status is neither AC nor CE is extracted as incorrect code. If there are both incorrect and correct codes created by the user, these source codes are extracted as a pair. Finally, by using the judgeIDs corresponding to these source codes, these data sets can be extracted.

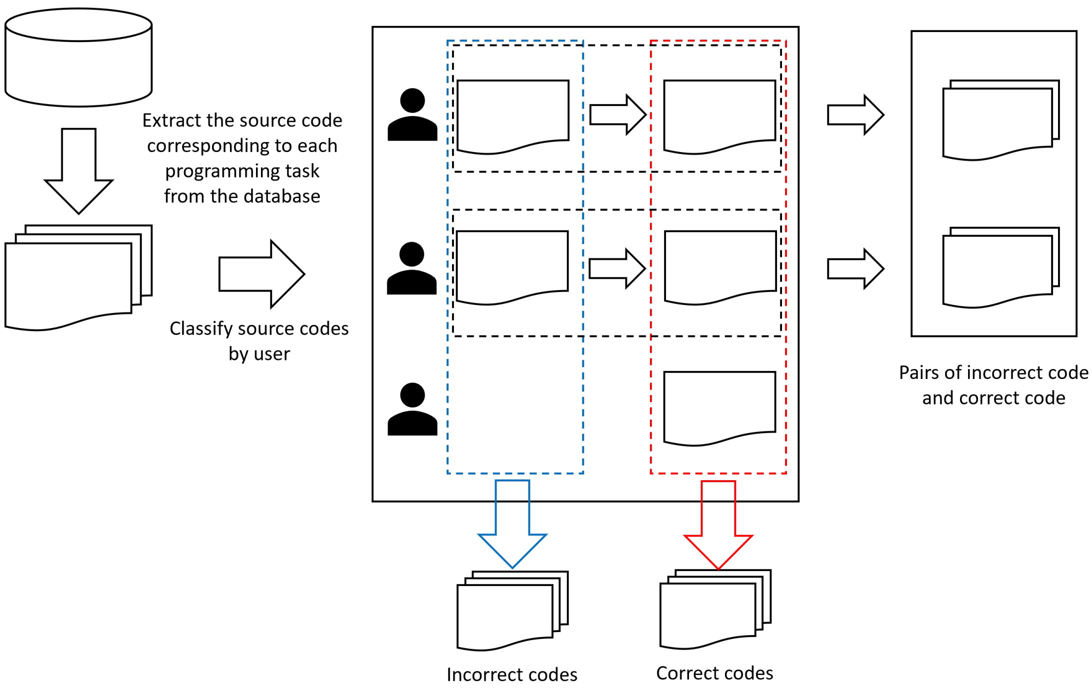

Extract the source code corresponding to each programming task from the database
Classify source codes by user
Pairs of incorrect code and correct code
Incorrect codes
Correct codes

Figure 3.2: An overview of extraction of dataset in AOJ

3.3 Selected Programming Task

Table 3.1 shows the specification details of the 32 selected programming tasks. These tasks are problems in Introduction to Programming 1 (ITP1), a set of problems for beginners in programming. The problems in ITP1 are recommended by the AOJ to be the first ones to be attempted in learning how to use the AOJ and programming.

Table 3.1: Details of 32 selected tasks.

Task ID	Task specification
ITP1_1_A	Outputs ”Hello World” to standard output.
ITP1_1_B	Outputs the cube of a given integer x.
ITP1_1_C	Outputs the area and perimeter of a given rectangle.
ITP1_1_D	Receives an S seconds and converts it to $h : m : s$.
ITP1_2_A	Outputs small/large/equal relation of two given integers a and b.
ITP1_2_B	Receives three integers a, b, and c and outputs ”Yes” if $a < b < c$, otherwise ”No”
ITP1_2_C	Receives integers and outputs them in ascending order.
ITP1_2_D	Receives a rectangle and a circle, and determines whether the circle is arranged inside the rectangle.
ITP1_3_A	Outputs ”Hello World” 1000 times.
ITP1_3_B	Receives an integer x and outputs it as is for multiple test cases.
ITP1_3_C	Receives two integers x and y, and outputs them in ascending order for multiple test cases.
ITP1_3_D	Receives three integers a, b, and c, and outputs the number of divisors of c in $[a, b]$
ITP1_4_A	Receives two integers a and b, and outputs a/b in different types.
ITP1_4_B	Receives a radius r, outputs the area and circumference of a circle.
ITP1_4_C	Receives two integers, a and b, and an operator op, and then outputs the value of $aopb$
ITP1_4_D	Receives a sequence of n integers a_i ($i = 1, 2, \dots n$), and outputs the minimum value, maximum value, and sum of the sequence.
ITP1_5_A	Draws a rectangle which has a height of H cm and a width of W cm. Draws the rectangle by ’#’
ITP1_5_B	Draws a frame which has a height of H cm and a width of W cm.
ITP1_5_C	Draws a chessboard which has a height of H cm and a width of W cm.
ITP1_5_D	Structured programming without goto statement.
ITP1_6_A	Receives a sequence and output
ITP1_6_B	Given a deck of cards, finds any missing cards.
ITP1_6_C	Counts the number of elements in a three-dimensional array.
ITP1_6_D	Receives an $n \times m$ matrix $\boldsymbol{A}$ and an $m \times 1$ vector $\boldsymbol{b}$, and prints their product $\boldsymbol{Ab}$
ITP1_7_A	Receives a list of student test scores and evaluates the performance of each student.
ITP1_7_B	Identifies the number of combinations of three integers which satisfy 1) you should select three distinct integers from 1 to n, and 2) the total sum of the three integers is x.
ITP1_7_C	Receives the number of rows r, columns c, and a table of $r \times c$ elements, and prints a new table, which includes the total sum for each row and column.
ITP1_7_D	Receives an $n \times m$ matrix $\boldsymbol{a}$ and an $m \times l$ matrix $\boldsymbol{B}$, and prints their product, an $n \times l$ matrix $\boldsymbol{C}$.
ITP1_8_A	Converts uppercase/lowercase letters to lowercase/uppercase letters for a given string.
ITP1_8_B	Receives an integer and prints the sum of its digits.
ITP1_8_C	Counts and reports the number of each letter. Ignores characters.
ITP1_8_D	Finds a pattern p in a ring-shaped text s.

Table 3.2 shows the statistics for each programming task. Users are the number of users who attempted each programming task. Submissions are the number of source codes submitted

to solve each programming task. The acceptance rate is the percentage of correct source codes among the Submissions. The acceptance rate is the percentage of correct source code among the submissions. These statistics can be checked for other programming tasks as well.

Table 3.2: Acceptance rate for each programming task.

Task ID	Users	Submissions	Acceptance rate (%)	Task ID	Users	Submissions	Acceptance rate (%)
ITP1_1_A	48673	124971	58.75	ITP1_5_A	17372	68141	31.69
ITP1_1_B	39774	102804	51.13	ITP1_5_B	16430	40159	49.57
ITP1_1_C	34372	105789	41.14	ITP1_5_C	15950	35404	54.47
ITP1_1_D	29368	77964	48.09	ITP1_5_D	11586	61899	22.53
ITP1_2_A	28292	94944	37.56	ITP1_6_A	15762	56824	33.81
ITP1_2_B	56609	26439	56.03	ITP1_6_B	13327	45454	37.93
ITP1_2_C	25061	81707	38.59	ITP1_6_C	12327	49640	29.77
ITP1_2_D	22154	63252	42.55	ITP1_6_D	11036	23756	55.96
ITP1_3_A	24217	49483	61.02	ITP1_7_A	12343	35493	39.89
ITP1_3_B	22012	84443	32.97	ITP1_7_B	13916	41926	42.24
ITP1_3_C	21052	63440	39.99	ITP1_7_C	10614	35466	36.61
ITP1_3_D	19729	44150	53.97	ITP1_7_D	9265	37140	30.72
ITP1_4_A	19061	74999	31.83	ITP1_8_A	10079	25685	48.02
ITP1_4_B	18633	69647	31.79	ITP1_8_B	9938	25726	47.05
ITP1_4_C	17573	50609	41.49	ITP1_8_C	8819	38775	30.03
ITP1_4_D	17312	78313	28.72	ITP1_8_D	8311	18526	53.37

3.4 Logic Errors in Each Programming Task

In cases where there were incorrect and correct codes, it was thought that the correct code could be created by correcting the incorrect code. Therefore, we thought that we could identify the logic error type and its location by analyzing the editing information between the incorrect and correct codes. The editing information includes tokens, their positions, and their editing operations (insertion, deletion, and replacement). We used the *difflib* module in Python, which is one of the frameworks for extracting the edit information, to perform regular expression and string searches on the extracted edit information and source code. We used regular expressions and string searches on the source code and the editing information extracted by *difflib* to identify the parts that were edited between the incorrect and correct codes. Table 3.3 shows the regular expressions that were used to classify the logic errors. Since the source code may contain multiple logic errors, multiple logic error types should be assigned to a single source code.

Tables 3.4, 3.5, 3.6, and 3.7 show the distribution of logic errors present in the source code of each programming task. Exclusion is the number of source codes created to solve different programming tasks. Outlier excludes cases where the edit distance between incorrect and correct codes is extremely long within the target programming task. No categorized is the number of source codes that did not match any of the regular expressions introduced in Table 3.3.

Table 3.3: Regular expressions used to classify logic errors.

Logic error type	regular expression
include_statement	\#+?\s*?include\s*?[\"<].+?[\">]\s*;? | \#+?\s*?import\s*?[\"<].+?[\">]\s*;?
switch_quote	(?<!\\)\" | (?<!\\)\'
string_format	(?!\\)\".+?(?<!\\)\" | (?!\\)\'.+?(?<!\\)\'
output_format	printf\s*?\(.+?\)\s*?;+(?!”) | (?:std\s*?::\s*?)?cout.*?;+(?!”) | puts\(.+\)\s*?;+(?!”)
input_statement	(?:std\s*?::\s*?)?cin.+?;+ | scanf\(.+?\);+
void_main_function	(?:int\s+)?main\s*?\(.*?\)\s* | (?:void\s+)?main\s*?\(.*?\)\s*
static	\wstatic\s+?
return_value	return\s*?.*?;
for_statement	(?<=\W)for\s*?\(
if_statement	(?<=\W)(?:else)?\s+?if\s*?\(
else_statement	(?<=\W)else
formula	(?<=\W)[a-zA-Z_][\[\w\]-]*?\s*?[\+\-*\/\%\&\ |]?=\s*?.+?;
do_while_statement	do\s*?.+?\s*?while\s*?\(
while_statement	(?<=\W)while\s*?\(
function	\W[a-zA-Z_]\w*(?<!main | for | while | if | return)\s*?\(
address_operator	&\s*?\w\w*
cast	\(\s*?(?:double | int | (?:long\s+?)long | short | char | float)\s*?\)
type	\W(?:int64_t\W | double\int | long | short | char | float | unsigned)\W
use_and_or_operator	\&\& | \ | \ |
array_size	(?<!>\s*?)\[
initial_value	(?<!\w)\s*?[+-]?\d+\.?\d+?\s*?;
variable_declaration	(?:const)?\s*?(?:int_64_t | double | int | (?:long\s+?)long | short | char | float | unsigned | bool) \s+?[\w,=\s\[\]*]*?;+?
unary_operation	[a-zA-Z_]\w*?\s*?\+\+; | \+\+\s*?[a-zA-Z_]\w*?; | [a-zA-Z_]\w*?\s*?–; | – \s*?[a-zA-Z_]\w*?;+?;
scope	\{ | \}
substituted_variable	(\w\w*?)\s*?=(?==)
switch_break_continue	break\s*?; | continue\s*?;
case_number	case\s+?.+?:
semicolon_position	[a-zA-Z_]\w*?\s*?;\s*?\(| else\s*?; | if\s*?\(.+?\)\s*?; | else\s+?if\s*?\(.+?\)\s*?; | for\s*?\(.+?);—while\s*?\(.+?\);
switch_equal_assign	==
array_elements	=\s*?\{[\-\d\.\s,]*?\}
assignment_operation	[\+\-*\/\%\ | \&]=(?!=)

However, the results with these regular expressions that we show now are not yet complete. The logic errors shown in Table 3.4 should be further subdivided, and the authors are aware of this. This paper does not deal with the classification and analysis of these logic errors in detail, but we believe that this information will be an important part of programming education and debugging support in the field of software engineering. We plan to work on these classification and analysis as future work.

Table 3.4: The location of logic errors that occurs in each programming (from ITP1_1_A to ITP1_2_D).

Logic error types	ITP1_1_A	ITP1_1_B	ITP1_1_C	ITP1_1_D	ITP1_2_A	ITP1_2_B	ITP1_2_C	ITP1_2_D
include_statement	142	62	23	27	13	18	28	21
switch_quote	4	322	135	51	78	124	159	101
string_format	1870	1151	1104	668	1102	505	468	280
output_format	34	486	502	175	190	105	243	96
input_statement	1	332	178	118	79	89	77	66
void_main_function	193	52	23	16	5	7	21	13
static	0	0	0	0	0	0	0	0
return_value	266	108	44	37	21	27	27	24
for_statement	0	15	0	0	1	0	73	1
if_statement	0	13	4	26	90	669	415	577
else_statement	0	3	0	5	26	73	64	59
formula	0	273	132	284	29	5	279	40
do_while_statement	0	0	0	2	0	0	3	0
while_statement	0	3	0	5	1	1	17	1
function	1	11	3	4	7	11	8	4
address_operator	0	0	0	0	0	0	2	2
cast	0	0	0	0	0	0	0	0
type	4	61	47	17	5	3	43	17
use_and_or_operator	0	0	0	1	0	0	2	2
array_size	0	3	0	0	3	1	26	1
initial_value	0	1	1	3	0	0	0	0
variable_declaration	3	225	124	109	29	32	116	49
unary_operation	0	0	1	3	0	1	4	1
scope	0	29	5	17	34	109	265	94
substituted_variable	0	0	0	0	0	0	0	0
switch_break_continue	0	0	0	0	0	0	0	0
case_number	0	0	0	0	0	0	0	0
semicolon_position	0	0	0	6	2	2	29	1
switch_equall_assign	0	0	0	1	0	0	0	0
array_elements	0	0	0	0	0	0	0	0
assignment_operation	0	0	0	0	0	0	0	0
Exclusion	2	17	10	8	10	12	89	15
Outlier	53	49	36	35	34	30	34	27
No categorized	0	0	0	0	0	0	0	0

Table 3.5: The location of logic errors that occurs in each programming (from ITP1_3_A to ITP1_4_D).

Logic error types	ITP1_3_A	ITP1_3_B	ITP1_3_C	ITP1_3_D	ITP1_4_A	ITP1_4_B	ITP1_4_C	ITP1_4_D
include_statement	8	27	18	11	24	53	16	dd
switch_quote	17	74	141	44	32	22	146	18
string_format	304	432	302	307	313	385	266	350
output_format	27	201	219	75	235	249	76	28
input_statement	8	158	112	54	77	46	88	40
void_main_function	8	19	16	12	12	5	10	3
static	0	0	0	0	0	0	0	0
return_value	36	35	35	25	19	11	31	4
for_statement	454	177	63	244	6	0	39	26
if_statement	12	173	346	164	9	0	137	50
else_statement	7	32	116	7	0	0	31	2
formula	44	242	208	118	399	229	119	152
do_while_statement	2	21	18	1	0	0	5	1
while_statement	56	132	116	32	1	1	98	4
function	1	5	3	3	1	0	16	3
address_operator	0	0	0	0	0	0	0	0
cast	0	0	0	0	0	0	0	0
type	9	57	31	32	240	286	42	253
use_and_or_operator	0	0	0	0	0	0	0	0
array_size	0	52	16	1	0	0	16	12
initial_value	1	0	0	0	0	0	0	0
variable_declaration	24	128	105	73	137	119	94	203
unary_operation	22	117	7	20	0	0	7	3
scope	31	147	206	63	9	1	134	26
substituted_variable	0	0	0	0	0	0	0	0
switch_break_continue	0	4	2	4	0	0	2	0
case_number	0	1	0	0	0	0	12	0
semicolon_position	7	14	9	8	0	0	4	1
switch_equall_assign	0	0	0	0	0	0	0	0
array_elements	0	0	0	0	0	0	0	0
assignment_operation	0	0	0	0	0	0	0	0
exclusion	5	9	27	9	7	12	9	22
Outlier	1	15	11	9	8	18	12	21
No categorized	0	0	0	0	0	0	0	0

Table 3.6: The location of logic errors that occurs in each programming (from ITP1_5_A to ITP1_6_D).

Logic error types	ITP1_5_A	ITP1_5_B	ITP1_5_C	ITP1_5_D	ITP1_6_A	ITP1_6_B	ITP1_6_C	ITP1_6_D
include_statement	13	13	13	12	8	34	9	28
switch_quote	422	187	175	85	336	185	63	62
string_format	440	283	210	104	441	290	143	104
output_format	384	176	164	103	402	181	82	102
input_statement	88	54	50	17	58	146	18	113
void_main_function	10	7	9	8	6	20	10	15
static	0	0	0	0	0	8	3	5
return_value	15	10	8	6	10	8	3	7
for_statement	87	90	80	37	164	170	45	132
if_statement	99	154	176	81	206	157	120	19
else_statement	10	39	60	43	86	23	10	3
formula	87	67	78	79	60	188	156	157
do_while_statement	10	4	1	19	2	0	0	0
while_statement	66	28	25	57	10	18	0	3
function	22	18	21	4	17	35	9	6
address_operator	0	0	0	0	0	0	1	0
cast	0	0	0	0	0	2	0	0
type	13	10	17	18	16	83	11	49
use_and_or_operator	0	0	0	1	0	0	1	0
array_size	8	1	4	1	30	79	17	66
initial_value	0	0	0	0	0	0	0	0
variable_declaration	43	33	48	86	46	130	25	68
unary_operation	4	2	12	3	4	13	1	3
scope	106	106	111	79	171	127	97	74
substituted_variable	0	0	1	1	0	0	0	0
switch_break_continue	0	0	1	3	0	3	1	0
case_number	0	1	2	0	0	18	0	0
semicolon_position	4	0	2	8	7	9	6	7
switch_equall_assign	0	0	0	0	0	0	0	0
array_elements	0	0	0	0	0	0	0	0
assignment_operation	0	0	0	0	0	0	20	0
exclusion	18	20	13	51	13	28	22	8
Outlier	20	14	7	3	14	10	16	11
No categorized	0	0	0	0	0	0	0	0

Table 3.7: The location of the logic errors that occurs in each programming (from ITP1_7_A to ITP1_8_D).

Logic error types	ITP1_7_A	ITP1_7_B	ITP1_7_C	ITP1_7_D	ITP1_8_A	ITP1_8_B	ITP1_8_C	ITP1_8_D
include_statement	8	13	11	19	18	58	22	6
switch_quote	157	34	156	44	169	155	121	31
string_format	193	80	212	271	213	160	161	62
output_format	147	92	184	58	138	66	45	24
input_statement	39	65	41	32	65	133	121	11
void_main_function	6	5	7	11	7	10	5	2
static	0	1	12	1	0	0	1	0
return_value	4	10	1	1	7	12	6	3
for_statement	8	156	120	38	42	112	55	24
if_statement	304	118	123	50	82	128	75	32
else_statement	66	11	41	18	28	16	7	8
formula	135	338	148	81	77	187	103	35
do_while_statement	5	7	0	0	5	6	7	0
while_statement	28	59	3	1	35	87	96	6
function	5	2	13	6	41	21	39	30
address_operator	0	0	0	0	0	1	0	0
cast	0	0	0	0	0	0	1	0
type	12	52	34	237	28	130	35	18
use_and_or_operator	0	1	0	0	0	1	0	0
array_size	4	18	57	22	44	129	44	44
initial_value	0	0	0	0	0	0	0	0
variable_declaration	41	82	56	137	43	134	51	29
unary_operation	6	23	2	1	12	37	11	7
scope	148	129	127	50	59	107	90	18
substituted_variable	0	0	0	0	0	0	0	0
switch_break_continue	0	2	0	0	1	3	2	0
case_number	0	0	0	0	0	0	0	0
semicolon_position	4	6	2	3	6	8	14	2
switch_equall_assign	0	0	0	0	0	0	0	0
array_elements	0	0	0	0	0	0	0	0
assignment_operation	0	0	1	0	0	0	0	0
exclusion	23	28	9	19	27	23	15	11
Outlier	16	11	10	12	8	2	3	6
No categorized	0	0	0	0	0	0	0	0

3.5 Chapter Summary

In this chapter, we introduced the AOJ dataset, which has been used by a wide range of users from beginners to advanced users to learn programming skills. The AOJ dataset can be used by using the API provided by AOJ. The programming task dealt with in this dissertation is ITP1, which is a problem set aimed at novice programmers. Although the programming task within ITP1 is simple, it is difficult to correct an incorrect code because it may contain multiple logic errors. We believe that detecting and correcting these logic errors will facilitate the identification of some implementation errors in modules in small and large software developments. Chapter 4 and the following chapters describe the analysis and development using the AOJ dataset.

Chapter 4

Hybrid Intelligence for Logic Error Detection

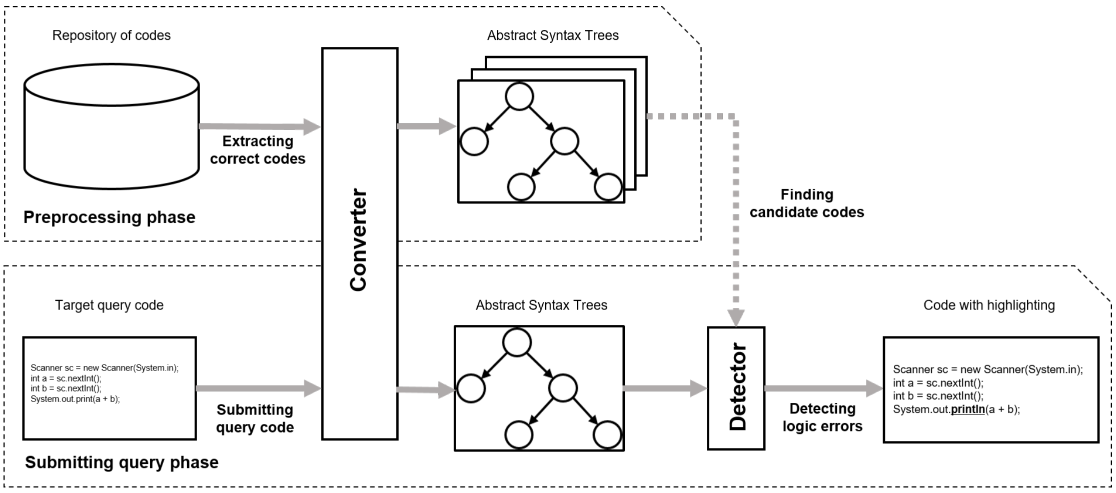
In Chapter 4, we analyze the detection performance of SPED [13] and LSTM-LM [14] for developing the hybrid intelligence combining the methods. Section 4.1 introduces an overview of the detection by SPED and LSTM-LM. Section 4.2 describes an experiment to analyze the detection performance of each method using the AOJ data set. Section 4.3 shows experimental results that show the strengths and weaknesses of SPED and LSTM-LM. Moreover, in the section, the appropriate basis for developing the hybrid intelligence is also discussed. Finally, Section 4.4 summarizes Chapter 4.

4.1 Overview of Algorithms for Logic Error Detection

In this section, we introduce overviews of SPED and LSTM-LM for logic error detection. Figure 4.1 shows overviews of logic error detection using the SPED and the LSTM-LM. Although the details of these algorithms and experimental results are available from [13] and [14], respectively, in this section, we introduce general concepts and algorithms for understanding the experiment and discussion presented in the following section.

4.1.1 Structure Patterns and Error Degree (SPED)

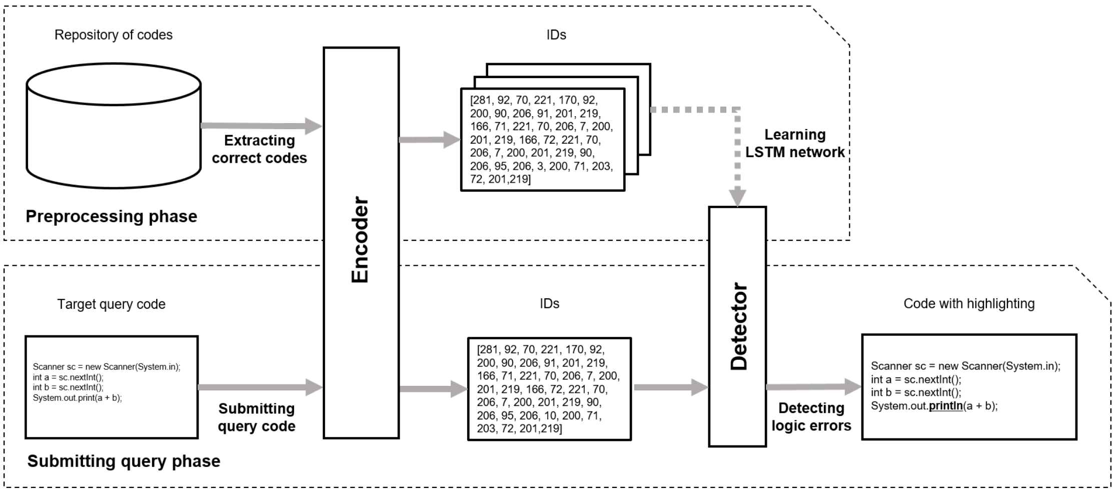
The first approach is based on a static analysis of source codes which use SPED. We have proposed a logic error detection algorithm based on structure patterns, which are an index of similarity based on AST, and error degree, which is a measure of appropriateness for feedback.



```
Scanner sc = new Scanner(System.in);
int a = sc.nextInt();
int b = sc.nextInt();
System.out.print(a + b);
```

```
Scanner sc = new Scanner(System.in);
int a = sc.nextInt();
int b = sc.nextInt();
System.out.println(a + b);
```

(a) Overview of the SPED.



```
[281, 92, 70, 221, 170, 92,
200, 90, 206, 91, 201, 219,
166, 71, 221, 70, 206, 7, 200,
201, 219, 166, 72, 221, 70,
206, 7, 200, 201, 219, 90,
206, 95, 206, 3, 200, 71, 203,
72, 201,219]
```

```
Scanner sc = new Scanner(System.in);
int a = sc.nextInt();
int b = sc.nextInt();
System.out.print(a + b);
```

```
[281, 92, 70, 221, 170, 92,
200, 90, 206, 91, 201, 219,
166, 71, 221, 70, 206, 7, 200,
201, 219, 166, 72, 221, 70,
206, 7, 200, 201, 219, 90,
206, 95, 206, 10, 200, 71,
203, 72, 201,219]
```

```
Scanner sc = new Scanner(System.in);
int a = sc.nextInt();
int b = sc.nextInt();
System.out.println(a + b);
```

(b) Overview of the LSTM-LM.

Figure 4.1: Overview of logic error detection using the SPED and the LSTM-LM.

As preprocessing, available source codes are converted into ASTs to represent their structure patterns through the converter. When a source code is given (submitted), the code is converted into an AST. The algorithm then finds source codes which have the same structure patterns as a given target source code. The detector then indicates the locations of errors according to error degree, which is defined by the impact factors of each element. Finally, the algorithm selects the optimal source code which has the highest error degree among the selected source codes with the same structure pattern. In addition, we developed a logic error detection application programming interface (API) based on the algorithm. In the previous study, the experimental results demonstrate that the API could properly detect logic errors in given target codes for a number of problems in an introduction to programming course [13].

4.1.2 Long Short-Term Memory Language Model (LSTM-LM)

The second approach is based on deep learning. We designed a model which calculates probabilities for the appearance of program elements based on an LSTM-LM. An LSTM-LM is a special kind of RNN which is superior for sequential prediction as compared to other kinds of neural networks. As preprocessing, available source codes are converted into sequences of IDs and these are used for the learning process to create the model of the LSTM-LM. When a source code is given (submitted), the model outputs a probability to show a detected logic error. In the proposed approach, to create the model, before training and evaluation, a source code is divided into a sequence of words. Then, we encode each word into an ID based on the mapping rules defined. In the conversion, as preprocessing, a source code is cleaned (deletion of comments, feed lines, tabs, spaces, etc.) and then divided into a sequence of method names, variable names, keywords, and characters. Since LSTM-LM have some hyperparameters, we should investigate the best model for logic error detection. We should also explore the number of units in a hidden layer which provide the best results in terms of perplexity and training time. In the previous study, the experimental results showed that the model trained by solution codes for a task related to the basic sorting algorithm could detect logic errors in given target codes to a high degree of accuracy [14].

The language model is constructed by RNN which can learn features of sequential data [82]. The language model consists of the following layers:

- Embedding layer
- RNN (LSTM) layer
- Softmax layer

A series of tokens in a program is transformed into a series of IDs $\boldsymbol{x}= [x_1, x_2, \ldots, x_t, \ldots, x_n]$ which can be treated as sequential data. n is the number of tokens in the series of IDs $\boldsymbol{x}$. The model learns a series of token patterns which include IDs related to variables, functions, reserved keywords, etc.

In the embedding layer, the obtained sequences of IDs $\boldsymbol{x}$ is transformed into the corresponding dense vector $\boldsymbol{e}=[e_1, e_2, \ldots, e_t, \ldots, e_n]$. The size of the vector $\boldsymbol{e}$ is equal to the size of the vocabulary table. The learning process is performed by importing the vector $\boldsymbol{e}$ to the RNN layer. Finally, the obtained output $\boldsymbol{h}_t$ (Eq. 4.1) from the learned RNN is transformed into the probability distribution $\boldsymbol{p}$ (Eq. 4.2) for each word in the Softmax layer.

$$\boldsymbol{h}_t = RNN(\boldsymbol{e}_t, \boldsymbol{h}_{t-1}) \tag{4.1}$$

$$\boldsymbol{p}_t = softmax(\boldsymbol{h}_t) \tag{4.2}$$

The model infers a token x_{t+1} from a sequence $[x_1, x_2, .., x_t]$ with the probability. If the probability is high, we consider that the token x_{t+1} is a pattern that tends to appear in the learning data. On the other hand, if the probability is low, we evaluate that the token x_{t+1} is a pattern that rarely appears in the data. So, after a threshold τ is defined, a token that has a probability of less than τ can be extracted as possible logic errors.

4.2 Experiment

In order to investigate the detection performance of these methods, experiments were conducted to verify the presence of logic errors in the correction candidates detected by SPED and LSTM-LM. As validation data, 20 incorrect codes submitted for one problem in the AOJ dataset are used. SPED and LSTM-LM output detection results for the logic errors in each source code input. The detection results were evaluated based on the detected, undetected, false positive, and detection time of the logic errors contained in each source code. These performances indicate the reliability of the detection results obtained from each method.

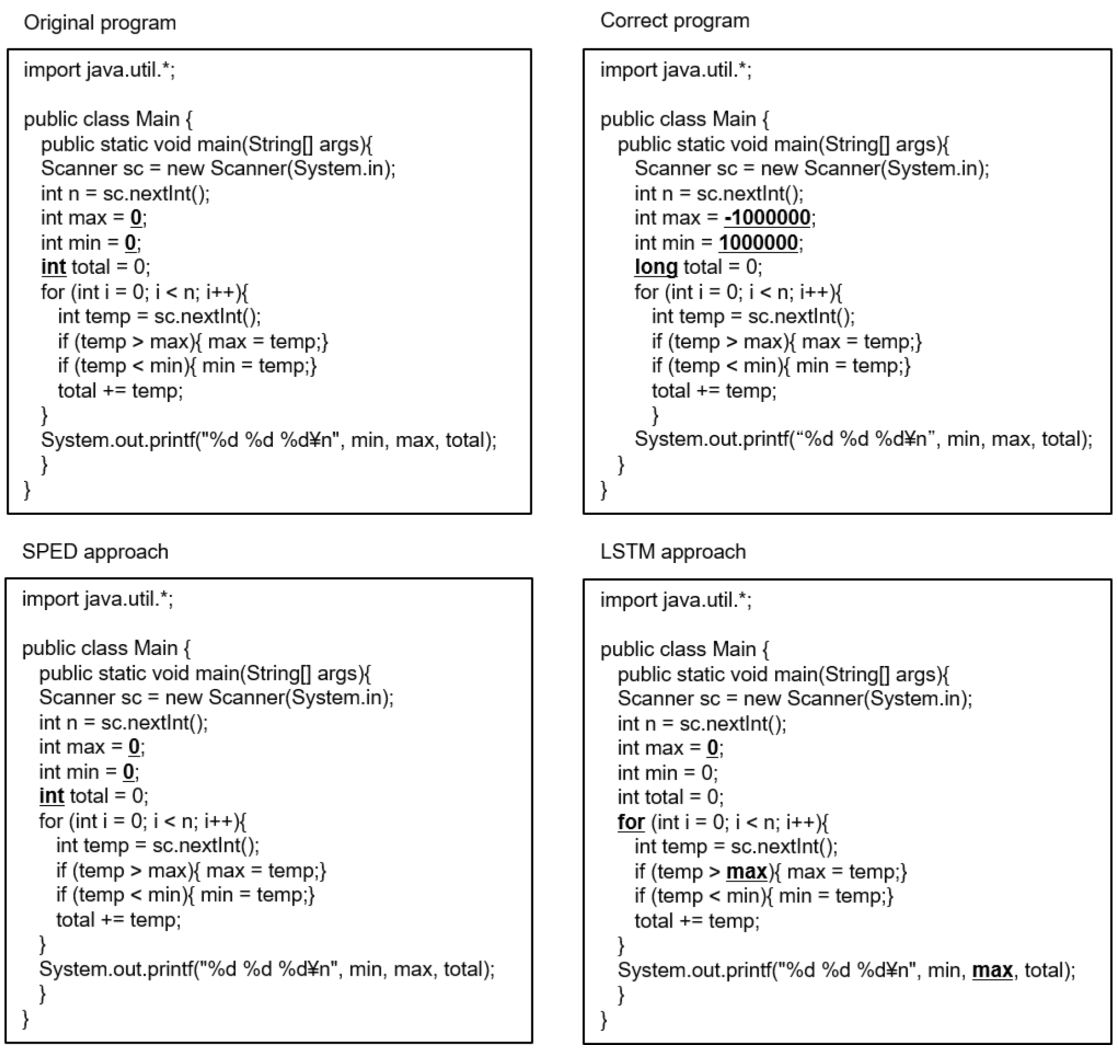
4.2.1 Materials

In this study, we conducted an experiment using the data in the AOJ [8,69], which is one of the major OJ systems and has around 90,000 registered users and more than 6 million judged source codes. In order to compare and investigate the weakness and strength of the algorithms, a common problem (task) was selected, and source codes submitted for solving this task were used for the experiment. The task was selected from the Introduction to Programming course, and the specification (goal) of the task is "For given N integers a_i $(i = 1, 2, ..., N)$, print the minimum value, maximum value and the sum of them". Note that the constraints for the task include $0 < N <= 10{,}000$ and $-1{,}000{,}000 <= a_i <= 1{,}000{,}000$. For example, if the given

Table 4.1: Keywords and characters encoded into IDs.

ID	word	ID	word	ID	word	ID	word	ID	word
140	abstract	169	native	198	&	227	D	256	b
141	assert	170	new	199	’	228	E	257	c
142	boolean	171	package	200	(	229	F	258	d
143	break	172	private	201	)	230	G	259	e
144	byte	173	protected	202	*	231	H	260	f
145	case	174	public	203	+	232	I	261	g
146	catch	175	return	204	,	233	J	262	h
147	char	176	short	205	-	234	K	263	i
148	class	177	static	206	.	235	L	264	j
149	const	178	strictfp	207	/	236	M	265	k
150	continue	179	super	208	0	237	N	266	l
151	default	180	switch	209	1	238	O	267	m
152	do	181	synchronized	210	2	239	P	268	n
153	double	182	this	211	3	240	Q	269	o
154	else	183	throw	212	4	241	R	270	p
155	enum	184	throws	213	5	242	S	271	q
156	extends	185	transient	214	6	243	T	272	r
157	final	186	try	215	7	244	U	273	s
158	finally	187	void	216	8	245	V	274	t
159	float	188	volatile	217	9	246	W	275	u
160	for	189	while	218	:	247	X	276	v
161	goto	190		219	;	248	Y	277	w
162	if	191	!	220	<	249	Z	278	x
163	implements	192	?	221	=	250	[	279	y
164	import	193	_	222	>	251	¥	280	z
165	instanceof	194	”	223	@	252	]	281	{
166	int	195	#	224	A	253	ˆ	282	|
167	interface	196	$	225	B	254	‘	283	}
168	long	197	%	226	C	255	a	284	˜

integers are {5, 2, 12, 8, 7}, then the program should output {2, 12, 34}. We selected this problem because it is one of the most error-prone problems in the course and can be solved by a variety of approaches. More than 40,000 source codes in various programming languages have been submitted for the task, but we have extracted source codes in Java for use in the experiment. In the experiment, 687 correct codes (codes without logic errors) were used for the learning model of the LSTM-LM and for candidate codes for the SPED. For target query codes, 20 incorrect codes (codes with logic errors) were randomly selected from the submission history for this task.



Original program

```
import java.util.*;

public class Main {
  public static void main(String[] args){
  Scanner sc = new Scanner(System.in);
  int n = sc.nextInt();
  int max = 0;
  int min = 0;
  int total = 0;
  for (int i = 0; i < n; i++){
    int temp = sc.nextInt();
    if (temp > max){ max = temp;}
    if (temp < min){ min = temp;}
    total += temp;
  }
  System.out.printf("%d %d %d¥n", min, max, total);
  }
}
```

Correct program

```
import java.util.*;

public class Main {
  public static void main(String[] args){
    Scanner sc = new Scanner(System.in);
    int n = sc.nextInt();
    int max = -1000000;
    int min = 1000000;
    long total = 0;
    for (int i = 0; i < n; i++){
      int temp = sc.nextInt();
      if (temp > max){ max = temp;}
      if (temp < min){ min = temp;}
      total += temp;
      }
    System.out.printf("%d %d %d¥n", min, max, total);
  }
}
```

SPED approach

```
import java.util.*;

public class Main {
  public static void main(String[] args){
  Scanner sc = new Scanner(System.in);
  int n = sc.nextInt();
  int max = 0;
  int min = 0;
  int total = 0;
  for (int i = 0; i < n; i++){
    int temp = sc.nextInt();
    if (temp > max){ max = temp;}
    if (temp < min){ min = temp;}
    total += temp;
  }
  System.out.printf("%d %d %d¥n", min, max, total);
  }
}
```

LSTM approach

```
import java.util.*;

public class Main {
  public static void main(String[] args){
  Scanner sc = new Scanner(System.in);
  int n = sc.nextInt();
  int max = 0;
  int min = 0;
  int total = 0;
  for (int i = 0; i < n; i++){
    int temp = sc.nextInt();
    if (temp > max){ max = temp;}
    if (temp < min){ min = temp;}
    total += temp;
  }
  System.out.printf("%d %d %d¥n", min, max, total);
  }
}
```

Figure 4.2: An example of logic error detection by two approaches for a given target query code

4.2.2 Experimental Setup

The logic error detection API and the corresponding experimental system are used to test the logic error detection based on the SPED. This algorithm manages accumulated source codes submitted to the AOJ with their verdict records. Through the experimental system, we can submit a query code, and the system finds source codes which have the same structure pattern as the query code. The system then tries to indicate the locations of logic errors depending on the error degree calculated for each element. Thus, in this experiment, we use the logic error detection API with accumulated source codes of the target problem to obtain results from the SPED.

On the other hand, for the LSTM-LM, since the previous study is oriented to the C programming language, we adjusted the model by expanding the encoding method for the LSTM-LM to learn and test source codes in Java. Table 4.1 shows mapping rules between keywords and

Table 4.2: Features of target query codes used in the experiment.

Query code #	NoSP	NoE	NoV	NoM	NoO	NoLE
1	7	11	5	3	6	1
2	17	19	6	2	3	2
3	4	18	7	2	5	1
4	6	13	6	4	6	1
5	9	23	8	2	4	1
6	4	15	7	2	4	2
7	6	13	7	4	5	1
8	7	11	5	3	5	1
9	2	23	8	2	4	1
10	10	20	7	2	4	1
11	0	25	7	3	4	1
12	2	14	7	4	5	2
13	7	11	5	3	5	4
14	3	20	7	4	6	1
15	1	7	5	15	9	2
16	3	21	7	2	4	1
17	1	21	7	2	4	1
18	0	15	4	4	5	1
19	1	26	7	3	4	1
20	8	12	5	4	6	1

the corresponding encoded IDs. Method names are encoded into IDs 0 to 69. Variable names are encoded into IDs 70 to 89 in order of appearance. Other names, such as class names and fields, are encoded into IDs 90 to 139. Keywords and characters are encoded into IDs defined in Table 4.1. Layers of the LSTM-LM are constructed based on [14]. We regularize the LSTM-LM by dropout. We determined that the dropout ratio is 0.5 [83]. An Adam optimizer with four hyperparameters, $\alpha = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$, and $\epsilon = 1e - 8$, was used based on the recommendation of [84]. As shown in the mapping rules, each source code is divided into elements from 294 words. Thus, the number of units in the input and output layers is 294. We use a 400-unit model in the hidden layer.

Figure 4.2 shows an example of logic error detection used to explain the experiment. The top-left code denotes a given target query code (original code) with some real logic errors indicated by underlines. The top-right code is the correct code. The bottom-left code is provided by the SPED with predicted logic errors. This example shows that the SPED could detect all real logic errors properly. The bottom-right code is provided by the LSTM-LM with predicted logic errors. The prediction shows that the LSTM-LM indicated one real logic error but missed another. In addition, the LSTM-LM indicated several excessive places which should not be pointed out as logic errors.

4.3 Results and Discussion

4.3.1 Results

Table 4.2 shows the features of target query codes used for the experiment. We consider the structural characteristics of the target code, taking into account the number of available structure patterns as well as the numbers of variables and functions. The number of available candidate codes which have the same structure pattern as the target query code is denoted NoSP. The larger this number is, the more common structure the target code contains. The number of elements (nodes) in a program code (the corresponding abstract syntax tree) is denoted as NoE. A larger number indicates a more complex program code. The numbers of variables and methods in each query code are denoted as NoV and NoM, respectively. The number of other elements, such as class names, class variables, and fields, is denoted as NoO. Larger numbers indicate greater complexity of the program code. The number of real logic errors in each query code is denoted as NoLE.

Table 4.3 shows the results of the experiment for the selected target query codes. The time to detect logic errors for each query code (without preprocessing and learning process) is denoted as DT. The number of places that the algorithm could correctly point out is denoted as OD, and the number of missing places is denoted as ND. The number of places that the algorithm excessively pointed out logic errors is denoted as MD.

4.3.2 Discussion

On the whole, the SPED is superior to the LSTM-LM if there are a number of available codes with the same structure pattern. Generally, SPED could detect logic errors with fewer excess items. The LSTM-LM could also detect logic errors, but tends to point out such in excess. Another advantage of SPED is that this approach may obtain compilable codes when it provides feedback in addition to the detection. As such, we can select SPED or use the SPED-based algorithm if there are sufficient available codes for the corresponding task.

On the other hand, the result shows that SPED is not universal. First of all, if there are no available codes with the same structure pattern for a given query code, SPED cannot point out errors at all. Fortunately, the LSTM-LM can cover the weaknesses of the SPED. For example, for a query code indicated as 18, SPED could not point out logic errors, because there are no available codes, whereas LSTM-LM could point out all errors. The SPED also does not work

Table 4.3: Results of the experiment for several selected target query codes.

	SPED				LSTM-LM			
Query code #	DT [s]	OD	ND	MD	DT [s]	OD	ND	MD
1	0.3948	1	0	0	0.0450	1	0	3
2	0.7819	3	0	0	0.0450	1	2	3
3	0.6379	1	0	0	0.0440	0	1	3
4	0.4637	1	0	1	0.0450	0	1	4
5	0.7434	1	0	24	0.0440	1	0	5
6	0.5024	2	0	0	0.0450	2	0	6
7	0.4757	2	0	0	0.0450	1	1	2
8	0.3840	1	0	0	0.0450	0	1	4
9	0.7381	1	0	0	0.0450	0	1	8
10	0.6511	1	0	7	0.0440	1	0	4
11	0.6317	0	1	0	0.0450	1	0	9
12	0.5320	2	0	1	0.0420	1	1	5
13	0.3530	4	0	2	0.0390	3	1	10
14	0.6281	0	1	0	0.0460	1	0	8
15	0.1560	0	2	0	0.0470	2	0	5
16	0.7018	1	0	0	0.0450	1	0	2
17	0.6677	1	0	2	0.0440	0	1	7
18	0.5164	0	1	0	0.0460	1	0	5
19	0.7106	1	0	2	0.0480	1	0	7
20	0.5851	1	0	7	0.0460	1	0	1

well if the given query code has a rather complex structure or contains a number of different elements (variables and methods). For example, for a query code indicated as 5, SPED generates a number of excess detections (= 24), whereas LSTM-LM generates fewer excess detections (= 5). The result of the query code indicated as 15, which includes many method elements (= 15), shows that the SPED could not point out the logic error, whereas LSTM-LM could point out all errors. Therefore, according to the complexity and the number of different elements, we should consider choosing the LSTM-LM.

In terms of performance (waiting time for receiving feedback from the submission), the LSTM-LM is superior to the SPED. Although, constructing the learning model of the RNN for each task takes several hours, the query phase can be performed rather quickly. One advantage is that the performance does not depend on the number of available source codes, which are used for the learning process. On the other hand, the waiting time of SPED depends on the complexity of the given source code because it is converted into the corresponding abstract syntax tree. Thus, although the waiting time is acceptable (less than one second), we should consider the trade-off between the performance and the accuracy of detection. Table 4.4 shows a summary of the experimental results.

Table 4.4: Summary of the experimental results.

	Strength	Weakness
SPED	• High accuracy according to the number of available structure patterns • Can obtain compilable code	• Depends on structure patterns • Result will be empty if there is no available structure pattern • Tends to generate excess detection for codes with complex structure • Waiting time depends on structural features of code
LSTM	• Does not depend on the number of available source codes with the same structure pattern • Quick response and the waiting time does not depend on features of the code or on the number of available codes	• Tend to generate excess detection • Need learning time (which is not for the querying process) • Obtained code may not be compilable

Next, we discuss how to improve the two approaches. In order to increase the accuracy of the SPED, we filter available codes using structure patterns. As such, the detection accuracy depends on the diversity of source code stored in the online judge system. Therefore, a promising approach is the automatic generation of source codes with different structure patterns from existing source codes. In order to perform such operations, we can introduce other AI techniques to create program codes which have the same functionalities but different structures. In the LSTM-LM, the detection did not work well for some cases with missing and excessive items. The reasons for its failure to work include the fact that the prediction of the model does not use the parts of the sequence which appear later in the code. In the future, we plan to use bidirectional long short-term memory (BLSTM) networks, which are special kinds of RNNs, for bug detection. Since source codes are tree structures, BLSTM networks can be expected to have higher accuracy than LSTM-LM. In the present study, although we only considered codes written in Java, the algorithm can be applied to other programming languages, such as C/C++, Python, and JavaScript, which can be represented in abstract syntax trees and sequences of tokens. There are more source codes written in C/C++, and thus logic errors can be detected with higher accuracy depending on the language.

Finally, we discuss how we should implement the hybrid algorithm considering the above observations. The algorithm depends on the situation and expectation of the service. If the user (service) does not mind a longer response time for the feedback, the algorithm can be based on the competition of SPED and LSTM-LM for which their better result or unification is returned (integrated feedback). On the other hand, if the user prefers a quicker response, the algorithm should be much more sophisticated. In this case, the complexity *C* of the given query code and the availability *L* of LSTM-LM can be calculated in advance without taking much time. The value *C* is calculated based on the structural features of the query code. The value *L* can be obtained based on the probability (reliability) of each detection. If *C* is less than the predefined threshold *C'* (the code is rather simple), then we can return the integrated feedback. If this is not the case (the code is complex), and if *L* is greater than the predefined threshold *L'* (we have sufficiently reliable detection from LSTM), we do not need to activate SPED. On the other hand, if there are no reliable results from LSTM-LM, we should consider using the integrated feedback by activating SPED.

4.4 Chapter Summary

In this chapter, in order to clarify the weaknesses and strengths of existing logic error detection methods, we investigated the performance of existing methods in detecting logic errors using source codes with different structures and inherent logic errors under the same experiment. The experiments were conducted using an OJ system that stored both correct and incorrect codes. The experimental results show that, in most cases, the static code analysis approach outperforms the deep learning approach, although serious mismatches occur depending on the structural features of the given query code and the available training data. We also found that the deep learning approach was able to adequately cover such mismatches and detect logic errors. We conclude that the combination of the two different approaches provides a basis for the development of hybrid intelligent algorithms that can accurately detect logic errors and provide appropriate feedback with higher accuracy. In addition, the detection of logic errors based on deep learning does not have a reliability measure to indicate whether the resulting correction candidate is a logic error or not. This work was presented at the international conference [85].

For the SPED, the more correct codes in the database that match the structure of the incorrect code, the better the performance of detecting logic errors. On the other hand, the LSTM-LM depends on the number and quality of correct codes in the database as well as SPED, but it is not certain whether the correction candidate detected by the LSTM-LM are logic errors or not. In Chapter 5, we introduce the proposed method to improve the detection performance by LSTM-LM.

Chapter 5

Improvement of Detection Performance of LSTM-LM

In this chapter, we proposed a method that improve the detection performance of LSTM-LM. Section 5.1 presents a method that optimizes thresholds that control correction candidates detected by LSTM-LM. Section 5.2 describes an experiment to evaluate the detection performance by each threshold value. Section 5.3 describes the experimental results that indicate the thresholds can control correction candidates detected by LSTM-LM. Finally, Section 5.4 summarizes Chapter 5.

5.1 Proposed method

In the conventional method, by using thresholds, the correction candidates detected by LSTM-LM are extracted as those containing logic errors. Due to the characteristics of LSTM-LM, the probability of occurrence of tokens that are sequences of correct codes, which are training data, is high. However, the probability of tokens that are suspected to contain logical errors is low. Therefore, there is a possibility that the correction candidates detected by LSTM-LM include correction candidates that do not contain logic errors. In existing methods, the threshold is empirically set to 0.1 because the probability obtained by the machine learning model is lower for logic errors. However, since this threshold may depend on the LSTM-LM and programming task, it is necessary to determine the appropriate threshold for each programming task. In this chapter, we propose a method to optimize this threshold for each programming task.

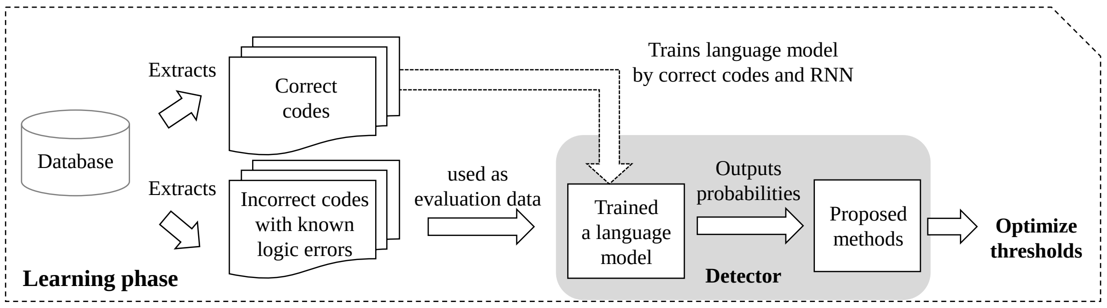


Figure 5.1: An overview of proposed method.

5.1.1 Overview

Fig. 5.1 outlines the method for optimizing the threshold value that regulates the correction candidates to logic errors only. This method uses a trained LSTM-LM and a set of incorrect codes; the LSTM-LM is capable of presenting a correction candidate for each incorrect code; the evaluation value is calculated according to the presence or absence of logic errors in the correction candidates detected by the LSTM-LM. Finally, the threshold value is optimized based on the calculated evaluation value.

Algorithm 1 shows the logic error detection based on the LSTM-LM. The model infers a token x_{t+1} from a sequence $[x_1, x_2, .., x_t]$ with the probability. If the probability is high, we consider that the token x_{t+1} is a pattern that tends to appear in the learning data. On the other hand, if the probability is low, we evaluate that the token x_{t+1} is a pattern that rarely appears in the data. So, after a threshold τ is defined, a token that has a probability of less than τ can be extracted as possible logic errors. In our previous approach, constant thresholds have been optimized manually based on a heuristic. Thus, in this paper, we propose the approach to optimize the thresholds that control the detection accuracy as well as its perspectives.

5.1.2 Optimization of threshold

Algorithm 2 shows an algorithm to optimize thresholds that enhance the detection performance of the LSTM-LM. The algorithm inputs X, a list of incorrect codes, and Logic Error Positions (LEP), a list of sets, each of which includes places of known logic errors in the correct code i. The algorithm outputs two thresholds $\tau_{OD_{max}}$ and $\tau_{ND_{min}}$ which control the detection performance of the LSTM-LM. The algorithm computes the detection performance of the LSTM-LM for each threshold τ in (0, 1.0].

Let LEP$_i$ be a set of known logic errors in the source code i from the set of incorrect codes

Algorithm 1 Positions DP = Logic error detection(x, lines, τ)

```
$\boldsymbol{p} \leftarrow$ LSTM-LM($\boldsymbol{x}$)
DP $\leftarrow \emptyset$
for t=0 to x.length do
    idx $\leftarrow \arg\max$($\boldsymbol{p}$[t+1])
    if x[t+1] $\neq$ idx and p[t+1] $\leq \tau$ then
        if lines[t+1] $\neq$ lines[t+2] then
            DP.append(lines[t+1]+1)
        else
            DP.append(lines[t+1])
        end if
    end if
end for
return DP
```

Algorithm 2 Threshold $\tau_{OD_{max}}$, $\tau_{ND_{min}}$ = threshold optimization(X, LINES, LEP)

```
$d \leftarrow$ 1000
$size \leftarrow$ X.size
OD, ND, MD, OVD $\leftarrow$ [$d$+1]
for $\tau$ = 1 to $d$+1 do
    OD[$\tau$] $\leftarrow$ 0, ND[$\tau$] $\leftarrow$ 0
    MD[$\tau$] $\leftarrow$ 0, OVD[$\tau$] $\leftarrow$ 0
    for $i$=0 to $size$ do
        $DP_i \leftarrow$ algorithm1($\boldsymbol{X}_i$, $LINES_i$, $\tau$)
        if DP = $\emptyset$ then
            ND[$\tau$] += 1
        else if $LEP_i \subseteq DP_i$ then
            OD[$\tau$] += 1
        else if $LEP_i \cap DP_i$ = $\emptyset$ then
            MD[$\tau$] += 1
        else
            OVD[$\tau$] += 1
        end if
    end for
    OD[$\tau$] $\leftarrow$ OD[$\tau$] $\div size * 100$
    ND[$\tau$] $\leftarrow$ ND[$\tau$] $\div size * 100$
    MD[$\tau$] $\leftarrow$ MD[$\tau$] $\div size * 100$
    OVD[$\tau$] $\leftarrow$ OVD[$\tau$] $\div size * 100$
end for
$\tau_{OD_{max}} \leftarrow \frac{\arg\max(OD)}{d}$
$\tau_{ND_{min}} \leftarrow \frac{\arg\min(ND)}{d}$
return $\tau_{OD_{max}}$, $\tau_{ND_{min}}$
```

X, and Detected Positions (DP$_i$) be a set of lines detected by Algorithm 1. The performance of the LSTM-LM with the threshold τ is estimated by the following evaluation values:

$$OD = \frac{\|\{i \in X \mid DP_i \subseteq LEP_i\}\|}{\|X\|} * 100 \tag{5.1}$$

$$ND = \frac{\|\{i \in X \mid DP_i \cap LEP_i = \emptyset\}\|}{\|X\|} * 100 \tag{5.2}$$

$$MD = \frac{\|\{i \in X \mid DP_i \setminus LEP_i \neq DP_i\}\|}{\|X\|} * 100 \tag{5.3}$$

$$OVD = \frac{\|\{i \in X \mid LEP_i \subset DP_i\}\|}{\|X\|} * 100 \tag{5.4}$$

We proposed evaluation values based on ratios of Only Detection (OD), Non-Detection (ND), MisDetection (MD), and OVerDetection (OVD) of the LSTM-LM to optimize the thresholds. These evaluation values are calculated by the presence or absence of logic errors in the correction candidates detected by LSTM-LM. The evaluation value OD is the ratio of source codes that LSTM-LM could detect only logic errors. The evaluation value ND is the ratio of source codes that LSTM-LM could not detect logic errors. The evaluation value MD is the ratio of source codes to which the LSTM-LM provides inappropriate places as misdetection. The evaluation value OVD is the ratio of source codes that the LSTM-LM provides for both proper detection and misdetection. These values indicate the reliability of the correction candidates detected by LSTM-LM.

$\tau_{OD_{max}}$ is the threshold where OD is maximized and the number of correction candidates obtained by the LSTM-LM is minimum. On the other hand, $\tau_{ND_{min}}$ is the threshold where ND is minimized and the number of correction candidates obtained by the LSTM-LM is maximum.

5.2 Experiment

In order to verify the validity of the thresholds optimized by the proposed method, experiments were conducted to compare the detection performance when using the optimized thresholds and a conventional threshold. In the experiment, the internal parameters of LSTM-LM were learned by using the correct codes stored in 32 programming tasks. The set of incorrect

Table 5.1: Overview of datasets.

Problem ID	#training data	#test data	Problem ID	#training data	#test data
ITP1_1_A	5820	1413	ITP1_5_A	1234	282
ITP1_1_B	4796	816	ITP1_5_B	1694	201
ITP1_1_C	4050	807	ITP1_5_C	1787	135
ITP1_1_D	3123	535	ITP1_5_D	366	71
ITP1_2_A	2555	630	ITP1_6_A	1211	213
ITP1_2_B	2997	683	ITP1_6_B	822	95
ITP1_2_C	2099	360	ITP1_6_C	871	192
ITP1_2_D	1963	383	ITP1_6_D	1136	99
ITP1_3_A	2999	514	ITP1_7_A	999	127
ITP1_3_B	1824	352	ITP1_7_B	986	217
ITP1_3_C	1881	269	ITP1_7_C	847	125
ITP1_3_D	2042	357	ITP1_7_D	497	79
ITP1_4_A	1981	335	ITP1_8_A	600	92
ITP1_4_B	1312	339	ITP1_8_B	492	41
ITP1_4_C	1381	99	ITP1_8_C	497	57
ITP1_4_D	653	115	ITP1_8_D	383	45

codes corresponding to each programming task was then used to optimize the thresholds. We analyzed the respective evaluation values when the thresholds are optimized.

5.2.1 Experimental Data

In the experiment, we employed source codes accumulated in AOJ [8, 69], which is one of major OJ system. AOJ provides a number of tasks and judges submitted source codes oriented to specific tasks. The submitted source code is compiled and executed with a number of strict test cases. Therefore, the judge provides verdict whether the code is correct as well as evaluates resource usage such as CPU time and memory.

Among a number of available tasks in AOJ, we select a set of tasks from Introduction to Programming 1 (ITP1), which is a special course for introduction to programming. Table 5.1 shows details of the selected tasks.

The number of training data denotes the number of correct codes to learn the LSTM-LM. The number of test data denotes the number of incorrect codes to optimize the thresholds as well as to evaluate the LSTM-LM. In this experiment, source codes, each of which has only one line with any logic error, were extracted as the test data.

Table 5.2: A vocabulary for tokens.

ID	word	ID	word	ID	word	ID	word	ID	word	ID	word
0	(masking value)	60	case	82	(	104	>	126	U	150	n
1-20	(variables)	61	char	83	)	105	@	127	V	151	o
21-40	(functions)	62	else	84	*	106	A	128	W	152	p
41	continue	63	enum	85	+	107	B	129	X	153	q
42	unsigned	64	goto	86	,	108	C	130	Y	154	r
43	default	65	long	87	-	109	D	131	Z	155	s
44	typedef	66	main	88	.	110	E	132	[	156	t
45	double	67	void	89	/	111	F	133	\	157	u
46	extern	68	for	90	0	112	G	134	]	158	v
47	signed	69	int	91	1	113	H	135	^	159	w
48	sizeof	70	do	92	2	114	I	136	‘	160	x
49	static	71	if	93	3	115	J	137	a	161	y
50	struct	72	(space)	94	4	116	K	138	b	162	z
51	switch	73	!	95	5	117	L	139	c	163	{
52	return	74	?	96	6	118	M	140	d	164	—
53	break	75	_	97	7	119	N	141	g	165	}
54	const	76	”	98	8	120	O	144	h	166	∼
55	float	77	#	99	9	121	P	145	i		
56	short	78	$	100	:	122	Q	146	j		
57	union	79	%	101	;	123	R	147	k		
58	while	80	&	102	<	124	S	148	l		
59	auto	81	’	103	=	125	T	149	m		

5.2.2 Preprocessing

We extracted source codes that are compilable without errors for both learning of the LSTM-LM and optimizing the thresholds. In addition, we excluded source codes that include functional macro and functions defined by the programmer. Finally, tokens related to tabs, space characters, and comments in extracted codes were removed.

Generally, an LSTM-LM can cause memory loss if a sequence $\boldsymbol{x}$ is too large, and makes it difficult to learn parameters for each layer. So, to statistically optimize the abnormal value, we apply the Hotelling's T^2 theory [86] for the length of the ID sequence. In this chapter, the source code was considered as the abnormal values when the chi-square value of the length of the source code exceeded 99 percent on a chi-square distribution with 1 degree of freedom. The source code that was judged as an outlier by the p-value which is $X^2_{0.99}(1) = 0.00016$, the source code is rejected from the extracted source codes.

$\boldsymbol{x}$ is a sequence of IDs transformed from tokens in the source code according to the vocabulary in table 5.2. In the transformation process, variable names, function names, keywords are encoded to integers in 1-20, 21-40 and 41-166 respectively. A sequence of IDs $\boldsymbol{x}$ which is variable-length is transformed to the sequence which is fixed length using the zero padding.

5.2.3 Hyperparameters

In this experiment, the data presented in Table 5.1 was used as the learning data of LSTM-LM. The number of hidden neurons is 500, the number of epochs is 50, and the number of neurons for the input and output layer is 167, which is the maximum number of vocabularies. The batch size is 16. To avoid overtraining, we set learning rate α = 0.001, β_1 = 0.9, β_2 = 0.999, ϵ = 1e-8 based on the Adam optimizer [84]. Moreover, we set a dropout ratio as 0.5 [83].

5.3 Result and Discussion

5.3.1 Optimized Thresholds for LSTM-LM

Table 5.3 shows the fixed threshold τ_1 and the threshold values $\tau_{OD_{max}}$ and $\tau_{ND_{min}}$ obtained by the proposed method for each programming task. It can be seen that the optimized thresholds are adapted to each programming task. This indicates that the thresholds $\tau_{OD_{max}}$ and $\tau_{ND_{min}}$ obtained by the proposed method instead of the threshold τ_1 can be set as thresholds that are more suitable for each programming task than the conventional thresholds.

Table 5.4 demonstrates the detection performance of the LSTM-LM for different thresholds presented in table 5.3. It can be seen that the threshold $\tau_{OD_{max}}$ optimized by the proposed method is smaller than $\tau_{ND_{min}}$. This indicates that the detection accuracy is higher when the threshold $\tau_{OD_{max}}$ is smaller than τ_1, which is the threshold used in the conventional method. This indicates that the threshold value $\tau_{OD_{max}}$ optimized by the proposed method is more suitable than tau_1 of the conventional method. On the other hand, the threshold value $\tau_{ND_{min}}$ is in many cases smaller than τ_1 of the conventional method.

In the programming tasks ITP1_4_B, ITP1_5_C, ITP1_7_B, and ITP1_7_C, MD is greater than OD when set to the threshold $\tau_{OD_{max}}$. This indicates that the detection accuracy by LSTM-LM is low. This indicates that there are many false positives in each programming task. However, this phenomenon does not occur for other tasks, suggesting that the problem is task- and LSTM-LM-dependent.

In this way, the different thresholds provide different perspectives to obtain the feedback of detection results. $\tau_{OD_{max}}$ is oriented to the situation in which the user wants to know only the place with true errors and less misdetection. On the other hand, $\tau_{ND_{min}}$ is oriented to the situation in which the user does not want to miss the true logic error even if there are relatively many excess correction candidates.

Table 5.3: Optimized thresholds by proposed method.

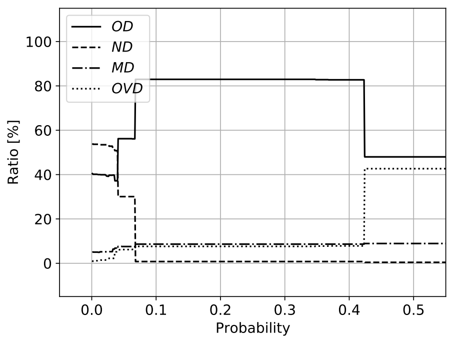
	Proposed method		Previous method
Problem ID	$\tau_{OD_{max}}$	$\tau_{ND_{min}}$	τ_1
ITP1_1_A	0.063	0.424	0.1
ITP1_1_B	0.006	0.011	0.1
ITP1_1_C	0.006	0.015	0.1
ITP1_1_D	0.001	0.029	0.1
ITP1_2_A	0.001	0.172	0.1
ITP1_2_B	0.003	0.395	0.1
ITP1_2_C	0.003	0.154	0.1
ITP1_2_D	0.003	0.146	0.1
ITP1_3_A	0.027	0.349	0.1
ITP1_3_B	0.007	0.206	0.1
ITP1_3_C	0.007	0.14	0.1
ITP1_3_D	0.006	0.035	0.1
ITP1_4_A	0.005	0.016	0.1
ITP1_4_B	0.001	0.008	0.1
ITP1_4_C	0.005	0.086	0.1
ITP1_4_D	0.007	0.049	0.1
ITP1_5_A	0.002	0.076	0.1
ITP1_5_B	0.002	0.064	0.1
ITP1_5_C	0.007	0.073	0.1
ITP1_5_D	0.003	0.056	0.1
ITP1_6_A	0.002	0.057	0.1
ITP1_6_B	0.007	0.148	0.1
ITP1_6_C	0.007	0.042	0.1
ITP1_6_D	0.003	0.156	0.1
ITP1_7_A	0.002	0.214	0.1
ITP1_7_B	0.009	0.227	0.1
ITP1_7_C	0.006	0.089	0.1
ITP1_7_D	0.021	0.058	0.1
ITP1_8_A	0.015	0.121	0.1
ITP1_8_B	0.004	0.122	0.1
ITP1_8_C	0.003	0.191	0.1
ITP1_8_D	0.02	0.287	0.1

Table 5.4: Experimental results.

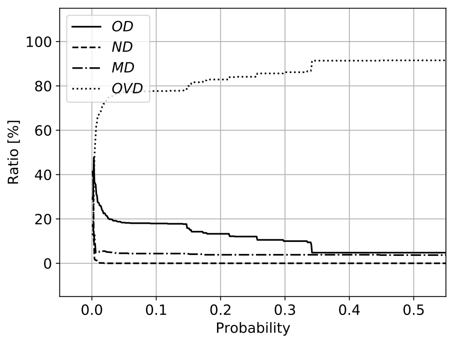
	Proposed method								Previous study			
Target	$\tau = \tau_{OD_{max}}$				$\tau = \tau_{ND_{min}}$				$\tau = \tau_1$			
Problem ID	OD	OVD	MD	ND	OD	OVD	MD	ND	OD	OVD	MD	ND
ITP1_1_A	82.95	7.63	8.63	0.79	47.99	42.66	8.92	0.43	82.95	7.63	8.63	0.79
ITP1_1_B	36.71	48.36	12.88	2.05	22.60	71.92	5.48	0.00	17.95	77.67	4.38	0.00
ITP1_1_C	76.05	14.99	8.54	0.42	72.27	19.61	8.12	0.00	43.56	49.72	6.72	0.00
ITP1_1_D	60.18	33.48	3.94	2.41	43.54	52.95	3.50	0.00	9.41	87.09	3.50	0.00
ITP1_2_A	71.67	15.88	4.72	7.73	26.61	66.52	6.87	0.00	41.63	50.21	7.30	0.86
ITP1_2_B	85.83	4.72	4.72	4.72	63.04	32.85	4.11	0.00	38.81	57.49	3.70	0.00
ITP1_2_C	56.43	20.75	7.05	15.77	6.22	85.89	7.88	0.00	11.62	78.42	9.13	0.83
ITP1_2_D	68.48	17.75	5.43	8.33	42.39	53.62	3.99	0.00	44.93	50.72	3.62	0.72
ITP1_3_A	80.66	10.29	6.17	2.88	64.81	31.89	3.29	0.00	71.81	22.63	4.12	1.44
ITP1_3_B	65.37	11.97	11.33	11.33	27.18	61.17	11.65	0.00	34.95	52.10	11.97	0.97
ITP1_3_C	52.22	21.67	12.78	13.33	15.00	71.11	13.89	0.00	22.22	63.89	12.78	1.11
ITP1_3_D	62.80	22.56	7.32	7.32	45.73	44.51	9.76	0.00	29.57	60.98	9.45	0.00
ITP1_4_A	50.65	45.45	3.03	0.87	50.65	47.19	2.16	0.00	28.57	69.70	1.73	0.00
ITP1_4_B	29.94	36.53	30.54	2.99	28.14	39.52	32.34	0.00	20.36	51.50	28.14	0.00
ITP1_4_C	66.25	13.75	5.00	15.00	30.00	65.00	5.00	0.00	30.00	65.00	5.00	0.00
ITP1_4_D	66.67	18.67	9.33	5.33	53.33	38.67	8.00	0.00	25.33	69.33	5.33	0.00
ITP1_5_A	63.42	10.12	11.67	14.79	19.46	65.37	15.18	0.00	7.39	77.43	15.18	0.00
ITP1_5_B	40.72	20.96	11.98	26.35	10.18	62.28	27.54	0.00	7.78	65.87	26.35	0.00
ITP1_5_C	27.43	31.86	23.01	17.70	7.96	68.14	23.89	0.00	4.42	71.68	23.89	0.00
ITP1_5_D	41.18	21.57	27.45	9.80	31.37	43.14	25.49	0.00	9.80	66.67	23.53	0.00
ITP1_6_A	50.39	13.18	25.58	10.85	23.26	50.39	26.36	0.00	17.83	58.91	23.26	0.00
ITP1_6_B	31.65	21.52	7.59	39.24	13.92	64.56	21.52	0.00	12.66	65.82	21.52	0.00
ITP1_6_C	36.77	25.81	27.74	9.68	9.03	77.42	13.55	0.00	7.74	81.94	10.32	0.00
ITP1_6_D	34.38	21.88	12.50	31.25	6.25	75.00	18.75	0.00	6.25	73.44	18.75	1.56
ITP1_7_A	65.66	19.19	9.09	6.06	18.18	69.70	12.12	0.00	13.13	75.76	11.11	0.00
ITP1_7_B	17.24	8.05	17.82	56.90	0.57	45.98	53.45	0.00	5.17	34.48	51.15	9.20
ITP1_7_C	14.94	24.14	28.74	32.18	6.90	75.86	17.24	0.00	6.90	75.86	17.24	0.00
ITP1_7_D	43.75	12.50	18.75	25.00	25.00	37.50	37.50	0.00	6.25	75.00	18.75	0.00
ITP1_8_A	22.50	25.00	10.00	42.50	5.00	73.75	21.25	0.00	13.75	58.75	25.00	2.50
ITP1_8_B	29.41	20.59	23.53	26.47	14.71	47.06	38.24	0.00	11.76	52.94	35.29	0.00
ITP1_8_C	53.19	25.53	10.64	10.64	19.15	72.34	8.51	0.00	19.15	72.34	6.38	2.13
ITP1_8_D	22.22	44.44	3.70	29.63	0.00	92.59	7.41	0.00	7.41	62.96	22.22	7.41

5.3.2 Evaluation Values for Each Programming Task

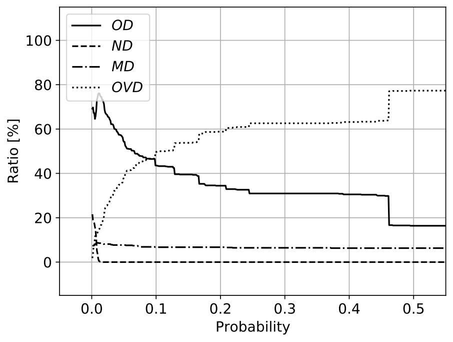
Figure 5.2 shows the evaluation values for different thresholds for each programming task. To show variation of the evaluation values for different tasks, we selected four tasks for the experiment. Experimental results for other programming tasks are given in Section A.1 in the Chapter Appendix.



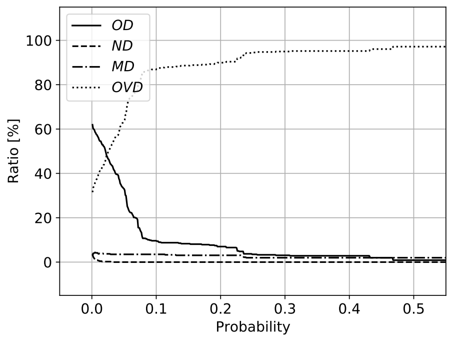
(a) ITP1_1_A.



(b) ITP1_1_B.



(c) ITP1_1_C.



(d) ITP1_1_D.

Figure 5.2: Evaluation values by changes of threshold τ.

The four evaluation values show different variations for each programming task because the tasks have different specifications and the number of available solution codes. This indicates that the performance of the LSTM-LM depends on each programming task. The results show that the four values do not change when the threshold τ is greater than 0.5. Thus, if the probability obtained by the LSTM-LM is greater than 0.5, the possibility that the detected place includes logic errors is low. Therefore, we focus on the correction candidates whose probability is less than 0.5 to evaluate the detection performance of the LSTM-LM. In other words, we observe the four evaluation values where τ is less than or equal to 0.5.

5.3.3 Limitations

The proposed method depend on the detection performance of LSTM-LM. Therefore, if LSTM-LM itself cannot detect logic errors, it cannot be restricted to only the logic errors in the correction candidates. Therefore, if LSTM-LM is unable to detect logic errors, it is possible to detect more logic errors by using language models based on the attention mechanism or by using the latest language models such as BERT [35]. However, these models require more training data.

5.4 Chapter Summary

We proposed a method that optimize thresholds that regulate correction candidates detected by LSTM-LM. The method enables to control correction candidates provided as logic errors by optimizing the thresholds of probabilities obtained from the language model. The experimental results show that the algorithm improves the detection performance of the LSTM-LM by using not only correct codes but also incorrect codes. The experiment has been conducted by using a problem set and the corresponding accumulated source codes of an OJ system. An important feature of the proposed approach is that the thresholds can be optimized for each programming task and optimize the detection performance of the LSTM-LM. On the other hand, there is room to improve the accuracy of the LSTM-LM. So, additional techniques such as attention mechanism will be considered to improve the learning model. This study was presented at an international conference [87].

Chapter 6

A Model with Iterative Trials for Correcting Logic Errors in Source Code

MLB logic error detection based on LSTM-LM shows a use case where logic errors in the source code can be presented as correction candidates [14]. However, the extent to which these methods can be used to correct logic errors in source code has not been verified in practice. In this chapter, we develop a debugging support model using the correction candidates obtained from LSTM-LM and evaluate the performance in correcting logic errors. In addition, we introduce a use case of the debugging support model.

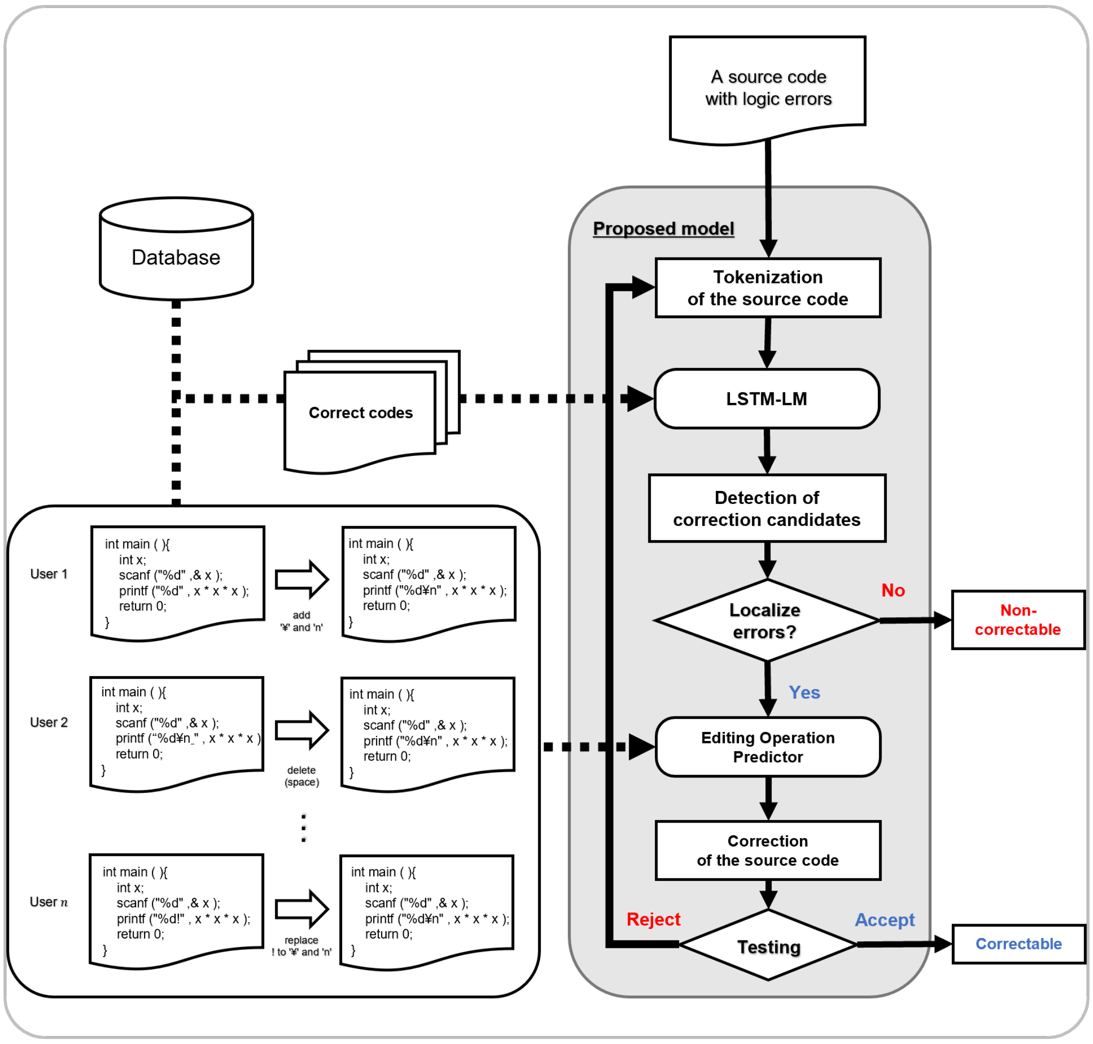
In Chapter 6, we proposed a model that detects and corrects logic errors iteratively. Section 6.1 introduces an overview of the proposed model. Section 6.2 describes an experiment to evaluate the detection and correction performance of the proposed model. Section 6.3 describes the experimental results that indicate the proposed model can correct multiple logic errors. Finally, Section 6.4 summarizes Chapter 6.

6.1 Proposed Model

This section introduces a proposed model using LSTM-LM and support vector machine (SVM). First, we explain that LSTM-LM is used to identify logic errors in the source code. Next, the proposed EOP is described.

6.1.1 Overview of Proposed Model

Figure 6.1 outlines the proposed model for correcting multiple logic errors in a given source code. The model debugs an incorrect code by iteratively identifying and correcting logic errors and testing the modified code. This model consists of LSTM-LM and EOP. The EOP predicts an editing operation from a correction candidate obtained by LSTM-LM. Based on the editing operation, EOP seeks to correct the logic error in the source code. Then the model tests whether the corrected source code is correct or not. If the source code is incorrect, it again becomes input data for LSTM-LM. In this way, the model can debug source code containing multiple logic errors.



User 1:

```
int main ( ){
    int x;
    scanf ("%d" ,& x );
    printf ("%d" , x * x * x );
    return 0;
}
```

```
int main ( ){
    int x;
    scanf ("%d" ,& x );
    printf ("%d¥n" , x * x * x );
    return 0;
}
```

User 2:

```
int main ( ){
    int x;
    scanf ("%d" ,& x );
    printf ("%d¥n_" , x * x * x )
    return 0;
}
```

```
int main ( ){
    int x;
    scanf ("%d" ,& x );
    printf ("%d¥n" , x * x * x );
    return 0;
}
```

User n:

```
int main ( ){
    int x;
    scanf ("%d" ,& x );
    printf ("%d!" , x * x * x );
    return 0;
}
```

```
int main ( ){
    int x;
    scanf ("%d" ,& x );
    printf ("%d¥n" , x * x * x );
    return 0;
}
```

Figure 6.1: Overview of the proposed model.

A set of source codes oriented to the specification of a programming task is required to build the proposed model. LSTM-LM has learned the structure of a set of correct codes to predict the position of logic errors. EOP can predict the editing operation for the correction candidate by learning the editing operation performed between the incorrect code and the corresponding correct code.

6.1.2 Editing Operation Predictor (EOP)

If the token $\hat{x}_{t+1}$ predicted by Algorithm 1 and the original token x_{t+1} do not match, the source code needs to be edited using the predicted token $\hat{x}_{t+1}$ for the position of the token x_t and the token x_{t+1}. The editing operations are considered as follows using the three tokens.

insert insert the predicted token $\hat{x}_{t+1}$ between token x_t and the next token x_{t+1}.

delete delete the next token x_{t+1} between token x_t and $\hat{x}_{t+1}$.

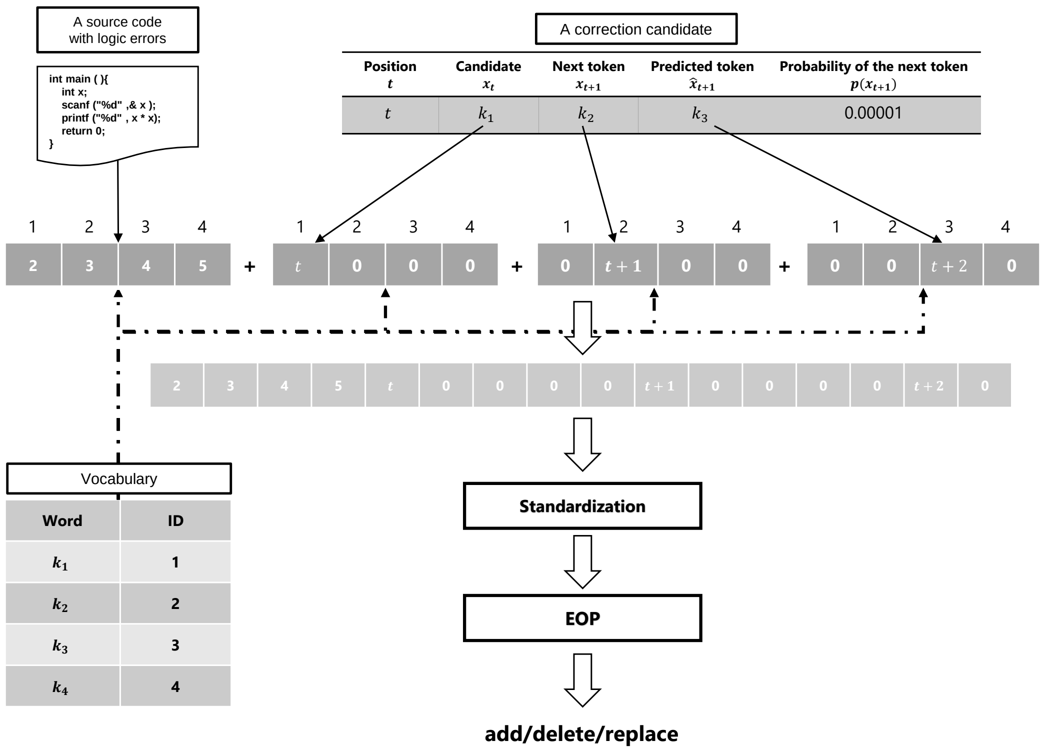
replace replace the next token x_{t+1} with the predicted token $\hat{x}_{t+1}$.

Editing operations can be predicted by using the tokens x_t, x_{t+1} and $\hat{x}_{t+1}$. This assumes that the programmer can edit the source code if he/she knows the token and position to modify. EOP predicts the editing operation from the structure of the source code $\boldsymbol{x}$ and the three tokens x_t, x_{t+1} and $\hat{x}_{t+1}$.

Figure 6.2 shows how the model edits the source code based on the correction candidates obtained from Algorithm 1. The correction candidate with the highest possibility of a logic error is selected from the correction candidates list obtained from Algorithm 1. The selected correction candidate is converted into feature vectors using the vocabulary table used for tokenizing. Four feature vectors of the same size as the vocabulary table are created. They are a vector for the position of token x_t, a vector for the position of token x_{t+1}, a vector for the position of token $\hat{x}_{t+1}$, and a total number of each vocabulary in the source code. The concatenation of these four vectors is used as the EOP input.

Editing operations of source code can be classified into three categories: insertion, deletion, or replacement of tokens. Therefore, we consider the prediction of edit operations for source code as a multi-classification problem. To construct the EOP, we use SVM [88], which can solve multi-classification problems.

To learn the internal parameters of EOP, feature vectors and teacher labels are extracted using a set of source codes obtained from the DB. The submission logs of all users are extracted from the set of source codes, and a set of pairs, each of which consists of a correct and an incorrect code are employed as learning data. The extracted pair corresponds to an editing



```
int main ( ){
    int x;
    scanf ("%d" ,& x );
    printf ("%d" , x * x);
    return 0;
}
```

Position t	Candidate x_t	Next token x_{t+1}	Predicted token $\hat{x}_{t+1}$	Probability of the next token $p(x_{t+1})$
t	k_1	k_2	k_3	0.00001

Word	ID
k_1	1
k_2	2
k_3	3
k_4	4

Figure 6.2: Role of the EOP.

process from an incorrect code to a correct code. By calculating the Shortest Editing Script (SES) that provides the editing operations between two source codes, it is possible to obtain the editing operations for the candidates to be corrected. The SES is calculated from the incorrect and correct codes by dynamic programming. Incorrect code and correct code are converted into an ID sequence in advance using the vocabulary table. By comparing the ID sequence of the incorrect code with the position of the SES, it is possible to label what kind of editing operation the token in the SES can perform on the editing position. EOP learns these feature vectors and labels for editing operations using SVM.

6.1.3 Iterative Trials

The source code can be automatically corrected by combining LSTM-LM and EOP with Algorithm 1. If the source code modified by the model does not meet the specifications of the given programming task, it may need to be corrected again. This mechanism is based on the iterative repair proposed by Deepfix [58]. In this model, the source code is tested using test cases to check whether the modified source code contains logic errors. If the modified source code passes the test, the debugging process is terminated assuming that all logic errors in the source code have been corrected. On the other hand, if the modified source code does not pass the test, the source code will be corrected again. If the target source code is written in a programming language that requires a compiler (e.g., C language), the model uses the compiler to convert it to an executable code before testing.

6.2 Experiment

To verify the usefulness of the proposed model, we compared it with the conventional model without iterative trials. In the conventional model, only one trial of correction is performed using the correction candidates predicted by LSTM-LM. It is necessary to define metrics for evaluating the usefulness of the proposed model. We evaluated the performance of the proposed model by focusing on the correction accuracy of logic errors, the number of trials, and the execution time. We defined the correction accuracy as the ratio of the source codes in which all logic errors are corrected in the experimental data. We defined the number of corrections as the average number of corrections until the correct code is obtained in the experimental data. We defined the execution time as the time until the given source code becomes correct by the proposed model. However, if the model could not correct the given source code, it was not included in this result. We used a 64-bit Windows 10 computer with an Intel Core i9-9900K CPU (3.60 GHz), 32 GB RAM, and Nvidia GeForce RTX 2070 SUPER GPU.

To train the proposed model and conduct an experiment to verify its usefulness, we used source codes in C language which were written by programming learners for 32 programming tasks.

6.2.1 Dataset

Table 6.4 shows the experimental data used to evaluate the proposed model. Here, targets is the number of incorrect codes that contain logic errors for each programming task. We selected the target codes with an edit distance of five or less and iterated until the code was corrected. Moreover, we categorized the source codes by the number of tokens that cause logic errors. The proposed model tries to correct logic errors until the given code is corrected. If the source code cannot be edited using the proposed model, modifications may be repeated infinitely as long as there are correction candidates. To avoid this situation, we set a termination condition in the proposed model. The model terminates its trials when the number of correction candidates indicated by Algorithm 1 becomes 0 or the number of iterations of the model exceeds 30.

To focus on solution codes which include logic errors, we excluded those that could not be properly compiled because of syntax errors or warnings. Moreover, source codes including functions and function macros defined by users were excluded. We deleted comments, tabs, and spaces in the source codes to remove unnecessary tokens.

6.2.2 Training and Training Accuracy

The source code used for training was tokenized to the sequence $\boldsymbol{x}$ based on Table 6.1. In LSTM-LM, if the sequence $\boldsymbol{x}$ becomes long, memory leak may occur due to the increase in internal parameters. Therefore, we apply Hotelling's theory [86] to the length of the sequence $\boldsymbol{x}$ and statistically determine outliers. When the chi-square value of the length of each sequence exceeds 99% of the χ^2 distribution with one degree of freedom, the source code is regarded as an abnormal value. Source codes judged to be outliers were rejected from the training data using the significance probability (p-value) $\chi^2_{0.99}(1) = 0.00016$. This means that source codes were rarely rejected.

Table 6.1: Vocabulary.

ID	word	ID	word	ID	word	ID	word	ID	word	ID	word
0	(masking value)	70	auto	92	’	114	=	126	T	160	m
1-20	(variables)	71	case	93	(	115	>	127	U	161	n
21-50	(functions)	72	char	94	)	116	@	128	V	162	o
51	continue	73	else	95	*	117	A	129	W	163	p
52	unsigned	74	enum	96	+	118	B	130	X	164	q
53	default	75	goto	97	,	119	C	131	Y	165	r
54	typedef	76	long	98	-	120	D	132	Z	166	s
55	define	77	main	99	.	121	E	133	[	167	t
56	double	78	void	90	/	122	F	134	\	168	u
57	extern	79	for	101	0	123	G	135	]	169	v
58	signed	80	int	102	1	124	H	136	ˆ	170	w
59	sizeof	81	do	103	2	125	I	137	‘	171	x
60	static	82	if	104	3	126	J	138	a	172	y
61	struct	83	(space)	105	4	127	K	139	b	173	z
62	switch	84	!	106	5	128	L	140	c	174	{
63	return	85	?	107	6	129	M	141	d	175	—
64	break	86	_	108	7	130	N	144	g	176	}
65	const	87	”	109	8	131	O	155	h	177	∼
66	float	88	#	110	9	132	P	156	i		
67	short	89	$	111	:	133	Q	157	j		
68	union	90	%	112	;	134	R	158	k		
69	while	91	&	103	<	135	S	159	l		

LSTM-LM was constructed using Tensorflow (2.4.0) [89]. To train LSTM-LM, we set the batch size to 4, the number of hidden neurons to 256, and the number of epochs to 100 as hyperparameters. We selected categorical entropy as the loss function. To prevent overfitting of LSTM-LM, we used the Adam optimizer with four parameters based on the recommendation of Ref. [84]: learning rate $\alpha = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$, and $\epsilon = 1e - 8$. As another countermeasure against overfitting, we set the dropout rate to be 0.5 based on the recommendation of Ref. [83].

Table 6.2 shows the accuracy of the LSTM-LM trained using the correct codes of 32 programming tasks. Task ID is the Problem ID of 32 programming tasks in AOJ respectively. #training data is the number of correct codes used for training, excluding duplicate correct codes. #training data shows that solution codes accumulated in AOJ are available for each programming task.

Table 6.2: Training accuracy of LSTM-LM.

Task ID	#training data	Perplexity	Training accuracy [%]
ITP1_1_A	73	1.08	93.28
ITP1_1_B	528	1.08	93.72
ITP1_1_C	1035	1.07	94.41
ITP1_1_D	1905	1.07	94.88
ITP1_2_A	886	1.06	96.24
ITP1_2_B	627	1.06	96.01
ITP1_2_C	1380	1.05	95.79
ITP1_2_D	1513	1.05	96.11
ITP1_3_A	380	1.04	94.10
ITP1_3_B	1241	1.07	94.09
ITP1_3_C	1530	1.06	95.05
ITP1_3_D	1266	1.06	95.29
ITP1_4_A	223	1.07	94.98
ITP1_4_B	212	1.06	94.01
ITP1_4_C	1148	1.05	96.61
ITP1_4_D	332	1.09	94.15
ITP1_5_A	946	1.06	96.17
ITP1_5_B	1673	1.07	95.82
ITP1_5_C	1712	1.05	95.44
ITP1_5_D	338	1.08	94.31
ITP1_6_A	1141	1.07	95.21
ITP1_6_B	760	1.07	95.19
ITP1_6_C	765	1.05	96.17
ITP1_6_D	1119	1.06	96.40
ITP1_7_A	934	1.05	96.50
ITP1_7_B	904	1.08	94.11
ITP1_7_C	769	1.09	94.98
ITP1_7_D	213	1.08	95.72
ITP1_8_A	569	1.08	94.02
ITP1_8_B	432	1.10	91.58
ITP1_8_C	455	1.04	94.21
ITP1_8_D	173	1.16	87.89

EOP was constructed using Scikit-learn (0.23.2) [90]. 10-fold cross validation was used to evaluate the performance of EOP. #training data was divided into 10 parts, 9 of which were used for training, and 1 was used as test data. From this process, 10 models were constructed and the accuracy of each of them was calculated.

Table 6.3 shows the accuracy of the trained EOP. Here, Task ID is the Problem ID of the 32 programming tasks. #training data is the number of features based on the editing information between the incorrect code and the corresponding correct code in the 32 programming tasks. We excluded duplicate data from #training data. Training accuracy and test accuracy of EOP are averages of 10 models obtained using k-fold cross validation.

Table 6.3: Training and Test accuracy of EOP.

Task ID	#training data	Training accuracy [%]	Test accuracy [%]
ITP1_1_A	1085	92.82	84.89
ITP1_1_B	5600	91.68	87.13
ITP1_1_C	5585	89.63	86.05
ITP1_1_D	5946	87.80	81.90
ITP1_2_A	6267	94.74	91.86
ITP1_2_B	3784	93.67	89.46
ITP1_2_C	16635	88.64	85.39
ITP1_2_D	7249	88.59	84.15
ITP1_3_A	1503	93.62	86.10
ITP1_3_B	6190	89.95	86.49
ITP1_3_C	8137	90.80	87.18
ITP1_3_D	2972	89.42	81.86
ITP1_4_A	608	94.15	82.74
ITP1_4_B	891	96.53	87.88
ITP1_4_C	5286	90.36	85.60
ITP1_4_D	2331	85.59	79.58
ITP1_5_A	5829	94.87	91.13
ITP1_5_B	4936	90.83	85.96
ITP1_5_C	5400	90.65	85.89
ITP1_5_D	3560	93.53	98.99
ITP1_6_A	6393	94.73	91.69
ITP1_6_B	12291	86.17	81.26
ITP1_6_C	5586	90.59	85.97
ITP1_6_D	4654	86.16	80.28
ITP1_7_A	8937	89.75	85.48
ITP1_7_B	6718	88.97	83.22
ITP1_7_C	7805	87.85	82.93
ITP1_7_D	2125	90.01	84.28
ITP1_8_A	4316	89.58	82.95
ITP1_8_B	6700	82.52	75.30
ITP1_8_C	5080	86.56	79.92
ITP1_8_D	492	94.94	88.41

Since the training accuracy and verification accuracy is 80% or more in many programming tasks, the EOP is considered to have sufficient predictive performance for editing operations using the editing position and its token. This means that the EOP can likely be effectively used to correct incorrect code.

Table 6.4: Details of experimental data

Datasets		The number of logic errors				
Task ID	Targets	1	2	3	4	5
ITP1_1_A	1524	281	1094	53	80	16
ITP1_1_B	630	42	536	13	31	8
ITP1_1_C	694	51	519	18	99	7
ITP1_1_D	500	16	365	20	88	11
ITP1_2_A	494	113	108	8	59	206
ITP1_2_B	727	17	466	15	160	69
ITP1_2_C	308	33	243	4	26	2
ITP1_2_D	350	28	72	11	236	3
ITP1_3_A	558	285	236	15	17	5
ITP1_3_B	336	105	175	18	27	11
ITP1_3_C	255	32	108	54	55	6
ITP1_3_D	379	123	202	15	35	4
ITP1_4_A	56	0	32	23	0	1
ITP1_4_B	42	8	7	7	7	13
ITP1_4_C	85	27	43	7	5	3
ITP1_4_D	63	10	30	6	17	0
ITP1_5_A	59	13	28	3	15	0
ITP1_5_B	132	10	93	2	26	1
ITP1_5_C	81	14	39	4	24	0
ITP1_5_D	50	8	24	3	14	1
ITP1_6_A	83	31	37	3	9	3
ITP1_6_B	70	41	20	5	1	3
ITP1_6_C	142	85	32	11	12	2
ITP1_6_D	89	24	44	4	17	0
ITP1_7_A	123	39	37	21	21	5
ITP1_7_B	183	8	41	5	127	2
ITP1_7_C	89	24	50	8	6	1
ITP1_7_D	23	2	6	1	14	0
ITP1_8_A	66	30	26	5	1	4
ITP1_8_B	46	15	10	6	14	1
ITP1_8_C	51	9	27	5	5	5
ITP1_8_D	5	2	1	0	2	0

6.3 Experimental Results and Discussion

6.3.1 Results

Table 6.5 shows the correction performance of the proposed model. The proposed model improves the correction accuracy of source code in all programming tasks. To ensure that the result was not due to a statistical chance, statistical tests were performed on the correction accuracy of the conventional and proposed models. The calculated p-value is 6.27e-16, which satisfies the general significance level of p-value¡0.05, indicating that these results are not chance results. The average correction accuracy of the conventional model for each programming task was 13.90%. On the other hand, the average correction accuracy of the proposed model was 72.55%. The average correction accuracy of the proposed model is 58.64% higher than the conventional model without iterative trials. This shows that the accuracy of correcting logic errors can be improved substantially by using the proposed model that introduces iterative trials. This means that the proposed model was able to correct hidden logic errors that could not be detected in the first trial.

Table 6.5: Correction performance.

Datasets		Correction accuracy [%]		Number of edits		Execution time [s]		
Task ID	Targets	Conventional model	Proposed model	Users	Proposed model	Average	Min	Max
ITP1_1_A	1524	18.11	99.54	2.00	2.12	0.38	0.23	2.62
ITP1_1_B	630	5.08	97.14	2.15	6.72	0.82	0.25	4.51
ITP1_1_C	694	4.18	93.08	2.44	4.87	0.60	0.22	4.27
ITP1_1_D	500	4.00	88.60	2.74	3.01	0.51	0.34	5.17
ITP1_2_A	494	24.09	89.07	3.68	9.85	1.47	0.25	4.28
ITP1_2_B	727	1.38	93.95	2.90	4.33	0.56	0.26	5.12
ITP1_2_C	308	6.49	72.73	2.88	4.82	0.52	0.27	3.82
ITP1_2_D	350	9.43	79.14	4.20	4.08	0.86	0.54	3.67
ITP1_3_A	558	30.47	97.49	1.65	4.98	0.62	0.20	8.10
ITP1_3_B	336	23.81	85.42	2.34	2.91	0.38	0.21	3.44
ITP1_3_C	255	6.27	69.41	3.73	5.08	0.58	0.21	8.11
ITP1_3_D	379	7.65	94.46	2.04	2.45	0.60	0.46	3.75
ITP1_4_D	63	7.94	77.78	3.18	3.39	0.74	0.48	5.16
ITP1_4_B	42	16.67	64.29	5.04	5.00	0.55	0.23	5.34
ITP1_4_C	85	14.12	80.00	2.49	3.66	0.70	0.27	6.38
ITP1_4_D	63	7.94	77.78	3.18	3.39	0.74	0.48	5.16
ITP1_5_A	59	27.12	83.05	2.82	3.06	0.42	0.26	1.58
ITP1_5_B	132	3.03	66.67	3.53	5.62	0.45	0.23	3.50
ITP1_5_C	81	3.70	60.49	4.08	6.16	0.47	0.26	3.44
ITP1_5_D	50	16.00	80.00	3.15	3.30	0.63	0.31	4.71
ITP1_6_A	83	16.87	81.93	2.43	5.04	0.56	0.26	1.90
ITP1_6_B	70	27.14	44.29	3.71	4.32	0.35	0.27	3.24
ITP1_6_C	142	16.90	58.45	2.89	8.11	0.73	0.27	6.09
ITP1_6_D	89	19.10	56.18	3.84	4.58	0.49	0.46	4.01
ITP1_7_A	123	22.76	69.92	3.31	3.45	0.56	0.26	10.07
ITP1_7_B	183	1.09	64.48	5.28	6.36	0.63	0.27	7.26
ITP1_7_C	89	6.74	26.97	7.38	3.79	0.17	0.28	2.10
ITP1_7_D	23	13.04	47.83	6.64	2.91	0.41	0.48	1.66
ITP1_8_A	66	18.18	40.91	4.48	2.56	0.16	0.26	1.90
ITP1_8_B	46	6.52	34.78	7.12	3.81	0.44	0.26	2.73
ITP1_8_C	51	47.06	72.55	3.32	3.00	0.38	0.26	2.04
ITP1_8_D	5	20.00	60.00	4.00	3.00	0.56	0.68	1.06
Average		13.90	72.55	3.57	4.42	0.56	0.30	4.16

We compared the number of edit operations in the proposed model with the number of edit operations by the user. The number of edit operations by the user is the edit distance between the experimental data created by each user and the corresponding correct code. The average number of edit operations of the proposed model is slightly larger than that of the user. This means that the correction candidates indicated by the proposed model include correction candidates that do not need to be edited.

The experimental results show the average, minimum, and maximum execution time of the proposed model for each programming task. The average execution time for all programming tasks is less than 1.5 [s]. At the shortest, a given code can be corrected within 0.2 [s]. At the longest, a given code can be corrected in 10.07 [s]. These results show that the proposed model can debug logic errors in the source code within a reasonable timeframe.

Table 6.6 shows the correction accuracy for each number of logic errors in the source code for all programming tasks. The proposed model can correct all logic errors in the source code, even if there are multiple logic errors. This means that any logic errors that could not be detected in a first attempt were corrected in a later attempt. Therefore, this shows that the proposed model, which leverages iterative trials, is suitable as a debugging model for correcting multiple logic errors.

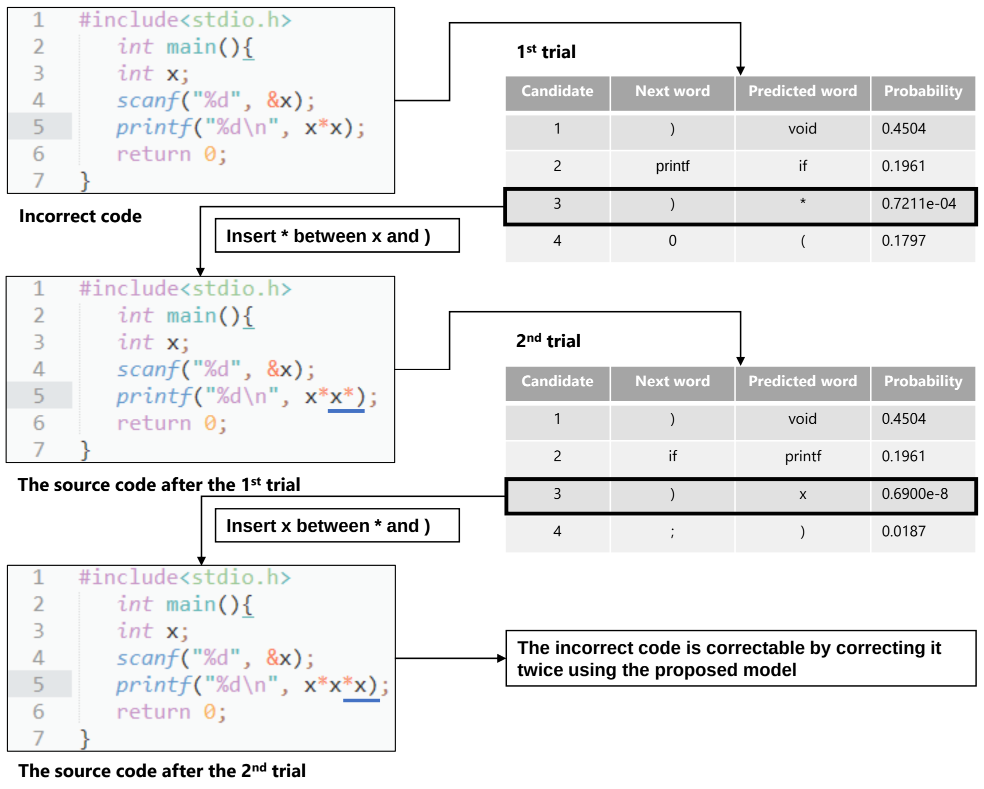
Table 6.6: Correction performance for each number of logic errors of all problems.

Datasets		Correction performance			
The number of logic errors	Targets	Conventional model ([%])		Proposed model ([%])	
1	1526	898	(58.8)	1304	(85.5)
2	4751	130	(2.7)	4305	(90.6)
3	373	6	(1.6)	291	(78.0)
4	1250	7	(0.6)	962	(77.0)
5	393	1	(0.3)	321	(81.7)

6.3.2 Application

Figure 6.3 shows an example of logic error correction using the proposed model for a simple programming task. The specification is to output the cube of a given integer. An example of incorrect code shown in the upper left of the figure outputs the square of the given integer. The logic error in this source code is "x*x" on the 5th line. Therefore, if " * x " is inserted in the source code, the source code will become correct.

The table on the upper right in Fig. 6.3 lists the prediction results for one application of the



```
#include<stdio.h>
    int main(){
    int x;
    scanf("%d", &x);
    printf("%d\n", x*x);
    return 0;
}
```

Incorrect code

1st trial

Candidate	Next word	Predicted word	Probability
1	)	void	0.4504
2	printf	if	0.1961
3	)	*	0.7211e-04
4	0	(	0.1797

Insert * between x and)

```
#include<stdio.h>
    int main(){
    int x;
    scanf("%d", &x);
    printf("%d\n", x*x*);
    return 0;
}
```

The source code after the 1st trial

2nd trial

Candidate	Next word	Predicted word	Probability
1	)	void	0.4504
2	if	printf	0.1961
3	)	x	0.6900e-8
4	;	)	0.0187

Insert x between * and)

```
#include<stdio.h>
    int main(){
    int x;
    scanf("%d", &x);
    printf("%d\n", x*x*x);
    return 0;
}
```

The source code after the 2nd trial

The incorrect code is correctable by correcting it twice using the proposed model

Figure 6.3: Example of logic error correction using proposed model.

proposed model to the incorrect code. The candidate with the lowest probability of occurrence for the next word is the position of “)”, and the proposed revision is “*”. This demonstrates that the model can correctly predict this logic error. In addition, the proposed model predicts “insert * between x and)” from this information. The table at the bottom right of the figure shows the correction candidates when the proposed model is applied a second time to the source code that has already been corrected once. The candidate with the lowest probability is the position of “)”, and the proposed amendment is “ x”. From the obtained information, the overall prediction by the proposed model is thus “insert x between * and)”. Therefore, it is possible to correct all logic errors in this source code by applying the proposed model.

In contrast, for the conventional model without iterative trials, the source code is corrected using only the correction candidates obtained from the first trial. In the first trial, “insert *” between “x” and “)” is performed, as in the case of the proposed model. However, since “x” is not present in the correction candidates in this first trial, not all logic errors in the source code can be corrected. Therefore, the conventional model can only correct some and not all of the logic errors in the incorrect code.

The aim of debugging support in programming education is to provide learners with hints for correcting their source code. However, giving too many hints to the learner can be problematic. Yi et al. [91] reported that novice programmers do not know how to modify programs efficiently using hints for correcting errors in the source code. This means that the hints must be adjusted according to the skills of the learner. Therefore, we suggest that it is possible to help individuals learn how to debug by showing not only correction candidates but also their editing operations.

The proposed model automatically modifies the source code based on the results obtained from each machine learning model. The modification of the proposed model can be replaced as a process to be performed by the user. Gradual hints related to the correction candidates and their editing operations by the proposed model can give the learners opportunities to think. Therefore, the model can control the quality of hints by displaying information according to the programming proficiency of the learner.

Novice programmers need debugging support in many situations. There are situations where such individuals do not know what or how to debug when they get the verdict that the source code is incorrect. The proposed model can show novice programmers whether the source code can be corrected by iterative modification. When the source code can be corrected, the correction candidates, the editing operations, and the number of edits obtained in each trial can be presented as hints to these individuals. They can then use these hints to correct the source code. On the other hand, debugging support for intermediate programmers can be achieved by giving partial hints obtained from the proposed model. For example, it may be sufficient to disclose only the position of the logic error as a hint. In this way, the intermediate learner needs to think about the editing operation for the given position. Instructors and other expert programmers could, for example, be provided with feedback from the proposed model with all details at the very beginning, and then use this to help students at their discretion depending on their assessment of individual needs. To apply the proposed model and obtained feedback to educational sites, we should carefully consider learning efficiency. Generally, the feedback should not be direct and immediate supports like those of conventional IDEs so that we can provide learners with chances to think and try to resolve the problem by themselves. The degree of such support should be controlled by learners or instructors according to their experience and learning modes.

The proposed model can also be used for software development. Generally, software consists of modules, packages, and subroutines. Implementations of these subroutines carry out numerical calculations and algorithm-level implementations to meet the specifications of a given task. It is necessary to use testing to verify whether the implemented subroutine is correct for the corresponding specification. Our model can be employed to modify the source code if there is a programming task with the same or similar specifications as the subroutine implemented in a certain educational system,

Although the proposed model is evaluated using source code written in C, it can be implemented in other programming languages by tokenization and learning the target source code. In addition, the proposed model can be used to detect syntactic errors because it is trained using correct and compilable source codes. For example, we believe that the proposed model can be applied to datasets such as Code4Bench [92], Codenet [52], and CodexGlue [93], where a lot of source codes are accumulated. To apply the proposed model to these datasets, it is necessary to prepare word lists and IDs corresponding to the words after tokenizing the source codes. The proposed model can be applied to these datasets by preparing the structures of LSTM-LM corresponding to the number of vocabularies and the length of input sequences.

The proposed model can be applied not only to programming languages but also to natural languages because it uses language models used for problem-solving in the field of NLP.

6.3.3 Limitations

These results show that the correction performance of the proposed model is high. However, focusing on detection performance and the number of trials, there is still room for improvement. We defined detection performance as the percentage of source codes in which a true logic error exists among the correction candidates obtained in the first trial. In the proposed model, a correction candidate most likely to be a logic error is selected and corrected. This means that if the top k correction candidates include true logic errors, it will be easier to correct those errors. Table 6.7 shows the detection performance of the proposed model. Task ID and Targets are the ID of each programming task and the number of experimental data. This shows the detection performance when the number of correction candidates is narrowed down to the top k. Where top $k = \infty$, this corresponds to all the correction candidates enumerated by the proposed model.

The detection performance of the proposed model is 95.54$\%$ when the top $k = \infty$. On the other hand, when the correction candidates are narrowed down to the top one, the true logic error can sometimes be missed. This means that true logic errors are less likely to appear when the correction candidates are narrowed down to the top k. The probabilities obtained from LSTM-LM are sufficient as a metric for detecting true logic errors in the source code. However, the probabilities may not be sufficient as a metric for selecting a correction candidate.

Table 6.7: Detection performance.

Datasets		Top-k [%]			
Task ID	Targets	$k = \infty$	$k = 1$	$k = 2$	$k = 3$
ITP1_1_A	1524	99.34	50.72	46.85	38.32
ITP1_1_B	630	99.84	90.79	85.08	43.17
ITP1_1_C	694	98.27	55.04	41.35	26.22
ITP1_1_D	500	99.60	47.00	28.40	8.20
ITP1_2_A	494	100.00	34.62	23.08	9.92
ITP1_2_B	727	100.00	54.61	44.70	30.12
ITP1_2_C	308	98.38	35.39	25.00	12.99
ITP1_2_D	350	98.57	24.00	9.43	2.57
ITP1_3_A	558	99.82	74.91	65.41	40.32
ITP1_3_B	336	98.51	45.54	22.92	7.44
ITP1_3_C	255	96.08	40.39	21.96	8.24
ITP1_3_D	379	100.00	57.26	33.77	14.78
ITP1_4_A	56	100.00	57.14	41.07	21.43
ITP1_4_B	42	92.86	19.05	11.90	2.38
ITP1_4_C	85	97.65	40.00	20.00	5.88
ITP1_4_D	63	98.41	12.70	3.17	0.00
ITP1_5_A	59	96.61	37.29	23.73	8.47
ITP1_5_B	132	96.21	19.70	9.09	1.52
ITP1_5_C	81	93.83	25.93	16.05	4.94
ITP1_5_D	50	100.00	14.00	6.00	2.00
ITP1_6_A	83	100.00	22.89	9.64	4.82
ITP1_6_B	70	90.00	7.14	2.86	1.43
ITP1_6_C	142	100.00	28.17	10.56	3.52
ITP1_6_D	89	96.63	10.11	3.37	1.12
ITP1_7_A	123	98.37	17.89	8.13	3.25
ITP1_7_B	183	98.91	17.49	14.21	7.10
ITP1_7_C	89	95.51	26.97	21.35	11.24
ITP1_7_D	23	100.00	4.35	0.00	0.00
ITP1_8_A	66	87.88	22.73	12.12	3.03
ITP1_8_B	46	95.65	26.09	21.74	13.04
ITP1_8_C	51	100.00	5.88	0.00	0.00
ITP1_8_D	5	100.00	0.00	0.00	0.00
Average		95.54	33.85	22.70	11.12

The correction candidates indicated by LSTM-LM are likely to be logic errors, but they are not always logic errors. To navigate this issue, one approach could be to narrow down the correction candidates by analyzing what kind of logic errors are likely to occur in each programming task.

Table 6.8 shows the performance of correcting the source code that the debugging support model could fix according to the classification of the places that contain logic errors introduced in Chapter 3. Total is the number of source codes classified by each type, while Corrected is the number and percentage of source codes fixed by the debug support model with iterative attempts. Corrected is the number and percentage of source code that the proposed model with iterative trials could correct. Not corrected is the number and percentage of source code that the proposed model with iterative trials. It can be seen that these source codes contain many logic errors related to string processing. Next, there are many errors caused by errors related to loops, conditional branches, and formulas. These analyses suggest that array_size and casts are difficult to correct. As for the correction performance, we can see that in most cases the correction performance is greater than 50%. However, not all each type of logic errors could be corrected, but the overall correction was found to be successful. This means that the correction performance of the debugging support model does not depend on the type of logic error. However, the classification of these logic errors is still incomplete, and a more detailed analysis is needed.

Table 6.8: Correction accuracy for logic error type.

Logic error type	Total	Corrected (%)		Not corrected (%)	
include_statement	100	91	(91.0)	9	(9.0)
switch_quote	145	102	(70.3)	43	(29.7)
string_format	5006	4618	(92.2)	388	(7.8)
output_format	549	442	(80.5)	107	(19.5)
input_statement	166	121	(72.9)	45	(27.1)
void_main_function	89	85	(95.5)	4	(4.5)
return_value	26	22	(84.6)	4	(15.4)
for_statement	654	553	(84.6)	101	(15.4)
if_statement	1137	898	(79.0)	239	(21.0)
else_statement	36	23	(63.9)	13	(36.1)
formula	541	354	(65.4)	187	(34.6)
do_while_statement	14	11	(78.6)	3	(21.4)
while_statement	92	73	(79.3)	19	(20.7)
function	22	14	(63.6)	8	(36.4)
cast	1	0	(0.0)	1	(100.0)
type	96	66	(68.8)	30	(31.2)
array_size	82	16	(19.5)	66	(80.5)
variable_declaration	93	68	(73.1)	25	(26.9)
unary_operation	30	27	(90.0)	3	(10.0)
scope	157	103	(65.6)	54	(34.4)
switch_break_continue	20	13	(65.0)	7	(35.0)
semicolon_position	11	9	(81.8)	2	(18.2)
assignment_operation	14	9	(64.3)	5	(35.7)
No_classification	2	2	(100.0)	0	(0.0)

In addition, the proposed model is a specialized model that can identify the logic errors that occur in each problem. Therefore, there is a problem of not being able to correct logic errors other than those corresponding to the source code used for training. A solution to these problems is the pre-trained language model such as BERT [35], which is adaptable to a variety of tasks. In addition, a debugging support model has been developed that can be applied to a variety of problems by embedding program IDs as input data. By building a model based on these models, we believe it is possible to develop a generalized debugging support model rather than a specialized model.

6.4 Chapter Summary

This chapter proposed a debugging model for correcting logic errors in a given source code. The model could correct multiple logic errors by repeatedly identifying and correcting errors, and testing the source code. In the experiment, we applied the proposed model to 32 programming tasks and the corresponding solution codes in an online judge system to verify the advantage of the proposed model, . By comparing the proposed model with another model without iterative trials, the results showed that the correction accuracy of the proposed model improved by 58.64% on average. In addition, this model can suggest an editing operation for correcting a source code depending on the features around the detection location in the process of the iterative trials. The proposed model can also control the granularity of hints according to the proficiency of programmers and learners. Therefore, the proposed model considers educational effectiveness and can be applied to e-learning systems that support education not only in programming but also in related subjects. This research has been published in *Applied Sciences* by Multidisciplinary Digital Publishing Institute (MDPI) [94].

Chapter 7

Application

In Chapter 7, we discuss an integrated debugging support model by combining the proposed approaches. Section 7.1 introduces an integrated debugging support model that combines proposed approaches. Section 7.2 describes the use-cases of the integrated debugging support model in education and software engineering. Section 7. 3 mentions the limitation of the model. Finally, Section 7.4 summarizes Chapter 7.

7.1 Integrated Debugging Support Model

Figure 7.1 shows an overview of the integrated debugging support model that supports debugging by combining the proposed methods in this dissertation. The model is a hybrid intelligence that combines SPED and LSTM-LM and provides detection and correction results as debugging information to programmers. In order to use the integrated debugging support model, it is necessary to prepare a software repository containing correct and incorrect code created to solve a programming task. SPED uses structure patterns based on AST using correct codes extracted from the software repository. On the other hand, LSTM-LM trains the internal parameters using correct codes. The EOP trains the internal parameters using the edit information between pairs of an incorrect code and the corresponding correct code created by each user in the software repository. Finally, a threshold is determined to control the number of correction candidates based on the results of using a set of incorrect codes.

When using the integrated debugging support model, the programmer provides a target source code as input. SPED searches for the correct code that has the same structure as the target code from the source codes stored in the data repository. If found, the correct code and

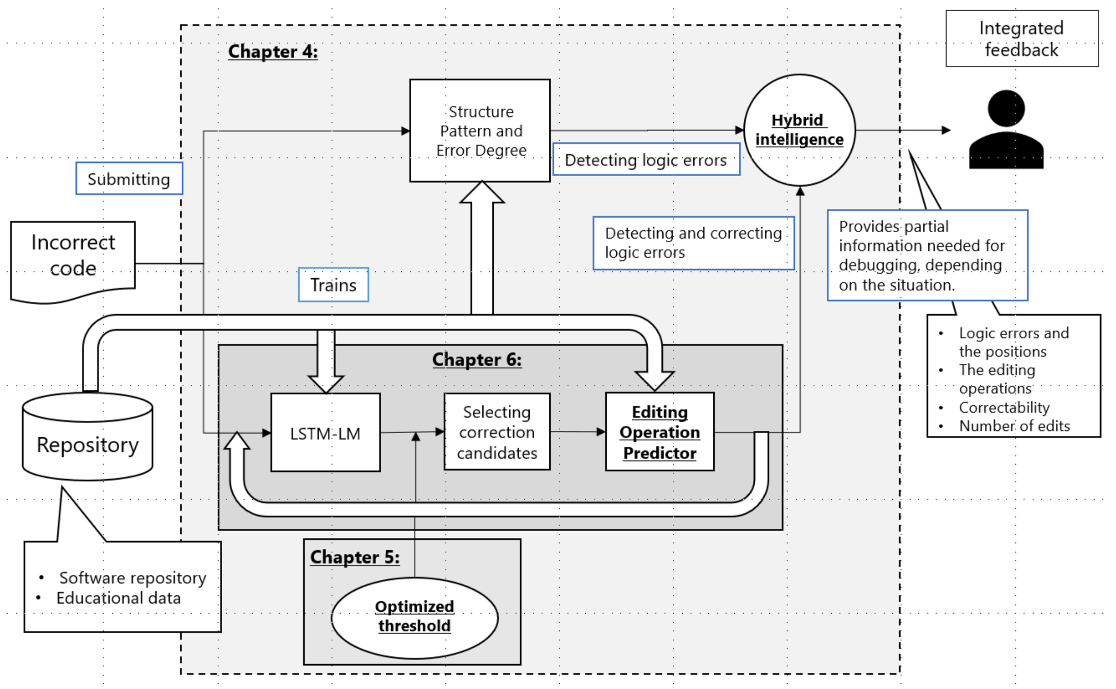


Figure 7.1: An overview of a integrated debugging support model.

the target are compared based on the structure of the abstract syntax tree, and the location of logic errors and their error types are presented. On the other hand, when the target is input to LSTM-LM, it outputs a probability distribution for the tokens in each time series. If the input and output data differ in each time series, it is considered highly likely to be a logic error, and those tokens are added to the list as candidates for correction. The token with the highest probability of being a logic error among the added tokens is entered as a correction candidate. The EOP then predicts the edit operation that should be performed on the candidate tokens. Based on the prediction, the candidate tokens are corrected. After correction, the tokens are tested to see if they meet the specifications of the programming task. If the test results indicate that the correction is not complete, the corrected source code is input to LSTM-LM. Finally, the results obtained by SPED and LSTM-LM are integrated by hybrid intelligence and provided to the programmer. This serves to show whether the source code submitted by the programmer can be corrected by SPED and LSTM-LM.

We also believe that the threshold selection method introduced in Section 5 provides debugging information to the programmer: if the source code detected and corrected by SPED and LSTM-LM is presented as it is, the programmer will be able to modify the source code to meet the specification of the programming task if the source code is corrected as it is. The programmer can modify the source code to meet the specification of the programming task.

However, the programmer, whose goal is to learn programming, may lose the opportunity to develop the problem-solving skills and logical thinking necessary to debug a program. Here, we use the threshold values obtained with the proposed method, which allows us to control the correction candidates; the correction candidates obtained by LSTM-LM are stored in a list. The programmer needs to appropriately select from among the correction candidates the one that satisfies the specification of the programming task. When using the debugging support model introduced in Section 6, one is selected based on the probability distribution obtained by LSTM-LM. However, if a programmer modifies the source code using this selection method, he/she can modify the source code to satisfy the use of the programming task without deep consideration. To prevent this, we control the correction candidates based on a threshold without displaying the probability assigned to the correction candidate. This provides the programmer with the task of appropriately selecting one of the correction candidates. Therefore, this realizes an interactive interaction between the programmer and the debugging support model. The automatic correction provides direct and immediate debugging support, but immediate and non-direct debugging support is difficult. Therefore, we believe that the proposed method solves different requirements in software engineering and programming education.

7.2 Use-Case

7.2.1 Educational Scene

The debugging support model introduced in Chapter 6 enables automatic correction by repeatedly detecting and correcting logic errors in the source code. Therefore, the model provides the source code in which logic errors are corrected to a learner when the learner inputs an incorrect code into the model. However, providing the source code may deprive programming learners of the opportunity to develop the logical thinking necessary to solve problems. To solve this problem, we leave the process of correcting the source code in our model to the learner's judgment. This means that the learner needs to think about which candidate to correct from the multiple candidates obtained from the LSTM-LM with thresholds set. In addition, each learner has a bias in programming skills, and it is necessary to provide debugging support tailored to that learner. The integrated debugging model can provide information for debugging, such as correction candidates, their editing operations, the number of edits until the correct code, and correctability. The proposed model can also control the granularity of hints according to the proficiency of programmers and learners.

Proposed methods support debugging for instructors that support learners. In general, an instructor must judge whether a source code created by the learner is correct or not. If the source code is incorrect, instructors need to support the learner in debugging a source code. Our integrated debugging support model can support this process. First, the model provides whether the model can correct the source code. Then, based on the results, it can quickly identify which locations in the incorrect source code are incorrect. The instructor provides the learner with the information necessary to debug the source code. This can be accomplished with or without the instructor's programming skills to support the learner. Therefore, this means that our model reduces the number of people involved in debugging support and the debugging time.

To validate the effectiveness of our methods, we used the AOJ dataset, which has accumulated a large number of incorrect and correct codes created to solve many programming tasks. If a software repository that contains source code created by learners to solve a programming task exists, it is possible to provide debugging support based on logic error detection and correction using the proposed method. Therefore, these methods can be applied to open data such as CodeNet and CodeXGlue, which are currently available to the public.

7.2.2 Software Engineering

To use the integrated debugging support model, it is necessary to use a software repository that contains source code written in the same programming language in a specific discipline. In software engineering, there are no specific tasks in programming education, but a combination of tasks. Therefore, it is difficult to apply this model to these tasks because they are not as simple as the tasks in programming education. However, they may consist of simple functions. This means that large source codes are a set of simple specifications, and software is built by using those functions. Therefore, we believe that at the level of simple functions, the debugging support model can detect logical errors within those functions. Furthermore, since the proposed method is a model that learns a specific task, it can be used to correct logic errors in areas related to the training data by changing the training data to a different data set.

7.3 Limitation

In this section, we describe the limitation of the integrated debugging support model. The model depends on the quality and quantity of the stored source codes, and in the case of SPED, it is necessary to search for correct codes that have the same structure as the incorrect codes in the database of correct codes. Since the algorithm is applied based on a comparison with the structure of the retrieved correct code, the detection result cannot be output if it cannot be retrieved. In addition, LSTM-LM learns internal parameters by using a set of correct codes stored in the database. Therefore, LSTM-LM detects and corrects logic errors based on the structure of the correct codes. However, LSTM-LM is not perfect in detecting and correcting logic errors. It is not possible to determine whether the detected correction candidates and corrected source code are correctly corrected by only LSTM-LM. Therefore, it is necessary to test separately whether the detected correction candidates and corrected source code meet the specifications of the programming task.

7.4 Chapter Summary

This chapter describes the applications and limitations of the integrated debugging support model by combining proposed methods in this dissertation. Since the model is built based on the accumulated source code, it is dependent on the quality and quantity of the source code. Therefore, the accuracy of detecting and correcting logic errors may not be significant if there is not enough data to train the machine learning model and the source code set required for retrieval in SPED. However, an online programming education system such as AOJ, which is capable of collecting data, stores source code and its metadata in its internal database. This is expected to improve the accuracy of detection and correction of programming tasks for which the current source code set does not improve accuracy.

Chapter 8

Conclusion and future works

8.1 Summary

Debugging logic errors in source code is one of the most difficult tasks in software development and programming education. From the standpoint of debugging support, if logic errors are falsely detected by the logic error detection method, the user may make incorrect modifications using them. In addition, although a use case for a logic error detection method based on machine learning has been presented, the performance of correcting logic errors using these methods has not been verified in practice. In addition, a debugging support model has not yet been developed for correcting logic errors based on this method.

In this dissertation, we elucidated the issues related to existing logic error detection methods and developed a debugging support model that corrects multiple logic errors for debugging support in software engineering and programming education. The contributions of this dissertation are as follows:

- We analyzed the detection performance of a SPED and an LSTM-LM using the AOJ dataset to clarify the limitations of static analysis and machine learning-based logic error detection methods. Experimental results show that LSTM-LM is less reliable than SPED in detecting whether a correction candidate contains a logic error. On the other hand, SPED cannot detect logic errors unless a correct code with the same structure as the target source code exists. These limitations indicated that there is a trade-off between detection accuracy and reliability. We also confirmed that each method could detect logic errors that either method could not. Therefore, the combination of the SPED and the LSTM-LM realize a basis for developing hybrid intelligence that accurately detects logic errors and provides appropriate feedback with higher accuracy.

- We proposed a method that optimizes a threshold that regulates the number of correction candidates detected by LSTM-LM to improve the reliability of the correction candidates. This threshold is optimized based on the detection performance (only detection, no detection, misdetection, overdetection) obtained by LSTM-LM for less than each probability in [0.0 1.0] using a set of incorrect codes. Experimental results show that the optimized threshold is better than the threshold in previous research in narrowing down the correction candidates that include only logic errors for each programming task. In addition, the detection performance visualizes the performance of the language models used to detect possible logic errors in each programming. This is useful for showing learners the reliability of the correction candidates detected by the LSTM-LM in advance.

- We developed a debugging support model that introduced an EOP that predicts an editing operation, such as insertion, deletion, and replacement, using the correction candidates detected by the LSTM-LM. The model iteratively corrects the incorrect code until it meets the specification of the programming task. In an experiment, the proposed model with iterative trials improved the average correction accuracy by 58.64% compared to the model without iterative trials. In addition, the number of corrections by the proposed model is smaller than the number of actual edits by the user. The proposed model can provide the correction candidates, their editing operations, and the number of edits. This is indirect and immediate debugging support, but it allows the learner to lose the opportunity to develop the logical thinking necessary to solve the programming task.

The proposed method and model in this dissertation can be applied to debugging support in programming education and in the fields of software engineering and natural language processing. In software engineering, it is necessary to use testing to verify whether the implemented subroutine is correct for the corresponding specification. If there is a programming task with the same or similar specifications as the subroutine implemented in a certain educational system, our model can be employed to modify the source code. The proposed method can also be applied to sentence generation and error correction in time-series data and natural language since it uses data accumulated in software repositories.

8.2 Future Work

Future directions of this research include:

- To develop the generalized deep learning-based model for logic error detection
- To develop the deep learning-based logic error model for multiple programming languages, such as C, C++, Java, Python, and so on.
- To realize Hybrid Intelligence for logic error detection.
- To evaluate the detection performance of integrated debugging support model applying to other datasets such as CodeNet, other educational data etc..
- To evaluate educational effects of integrated debugging support model

Acknowledgment

I have been ten years and a half since I entered The University of Aizu and three years since I began my research on the machine learning-based bug detection and debugging support model.

First of all, I would like to acknowledge my research supervisor, Professor Yutaka Watanobe, for his constant support, advice, and encouragement throughout my doctoral program. I have known Professor Watanobe since my first year as an undergraduate, and he is the advisor of the international collegiate programming contest (ICPC), to which I belonged at the time. Although I was a member of ICPC for a short time, my experience at ICPC was a chance for me to learn the joy of programming. Professor Watanobe graciously accepted me into his laboratory, as I did not know anything about software engineering and programming education. I have failed and disappointed him many times, but he has been very kind and patient with me. I am grateful to Professor Watanobe for giving me the topic for my research on debugging support for Aizu Online Judge. Professor Watanobe, I look forward to working with you in the future.

I would also like to thank the members of my doctoral dissertation reviewing committee – Professor Incheon Paik, Professor Rentaro Yoshioka, Professor Keita Nakamura, and Professor Emeritus Alexander Vazhenin – who have given me valuable suggestions, insightful remarks, constructive criticisms, and advice on the doctoral dissertation.

I really have to thank Professor Keita Nakamura and Professor Jun Ogawa. From the first year of my master's degree to the present, Professor Nakamura has discussed my research and corrected my papers. He is also a gentle teacher who takes me seriously and scolds me. On the other hand, Professor Jun Ogawa has taken care of me from the second year of my master's degree to the first year of my doctoral degree. Seeing him enjoying his research made me want to get a Ph.D.

I am grateful to my friend Tomohiro Saito, who introduced me to Professor Watanobe. When I was thinking about changing labs at the end of my first year of the doctoral course, he was kind enough to give me advice. Without his support, I might have given up on getting my

Ph.D.

I am also grateful to members of the Robot Engineering Laboratory (formerly the Image Processing Laboratory). I am what I am today because of my daily accumulation of experiences with the members of this laboratory. I wanted to do a doctorate because my time with them was so fulfilling. The days I spent with them are still a precious treasure to me.

I would like to thank members of the Office for Learning Support (OFLS) at The University of Aizu. I have worked as a teaching assistant for five years and a half from the second year of my master's program to the present, and my interaction with everyone at OFLS has greatly supported me as a doctoral student.

I would like to thank everyone who supported me at The University of Aizu. I feel that my days at The University of Aizu were very fulfilling for me because of my encounters with them.

I am also grateful for the support of the Japan Student Services Organization (JASSO). JASSO appreciated my educational and research activities and waived half of the scholarship for my bachelor's, master and doctoral programs. I spent a fulfilling life at The University of Aizu.

Finally, I was born and raised in Aizu-Wakamatsu. I am grateful to my mother Teruko and my brother Jun for their kind support in my life.

References

[1] "A Collection of Well-Known Software Failures," available online: http://www.cse.psu.edu/~gxt29/bug/softwarebug.html (accessed on August 2, 2022).

[2] I. Sommerville, "Software engineering 10th edition," *ISBN-10*, vol. 137035152, p. 18, 2015.

[3] B. Du Boulay, "Some difficulties of learning to program," *Journal of Educational Computing Research*, vol. 2, no. 1, pp. 57–73, 1986.

[4] T. Staubitz, H. Klement, J. Renz, R. Teusner, and C. Meinel, "Towards practical programming exercises and automated assessment in massive open online courses," in *2015 IEEE International Conference on Teaching, Assessment, and Learning for Engineering (TALE)*. IEEE, 2015, pp. 23–30.

[5] T. Crow, A. Luxton-Reilly, and B. Wuensche, "Intelligent tutoring systems for programming education: a systematic review," in *Proceedings of the 20th Australasian Computing Education Conference*, 2018, pp. 53–62.

[6] S. Wasik, M. Antczak, J. Badura, A. Laskowski, and T. Sternal, "A survey on online judge systems and their applications," *ACM Computing Surveys (CSUR)*, vol. 51, no. 1, pp. 1–34, 2018.

[7] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo, "The oracle problem in software testing: A survey," *IEEE transactions on software engineering*, vol. 41, no. 5, pp. 507–525, 2014.

[8] Y. Watanobe, "Aizu Online Judge." available online: https://onlinejudge.u-aizu.ac.jp/ (accessed on August 2, 2022).

[9] "AtCoder.jp," available online: https://atcoder.jp/ (accessed on August 2, 2022).

[10] "GDB: The GNU Project Debugger," available online: https://www.sourceware.org/gdb/ (accessed on August 2, 2022).

[11] J. T. Stasko, "Tango: A framework and system for algorithm animation," *Computer*, vol. 23, no. 9, pp. 27–39, 1990.

[12] A. Luxton-Reilly, I. Albluwi, B. A. Becker, M. Giannakos, A. N. Kumar, L. Ott, J. Paterson, M. J. Scott, J. Sheard, and C. Szabo, "Introductory programming: a systematic literature review," in *Proceedings Companion of the 23rd Annual ACM Conference on Innovation and Technology in Computer Science Education*, 2018, pp. 55–106.

[13] Y. Yoshizawa and Y. Watanobe, "Logic Error Detection System based on Structure Pattern and Error Degree," *Advances in Science, Technology and Engineering Systems Journal*, vol. 4, no. 5, pp. 1–15, 2019.

[14] Y. Teshima and Y. Watanobe, "Bug detection based on lstm networks and solution codes," in *2018 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*. IEEE, 2018, pp. 3541–3546.

[15] "Java Platform Debugger Architecture (JPDA)," available online: https://www.docs.oracle.com/avase/8/docs/technotes/guides/jpda/index.html (accessed on August 2, 2022).

[16] A. Moreno, N. Myller, E. Sutinen, and M. Ben-Ari, "Visualizing programs with jeliot 3," in *Proceedings of the working conference on Advanced visual interfaces*, 2004, pp. 373–376.

[17] A. Gosain and G. Sharma, "Static analysis: A survey of techniques and tools," in *Intelligent Computing and Applications*. Springer, 2015, pp. 581–591.

[18] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa, "A survey on software fault localization," *IEEE Transactions on Software Engineering*, vol. 42, no. 8, pp. 707–740, 2016.

[19] B. A. Becker, K. Goslin, and G. Glanville, "The effects of enhanced compiler error messages on a syntax error debugging test," in *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*, 2018, pp. 640–645.

[20] P. Denny, A. Luxton-Reilly, and D. Carpenter, "Enhancing syntax error messages appears ineffectual," in *Proceedings of the 2014 conference on Innovation & technology in computer science education*, 2014, pp. 273–278.

[21] M. Weiser, "Program slicing," *IEEE Transactions on software engineering*, no. 4, pp. 352–357, 1984.

[22] H. Agrawal, R. A. DeMillo, and E. H. Spafford, "Debugging with dynamic slicing and backtracking," *Software: Practice and Experience*, vol. 23, no. 6, pp. 589–616, 1993.

[23] W. E. Wong and Y. Qi, "Effective program debugging based on execution slices and inter-block data dependency," *Journal of Systems and Software*, vol. 79, no. 7, pp. 891–903, 2006.

[24] J. Lee, D. Song, S. So, and H. Oh, "Automatic diagnosis and correction of logical errors for functional programming assignments," *Proceedings of the ACM on Programming Languages*, vol. 2, no. OOPSLA, pp. 1–30, 2018.

[25] R. Abreu, P. Zoeteweij, and A. J. Van Gemund, "On the accuracy of spectrum-based fault localization," in *Testing: Academic and industrial conference practice and research techniques-MUTATION (TAICPART-MUTATION 2007)*. IEEE, 2007, pp. 89–98.

[26] P. F. Brown, V. J. Della Pietra, P. V. Desouza, J. C. Lai, and R. L. Mercer, "Class-based n-gram models of natural language," *Computational linguistics*, vol. 18, no. 4, pp. 467–480, 1992.

[27] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," *arXiv preprint arXiv:1301.3781*, 2013.

[28] J. L. Elman, "Finding structure in time," *Cognitive science*, vol. 14, no. 2, pp. 179–211, 1990.

[29] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[30] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," in *Advances in neural information processing systems*, 2014, pp. 3104–3112.

[31] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in neural information processing systems*, 2017, pp. 5998–6008.

[32] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," *arXiv preprint arXiv:1409.0473*, 2014.

[33] X. Qiu, T. Sun, Y. Xu, Y. Shao, N. Dai, and X. Huang, "Pre-trained models for natural language processing: A survey," *Science China Technological Sciences*, pp. 1–26, 2020.

[34] A. Wang, A. Singh, J. Michael, F. Hill, O. Levy, and S. R. Bowman, "Glue: A multi-task benchmark and analysis platform for natural language understanding," *arXiv preprint arXiv:1804.07461*, 2018.

[35] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[36] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell *et al.*, "Language models are few-shot learners," *arXiv preprint arXiv:2005.14165*, 2020.

[37] L. Ouyang, J. Wu, X. Jiang, D. Almeida, C. L. Wainwright, P. Mishkin, C. Zhang, S. Agarwal, K. Slama, A. Ray *et al.*, "Training language models to follow instructions with human feedback," *arXiv preprint arXiv:2203.02155*, 2022.

[38] M. Allamanis, E. T. Barr, P. Devanbu, and C. Sutton, "A survey of machine learning for big code and naturalness," *ACM Computing Surveys (CSUR)*, vol. 51, no. 4, pp. 1–37, 2018.

[39] A. Svyatkovskiy, S. K. Deng, S. Fu, and N. Sundaresan, "Intellicode compose: Code generation using transformer," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 1433–1443.

[40] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang *et al.*, "Codebert: A pre-trained model for programming and natural languages," *arXiv preprint arXiv:2002.08155*, 2020.

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

[42] T. Chappelly, C. Cifuentes, P. Krishnan, and S. Gevay, "Machine learning for finding bugs: An initial report," in *2017 IEEE Workshop on Machine Learning Techniques for Software Quality Evaluation (MaLTeSQuE)*. IEEE, 2017, pp. 21–26.

[43] S. Jadon, "Code clones detection using machine learning technique: Support vector machine," in *2016 International Conference on Computing, Communication and Automation (ICCCA)*. IEEE, 2016, pp. 399–303.

[44] A. Miltiadis, E. Barr, D. Premkumar, and S. Charles, "A survey of machine learning for big code and naturalness," *ACM Comput. Surv*, vol. 51, pp. 1–37, 2018.

[45] A. Kaur, K. Kaur, and D. Chopra, "Entropy based bug prediction using neural network based regression," in *International Conference on Computing, Communication & Automation*. IEEE, 2015, pp. 168–174.

[46] S. Wang, D. Chollak, D. Movshovitz-Attias, and L. Tan, "Bugram: bug detection with n-gram language models," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, 2016, pp. 708–719.

[47] "GitHub Copilot," available online: https://copilot.github.com/ (accessed on August 2, 2022).

[48] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. d. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman *et al.*, "Evaluating large language models trained on code," *arXiv preprint arXiv:2107.03374*, 2021.

[49] Y. Li, D. Choi, J. Chung, N. Kushman, J. Schrittwieser, R. Leblond, T. Eccles, J. Keeling, F. Gimeno, A. D. Lago *et al.*, "Competition-level code generation with alphacode," *arXiv preprint arXiv:2203.07814*, 2022.

[50] S. Lu, D. Guo, S. Ren, J. Huang, A. Svyatkovskiy, A. Blanco, C. Clement, D. Drain, D. Jiang, D. Tang *et al.*, "Codexglue: A machine learning benchmark dataset for code understanding and generation," *arXiv preprint arXiv:2102.04664*, 2021.

[51] "GCJ-dataset," available online: https://openreview.net/attachment?id=AZ4vmLoJft&name=supplementary_material. (accessed on August 2, 2022).

[52] R. Puri, D. S. Kung, G. Janssen, W. Zhang, G. Domeniconi, V. Zolotov, J. Dolby, J. Chen, M. Choudhury, L. Decker *et al.*, "Codenet: A large-scale ai for code dataset for learning a diversity of coding tasks," in *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 2)*, 2021.

[53] M. Monperrus, "The living review on automated program repair," 2020.

[54] Y. Pu, K. Narasimhan, A. Solar-Lezama, and R. Barzilay, "sk_p: a neural program corrector for moocs," in *Companion Proceedings of the 2016 ACM SIGPLAN International Conference on Systems, Programming, Languages and Applications: Software for Humanity*, 2016, pp. 39–40.

[55] H. Cao, Y. Meng, J. Shi, L. Li, T. Liao, and C. Zhao, "A survey on automatic bug fixing," in *2020 6th International Symposium on System and Software Reliability (ISSSR)*. IEEE, 2020, pp. 122–131.

[56] D. Drain, C. Wu, A. Svyatkovskiy, and N. Sundaresan, "Generating bug-fixes using pretrained transformers," *arXiv preprint arXiv:2104.07896*, 2021.

[57] Y. Ueda, T. Ishio, A. Ihara, and K. Matsumoto, "Devreplay: Automatic repair with editable fix pattern," *arXiv preprint arXiv:2005.11040*, 2020.

[58] R. Gupta, S. Pal, A. Kanade, and S. Shevade, "Deepfix: Fixing common c language errors by deep learning," in *Proceedings of the aaai conference on artificial intelligence*, 2017.

[59] H. Hajipour, A. Bhattacharya, and M. Fritz, "Samplefix: Learning to correct programs by sampling diverse fixes," *arXiv preprint arXiv:1906.10502*, 2019.

[60] S. Huang, X. Zhou, and S. Chin, "Application of seq2seq models on code correction," *Frontiers in artificial intelligence*, vol. 4, p. 3, 2021.

[61] R. Gupta, A. Kanade, and S. Shevade, "Neural attribution for semantic bug-localization in student programs," in *Advances in Neural Information Processing Systems*, 2019, pp. 11 861–11 871.

[62] M. Vasic, A. Kanade, P. Maniatis, D. Bieber, and R. Singh, "Neural program repair by jointly learning to localize and repair," *arXiv preprint arXiv:1904.01720*, 2019.

[63] B. Berabi, J. He, V. Raychev, and M. Vechev, "Tfix: Learning to fix coding errors with a text-to-text transformer," in *International Conference on Machine Learning*. PMLR, 2021, pp. 780–791.

[64] Y. Yoshizawa and Y. Watanobe, "Logic error detection algorithm for novice programmers based on structure pattern and error degree," in *2018 9th International Conference on Awareness Science and Technology (iCAST)*. IEEE, 2018, pp. 297–301.

[65] M. Rahman, Y. Watanobe, K. Nakamura *et al.*, "Source code assessment and classification based on estimated error probability using attentive lstm language model and its application in programming education," *Applied Sciences*, vol. 10, no. 8, p. 2973, 2020.

[66] ——, "A neural network based intelligent support model for program code completion," *Scientific Programming*, vol. 2020, 2020.

[67] M. M. Rahman, Y. Watanobe, and K. Nakamura, "A bidirectional lstm language model for code evaluation and repair," *Symmetry*, vol. 13, no. 2, 2021. [Online]. Available: https://www.mdpi.com/2073-8994/13/2/247

[68] K. Terada and Y. Watanobe, "Code completion for programming education based on recurrent neural network," in *2019 IEEE 11th International Workshop on Computational Intelligence and Applications (IWCIA)*. IEEE, 2019, pp. 109–114.

[69] Y. Watanobe, "Development and operation of an online judge system," *IPSJ Magazine*, vol. 56, no. 10, pp. 998–1005, 2015.

[70] Y. Watanobe, M. M. Rahman, T. Matsumoto, U. K. Rage, and P. Ravikumar, "Online judge system: Requirements, architecture, and experiences," *International Journal of Software Engineering and Knowledge Engineering*, vol. 32, no. 06, pp. 917–946, 2022.

[71] Y. Watanobe, C. Intisar, R. Cortez, and A. Vazhenin, "Next-generation programming learning platform: Architecture and challenges," *SHS Web Conf.*, vol. 77, p. 01004, 2020. [Online]. Available: https://doi.org/10.1051/shsconf/20207701004

[72] K. Terada and Y. Watanobe, "Automatic generation of fill-in-the-blank programming problems," in *2019 IEEE 13th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC)*. IEEE, 2019, pp. 187–193.

[73] M. Rahman, Y. Watanobe, R. U. Kiran, R. Kabir *et al.*, "A stacked bidirectional lstm model for classifying source codes built in mpls," in *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*. Springer, 2021, pp. 75–89.

[74] S. Kawabayashi, M. M. Rahman, and Y. Watanobe, "A model for identifying frequent errors in incorrect solutions," *2021 10th International Conference on Educational and Information Technology (ICEIT)*, pp. 258–263, 2021.

[75] M. M. Rahman, S. Kawabayashi, and Y. Watanobe, "Categorization of frequent errors in solution codes created by novice programmers," in *SHS Web of Conferences*, vol. 102. EDP Sciences, 2021, p. 04014.

[76] H. Ohashi and Y. Watanobe, "Convolutional neural network for classification of source codes," in *2019 IEEE 13th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC)*. IEEE, 2019, pp. 194–200.

[77] T. Saito and Y. Watanobe, “Learning path recommendation system for programming education based on neural networks,” *International Journal of Distance Education Technologies*, vol. 18, pp. 36–64, 01 2020.

[78] C. M. Intisar and Y. Watanobe, “Classification of online judge programmers based on rule extraction from self organizing feature map,” in *2018 9th International Conference on Awareness Science and Technology (iCAST)*, 2018, pp. 313–318.

[79] M. M. Rahman, Y. Watanobe, R. U. Kiran, T. C. Thang, and I. Paik, “Impact of practical skills on academic performance: A data-driven analysis,” *IEEE Access*, vol. 9, pp. 139 975–139 993, 2021.

[80] M. M. Rahman, Y. Watanobe, T. Matsumoto, R. U. Kiran, and K. Nakamura, “Educational data mining to support programming learning using problem-solving data,” *IEEE Access*, vol. 10, pp. 26 186–26 202, 2022.

[81] “Aizu Online Judge. Developers Site (API),” available online: http://developers.u-aizu.ac.jp/index (accessed on August 2, 2022).

[82] Y. Bengio, R. Ducharme, P. Vincent, and C. Jauvin, “A neural probabilistic language model,” *Journal of machine learning research*, vol. 3, no. Feb, pp. 1137–1155, 2003.

[83] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, “Dropout: a simple way to prevent neural networks from overfitting,” *The journal of machine learning research*, vol. 15, no. 1, pp. 1929–1958, 2014.

[84] D. P. Kingma and J. Ba, “Adam: A method for stochastic optimization,” *arXiv preprint arXiv:1412.6980*, 2014.

[85] T. Matsumoto and Y. Watanobe, “Towards hybrid intelligence for logic error detection,” in *Advancing Technology Industrialization Through Intelligent Software Methodologies, Tools and Techniques: Proceedings of the 18th International Conference on New Trends in Intelligent Software Methodologies, Tools and Techniques (SoMeT_19)*, vol. 318. IOS Press, 2019, p. 120.

[86] H. Hotelling *et al.*, “The generalization of student’s ratio,” *The Annals of Mathematical Statistics*, vol. 2, no. 3, pp. 360–378, 1931.

[87] T. Matsumoto, Y. Watanobe, K. Nakamura, and Y. Teshima, “Logic error detection algorithm based on rnn with threshold selection,” in *Knowledge Innovation Through Intelligent Software Methodologies, Tools and Techniques: Proceedings of the 19th International Conference on New Trends in Intelligent Software Methodologies, Tools and Techniques (SoMeT_20)*, vol. 327. IOS Press, 2020, pp. 76–87.

[88] V. Vapnik, “Pattern recognition using generalized portrait method,” *Automation and remote control*, vol. 24, pp. 774–780, 1963.

[89] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, “TensorFlow: Large-scale machine learning on heterogeneous systems,” 2015, available online: https://www.tensorflow.org/ (accessed on August 2, 2022).

[90] 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, pp. 2825–2830, 2011.

[91] J. Yi, U. Z. Ahmed, A. Karkare, S. H. Tan, and A. Roychoudhury, "A feasibility study of using automated program repair for introductory programming assignments," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, 2017, pp. 740–751.

[92] A. Majd, M. Vahidi-Asl, A. Khalilian, A. Baraani-Dastjerdi, and B. Zamani, "Code4bench: A multidimensional benchmark of codeforces data for different program analysis techniques," *Journal of Computer Languages*, vol. 53, pp. 38–52, 2019.

[93] S. Lu, D. Guo, S. Ren, J. Huang, A. Svyatkovskiy, A. Blanco, C. B. Clement, D. Drain, D. Jiang, D. Tang, G. Li, L. Zhou, L. Shou, L. Zhou, M. Tufano, M. Gong, M. Zhou, N. Duan, N. Sundaresan, S. K. Deng, S. Fu, and S. Liu, "Codexglue: A machine learning benchmark dataset for code understanding and generation," *CoRR*, vol. abs/2102.04664, 2021.

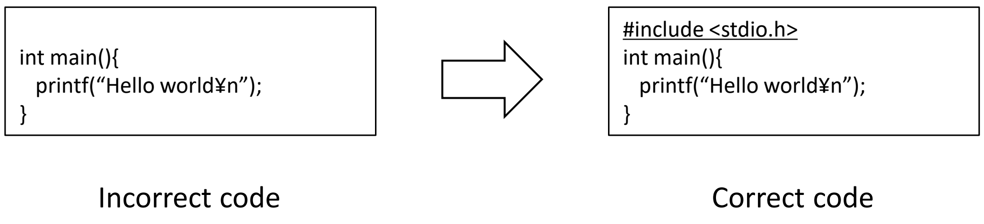
[94] T. Matsumoto, Y. Watanobe, and K. Nakamura, "A model with iterative trials for correcting logic errors in source code," *Applied Sciences*, vol. 11, no. 11, 2021. [Online]. Available: https://www.mdpi.com/2076-3417/11/11/4755

Appendix A

Appendix

A.1 Examples for Classification

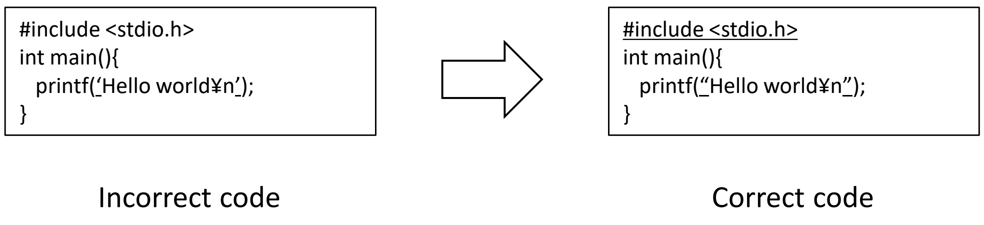
Here is an example that matches the regular expression for classifying logic errors introduced in Table 3.3. Logical errors are classified based on the edit information between incorrect and correct codes. Examples of logical errors extracted by using each regular expression are shown below.



```
int main(){
  printf("Hello world¥n");
}
```

```
#include <stdio.h>
int main(){
  printf("Hello world¥n");
}
```

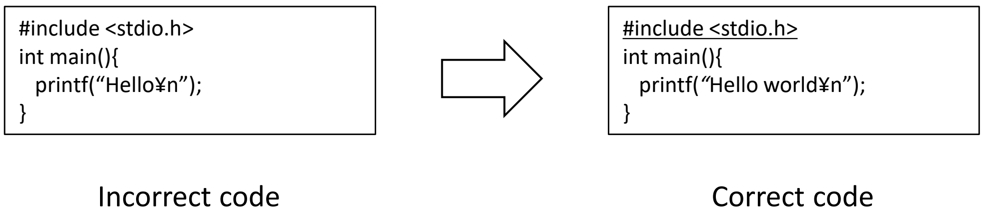
Figure A.1: An example of "include_statement".



```
#include <stdio.h>
int main(){
  printf('Hello world¥n');
}
```

```
#include <stdio.h>
int main(){
  printf("Hello world¥n");
}
```

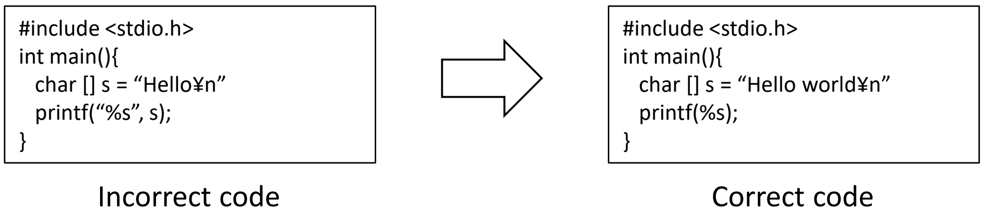
Figure A.2: An example of "switch_quote".



```
#include <stdio.h>
int main(){
  printf("Hello¥n");
}
```

```
#include <stdio.h>
int main(){
  printf("Hello world¥n");
}
```

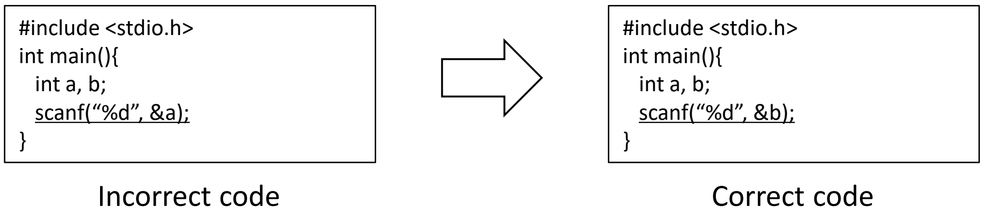
Figure A.3: An example of ”string_format”.



```
#include <stdio.h>
int main(){
  char [] s = "Hello¥n"
  printf("%s", s);
}
```

```
#include <stdio.h>
int main(){
  char [] s = "Hello world¥n"
  printf(%s);
}
```

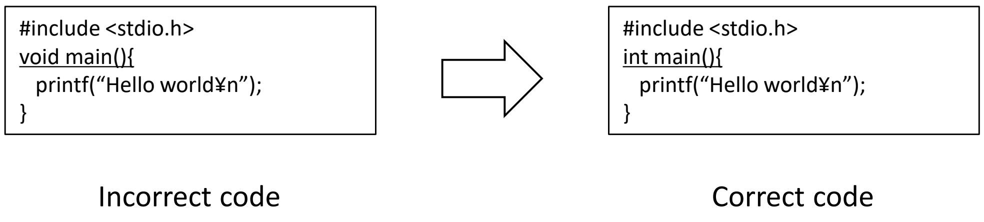
Figure A.4: An example of ”output_format”.



```
#include <stdio.h>
int main(){
  int a, b;
  scanf("%d", &a);
}
```

```
#include <stdio.h>
int main(){
  int a, b;
  scanf("%d", &b);
}
```

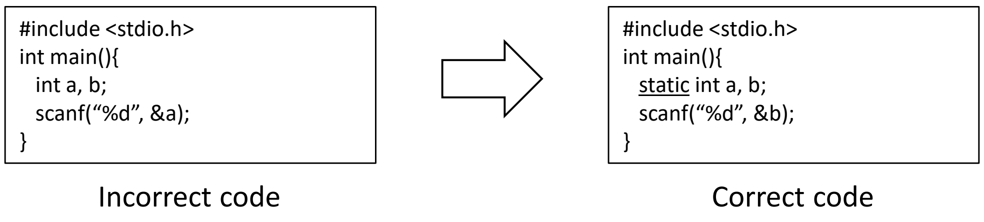
Figure A.5: An example of ”input_statement”.



```
#include <stdio.h>
void main(){
  printf("Hello world¥n");
}
```

```
#include <stdio.h>
int main(){
  printf("Hello world¥n");
}
```

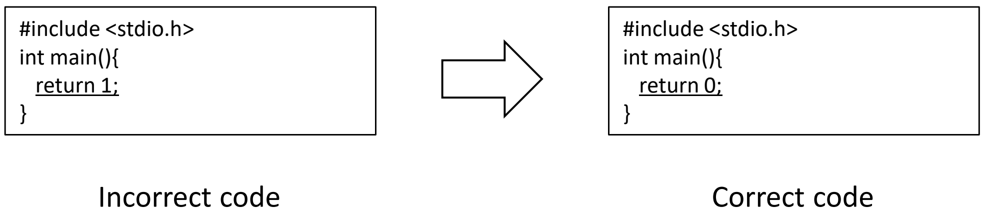
Figure A.6: An example of "void_main_function".



```
#include <stdio.h>
int main(){
  int a, b;
  scanf("%d", &a);
}
```

```
#include <stdio.h>
int main(){
  static int a, b;
  scanf("%d", &b);
}
```

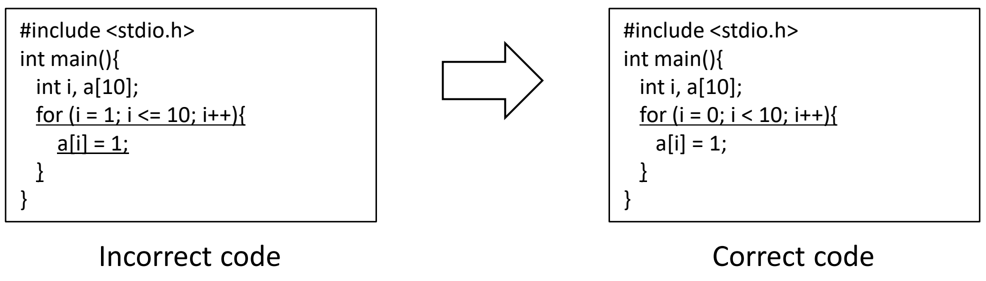
Figure A.7: An example of "static".



```
#include <stdio.h>
int main(){
  return 1;
}
```

```
#include <stdio.h>
int main(){
  return 0;
}
```

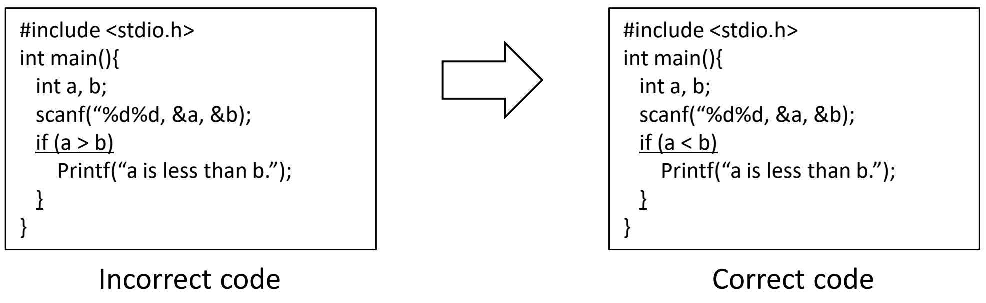
Figure A.8: An example of "return_value".



```
#include <stdio.h>
int main(){
  int i, a[10];
  for (i = 1; i <= 10; i++){
     a[i] = 1;
  }
}
```

```
#include <stdio.h>
int main(){
  int i, a[10];
  for (i = 0; i < 10; i++){
     a[i] = 1;
  }
}
```

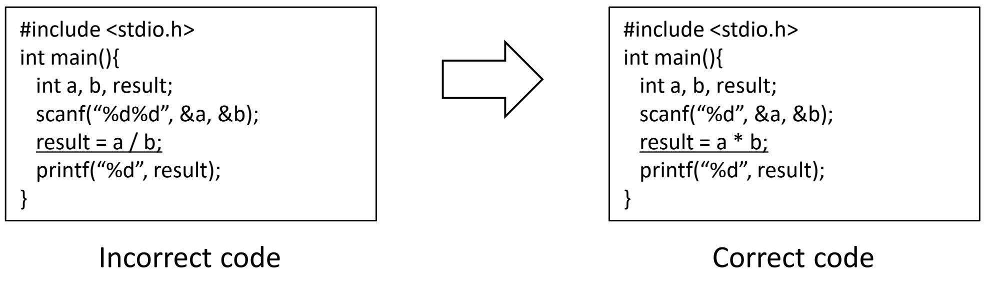
Figure A.9: An example of ”for_statement”.



```
#include <stdio.h>
int main(){
  int a, b;
  scanf(“%d%d, &a, &b);
  if (a > b)
     Printf(“a is less than b.”);
  }
}
```

```
#include <stdio.h>
int main(){
  int a, b;
  scanf(“%d%d, &a, &b);
  if (a < b)
     Printf(“a is less than b.”);
  }
}
```

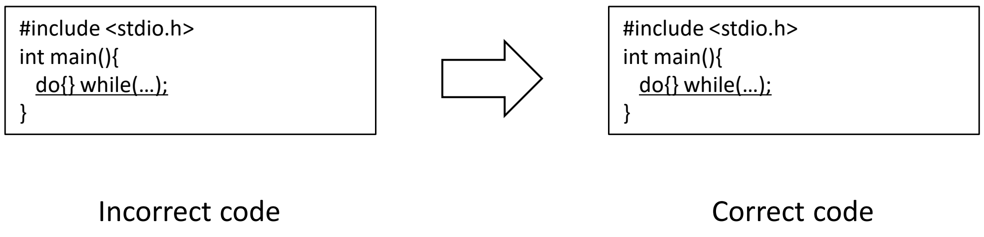
Figure A.10: An example of ”if_statement”.



```
#include <stdio.h>
int main(){
  int a, b, result;
  scanf(“%d%d”, &a, &b);
  result = a / b;
  printf(“%d”, result);
}
```

```
#include <stdio.h>
int main(){
  int a, b, result;
  scanf(“%d”, &a, &b);
  result = a * b;
  printf(“%d”, result);
}
```

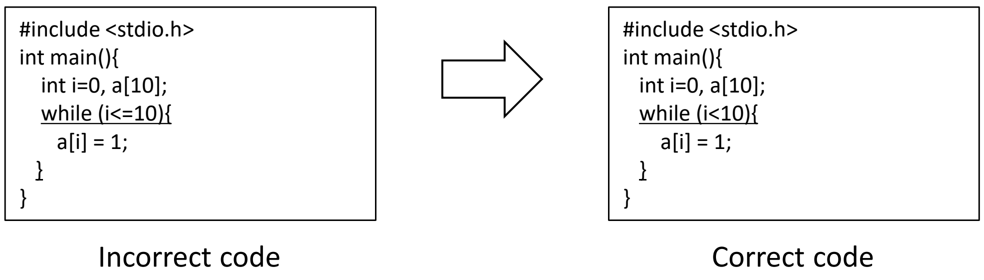
Figure A.11: An example of ”formula”.



```
#include <stdio.h>
int main(){
  do{} while(…);
}
```

```
#include <stdio.h>
int main(){
  do{} while(…);
}
```

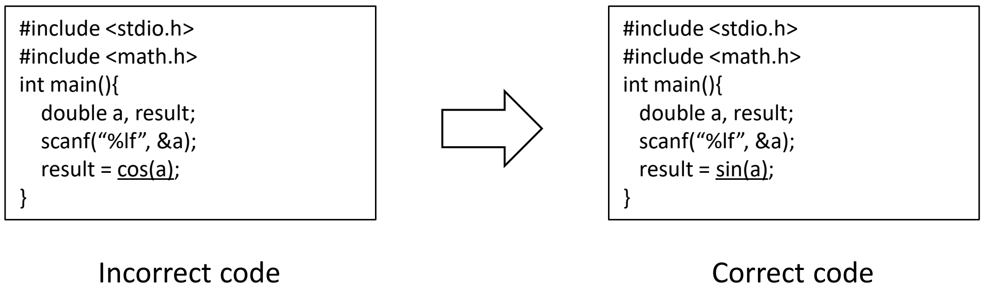
Figure A.12: An example of ”do_while”.



```
#include <stdio.h>
int main(){
   int i=0, a[10];
   while (i<=10){
     a[i] = 1;
   }
}
```

```
#include <stdio.h>
int main(){
  int i=0, a[10];
  while (i<10){
     a[i] = 1;
  }
}
```

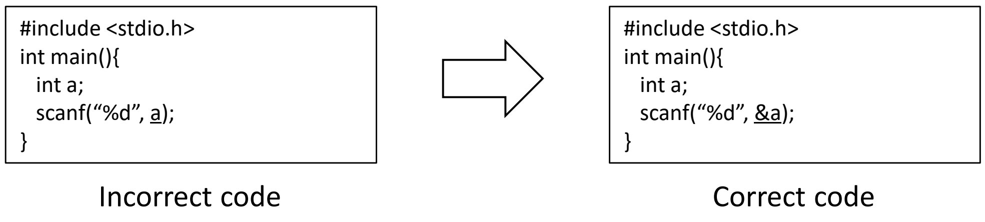
Figure A.13: An example of ”while_statement”.



```
#include <stdio.h>
#include <math.h>
int main(){
   double a, result;
   scanf(“%lf”, &a);
   result = cos(a);
}
```

```
#include <stdio.h>
#include <math.h>
int main(){
  double a, result;
  scanf(“%lf”, &a);
  result = sin(a);
}
```

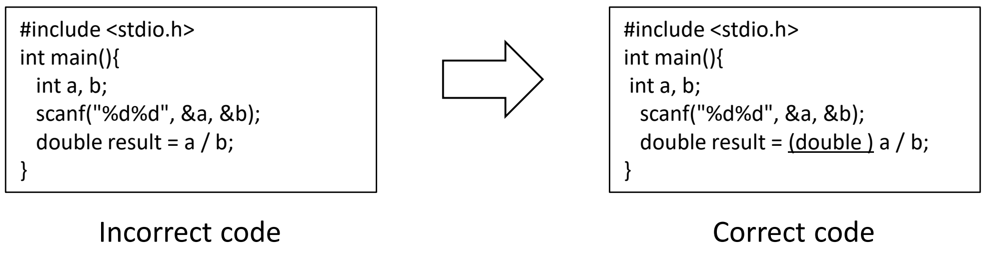
Figure A.14: An example of ”function”.



```
#include <stdio.h>
int main(){
  int a;
  scanf("%d", a);
}
```

```
#include <stdio.h>
int main(){
  int a;
  scanf("%d", &a);
}
```

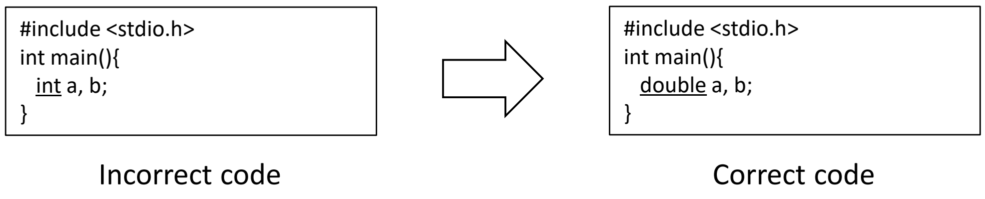
Figure A.15: An example of ”address_operator”.



```
#include <stdio.h>
int main(){
  int a, b;
  scanf("%d%d", &a, &b);
  double result = a / b;
}
```

```
#include <stdio.h>
int main(){
 int a, b;
  scanf("%d%d", &a, &b);
  double result = (double ) a / b;
}
```

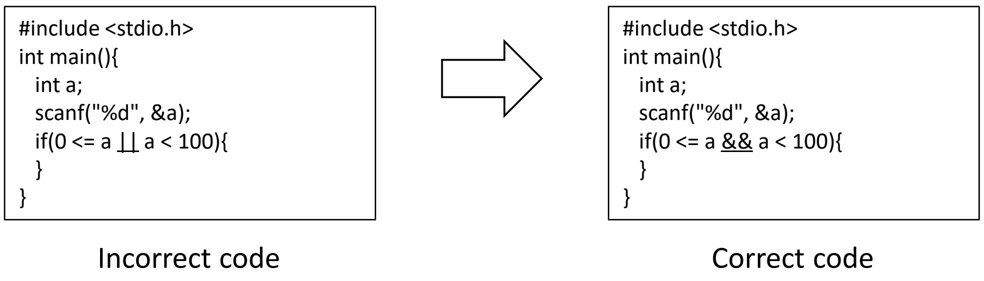
Figure A.16: An example of ”cast”.



```
#include <stdio.h>
int main(){
  int a, b;
}
```

```
#include <stdio.h>
int main(){
  double a, b;
}
```

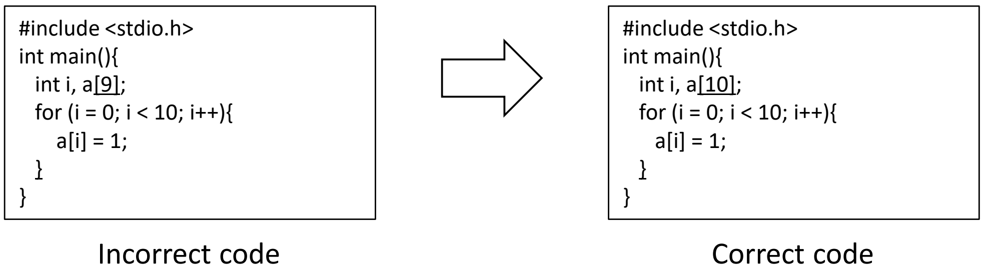
Figure A.17: An example of ”type”.



```
#include <stdio.h>
int main(){
   int a;
   scanf("%d", &a);
   if(0 <= a || a < 100){
   }
}
```

```
#include <stdio.h>
int main(){
   int a;
   scanf("%d", &a);
   if(0 <= a && a < 100){
   }
}
```

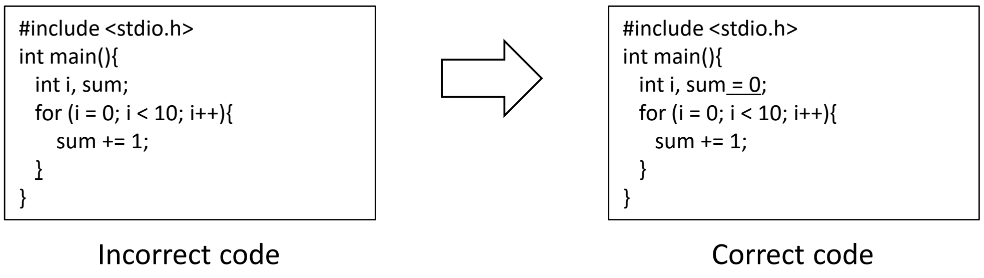
Figure A.18: An example of ”and_or”.



```
#include <stdio.h>
int main(){
   int i, a[9];
   for (i = 0; i < 10; i++){
      a[i] = 1;
   }
}
```

```
#include <stdio.h>
int main(){
   int i, a[10];
   for (i = 0; i < 10; i++){
     a[i] = 1;
   }
}
```

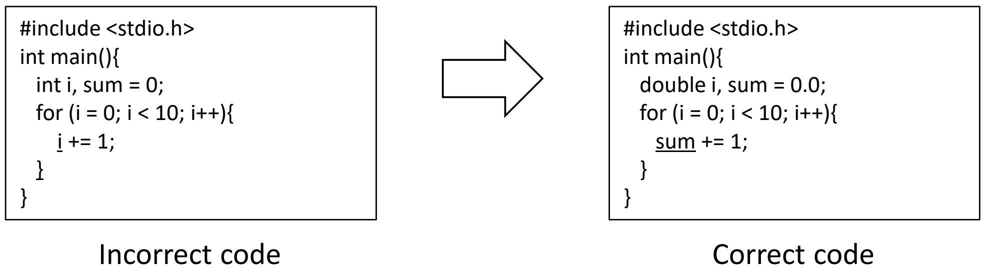
Figure A.19: An example of ”array_index”.



```
#include <stdio.h>
int main(){
   int i, sum;
   for (i = 0; i < 10; i++){
      sum += 1;
   }
}
```

```
#include <stdio.h>
int main(){
   int i, sum = 0;
   for (i = 0; i < 10; i++){
     sum += 1;
   }
}
```

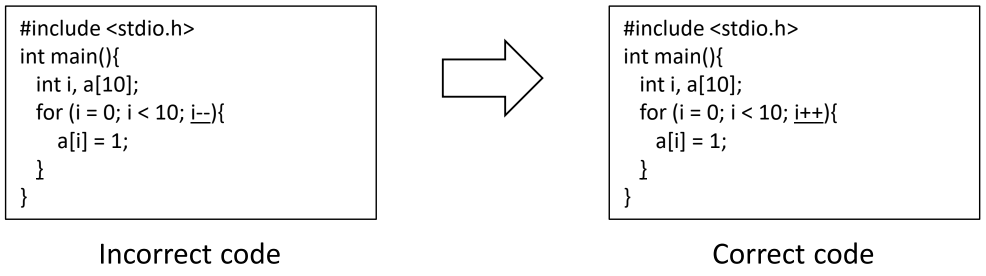
Figure A.20: An example of ”initial_value”.



```
#include <stdio.h>
int main(){
  int i, sum = 0;
  for (i = 0; i < 10; i++){
     i += 1;
  }
}
```

```
#include <stdio.h>
int main(){
  double i, sum = 0.0;
  for (i = 0; i < 10; i++){
    sum += 1;
  }
}
```

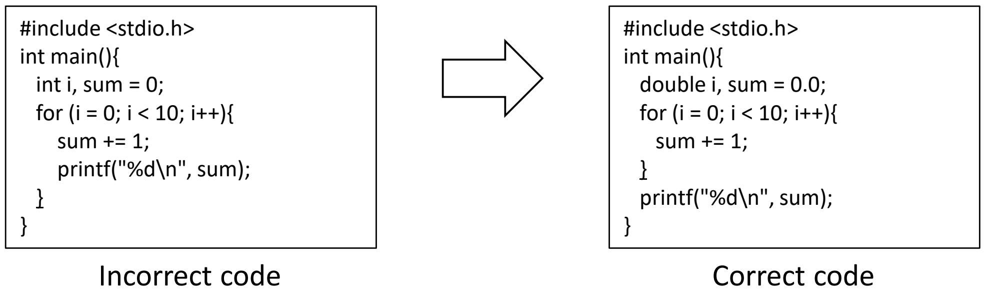
Figure A.21: An example of ”variable_declaration”.



```
#include <stdio.h>
int main(){
  int i, a[10];
  for (i = 0; i < 10; i--){
     a[i] = 1;
  }
}
```

```
#include <stdio.h>
int main(){
  int i, a[10];
  for (i = 0; i < 10; i++){
    a[i] = 1;
  }
}
```

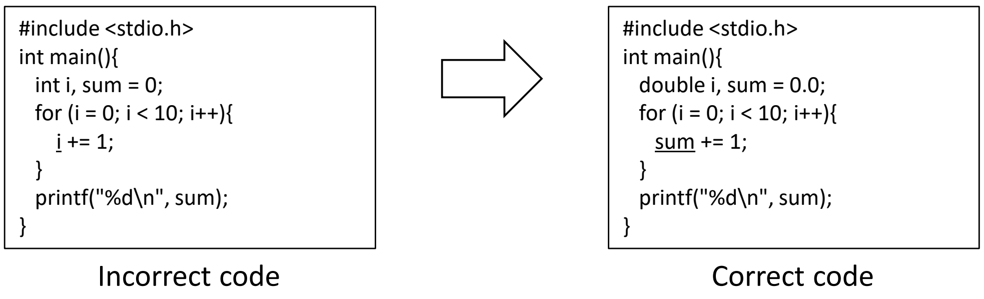
Figure A.22: An example of ”unary_operation”.



```
#include <stdio.h>
int main(){
  int i, sum = 0;
  for (i = 0; i < 10; i++){
     sum += 1;
     printf("%d\n", sum);
  }
}
```

```
#include <stdio.h>
int main(){
  double i, sum = 0.0;
  for (i = 0; i < 10; i++){
    sum += 1;
  }
  printf("%d\n", sum);
}
```

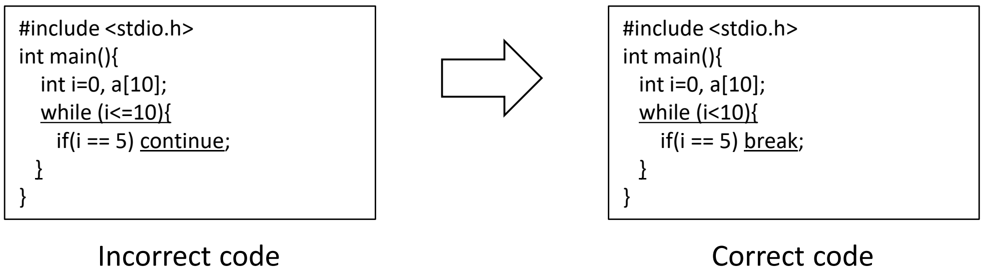
Figure A.23: An example of ”scope”.



```
#include <stdio.h>
int main(){
  int i, sum = 0;
  for (i = 0; i < 10; i++){
     i += 1;
  }
  printf("%d\n", sum);
}
```

```
#include <stdio.h>
int main(){
  double i, sum = 0.0;
  for (i = 0; i < 10; i++){
    sum += 1;
  }
  printf("%d\n", sum);
}
```

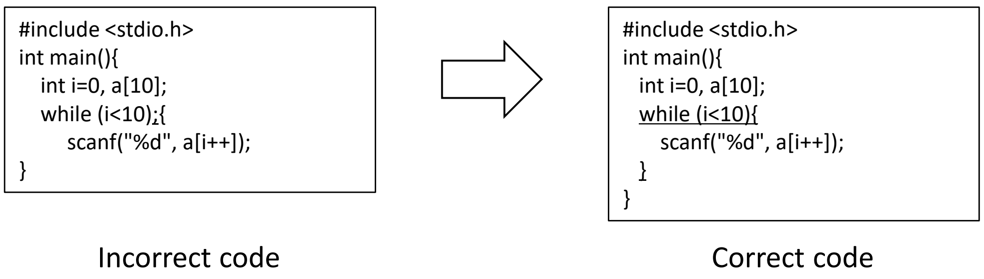
Figure A.24: An example of ”substituted_variable”.



```
#include <stdio.h>
int main(){
   int i=0, a[10];
   while (i<=10){
     if(i == 5) continue;
  }
}
```

```
#include <stdio.h>
int main(){
  int i=0, a[10];
  while (i<10){
     if(i == 5) break;
  }
}
```

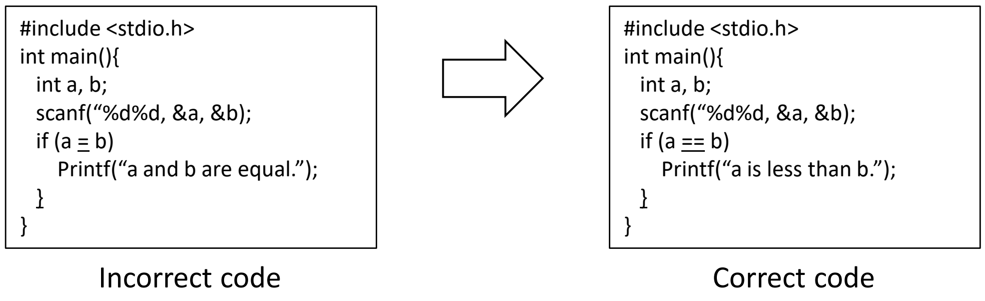
Figure A.25: An example of ”switch_continue_break”.



```
#include <stdio.h>
int main(){
   int i=0, a[10];
   while (i<10);{
      scanf("%d", a[i++]);
}
```

```
#include <stdio.h>
int main(){
  int i=0, a[10];
  while (i<10){
     scanf("%d", a[i++]);
  }
}
```

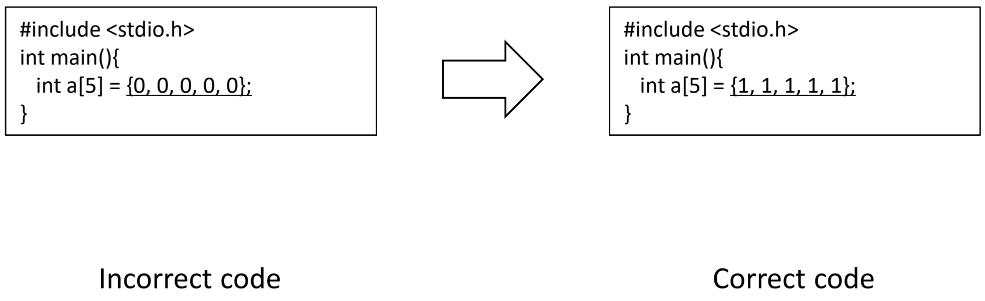
Figure A.26: An example of ”semicolon_position”.



```
#include <stdio.h>
int main(){
  int a, b;
  scanf("%d%d, &a, &b);
  if (a = b)
     Printf("a and b are equal.");
  }
}
```

```
#include <stdio.h>
int main(){
  int a, b;
  scanf("%d%d, &a, &b);
  if (a == b)
     Printf("a is less than b.");
  }
}
```

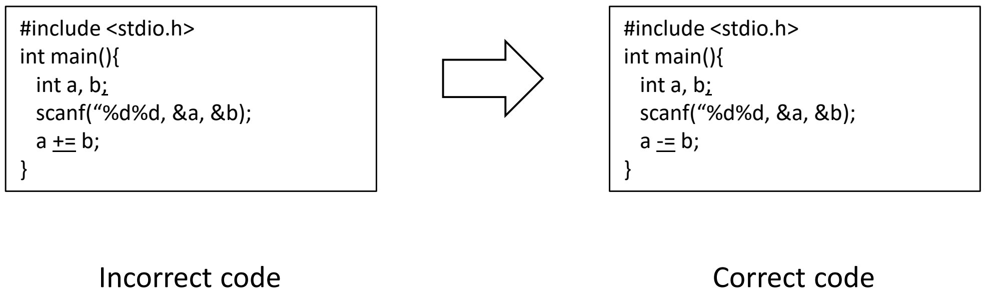
Figure A.27: An example of "switch_equal_assign".



```
#include <stdio.h>
int main(){
  int a[5] = {0, 0, 0, 0, 0};
}
```

```
#include <stdio.h>
int main(){
  int a[5] = {1, 1, 1, 1, 1};
}
```

Figure A.28: An example of "array_elements".



```
#include <stdio.h>
int main(){
  int a, b;
  scanf("%d%d, &a, &b);
  a += b;
}
```

```
#include <stdio.h>
int main(){
  int a, b;
  scanf("%d%d, &a, &b);
  a -= b;
}
```

Figure A.29: An example of "assignment_operator".

A.2 Experimental Results for Each Programming Task in Chapter 4

Here, the results of experiments other than the programming task shown in Figure 5.2 in Chapter 4 are presented.

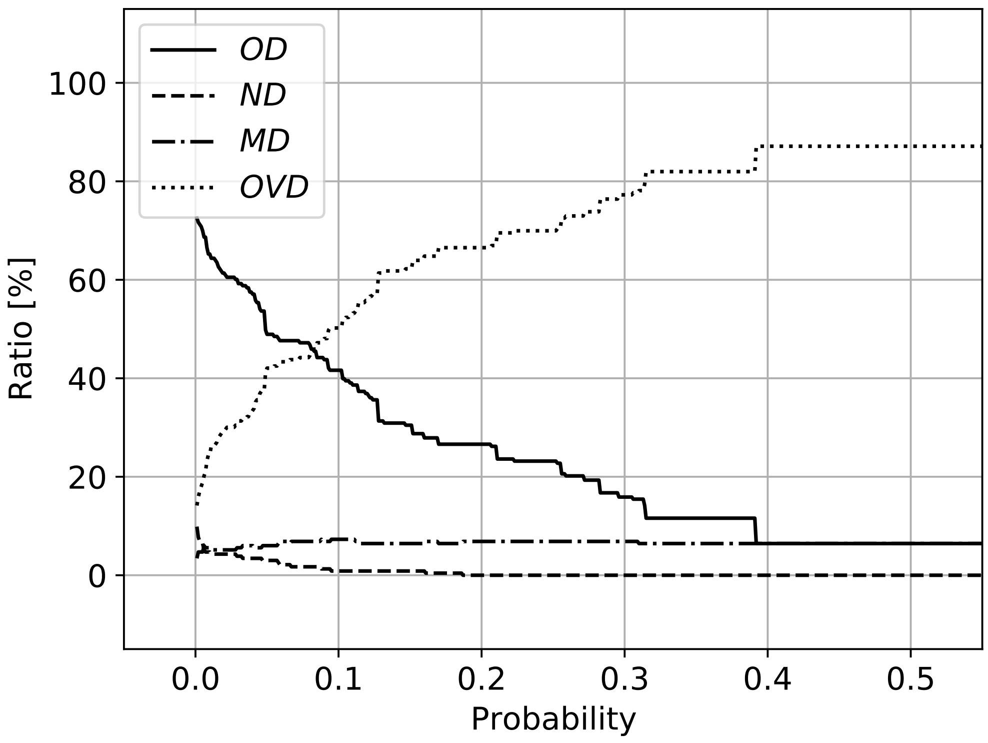


Figure A.30: A result of ITP1_2_A.

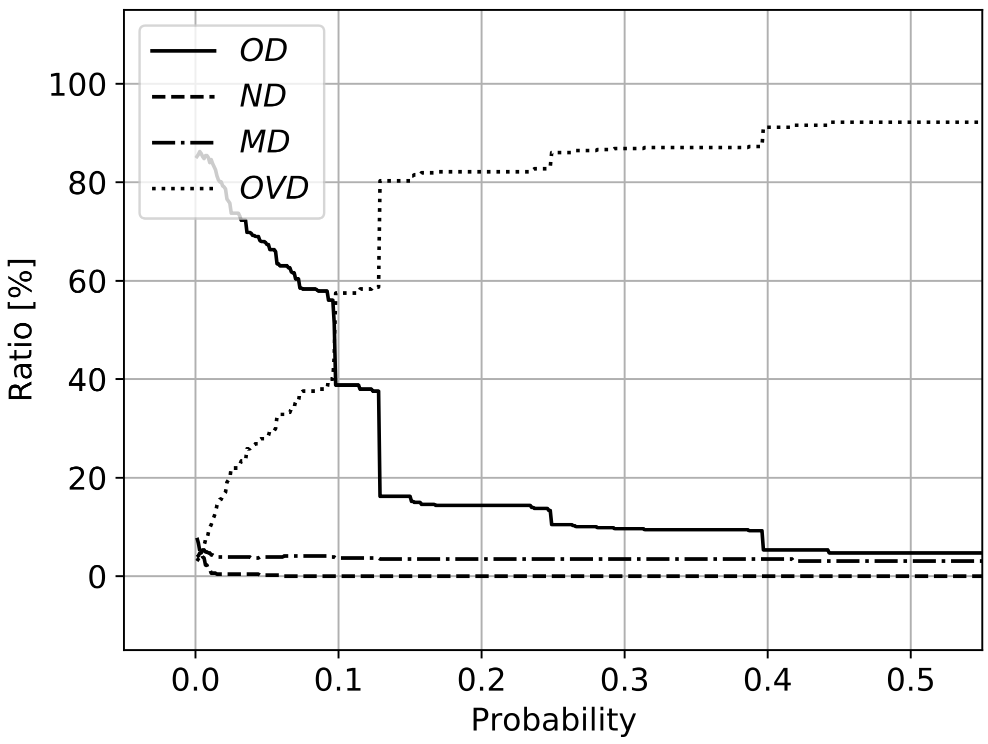


Figure A.31: A result of ITP1_2_B.

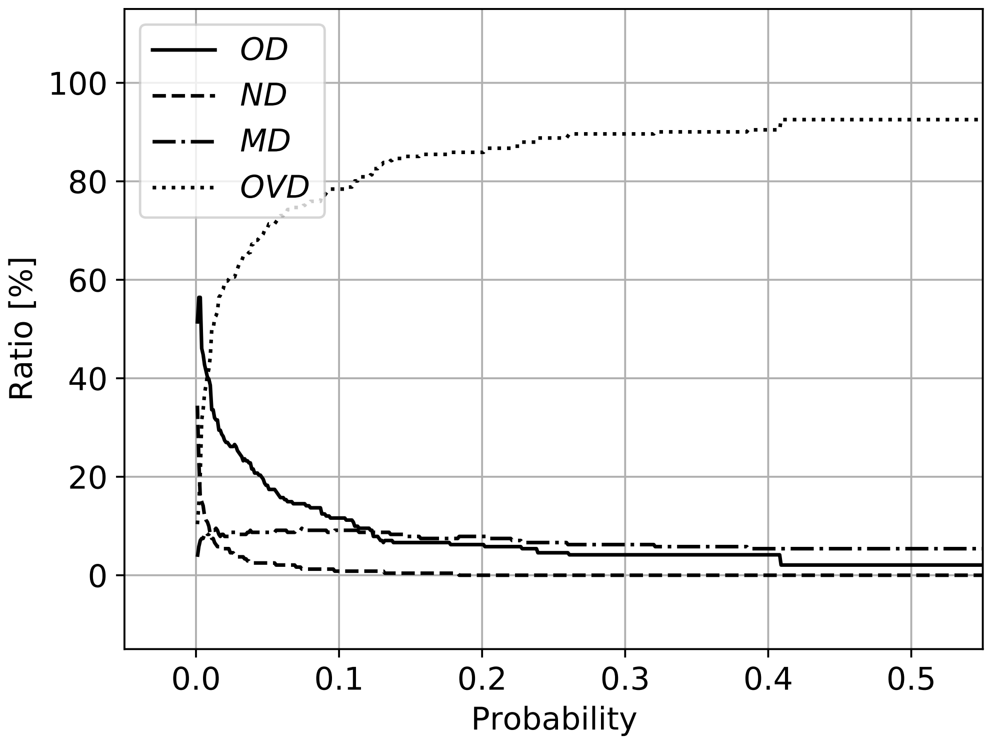


Figure A.32: A result of ITP1_2_C.

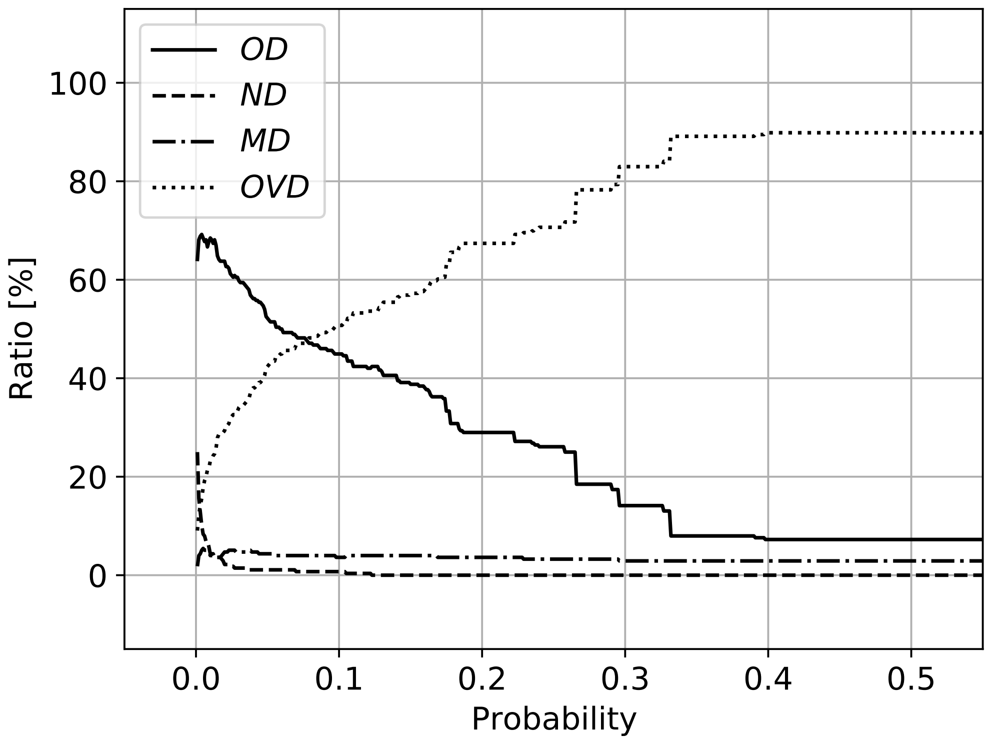


Figure A.33: A result of ITP1_2_D.

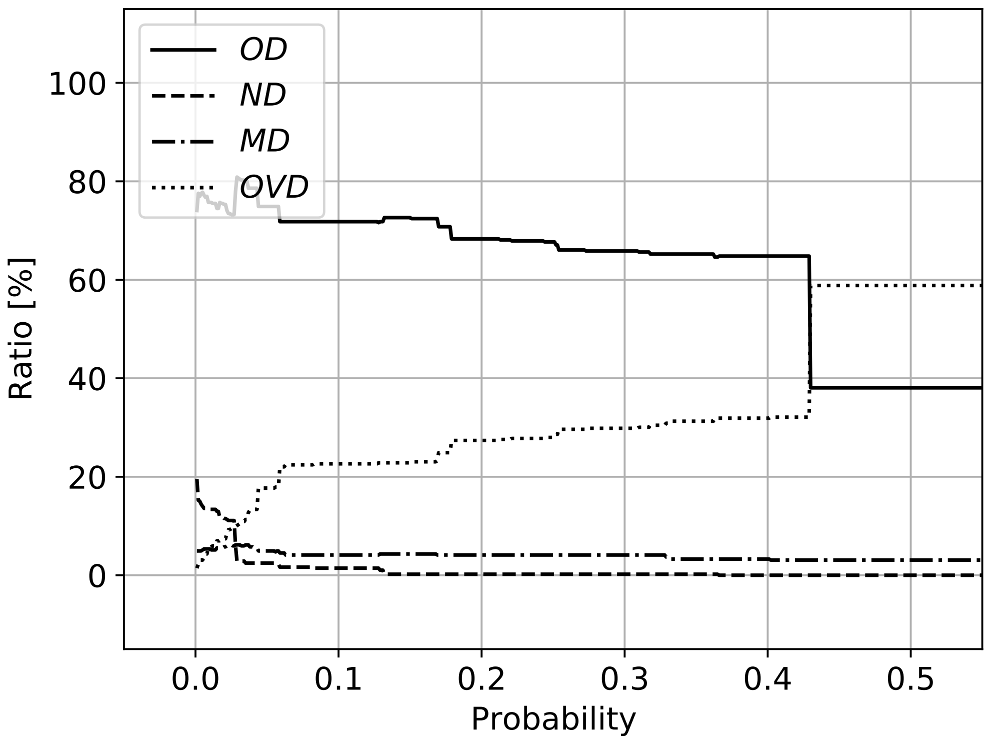


Figure A.34: A result of ITP1_3_A.

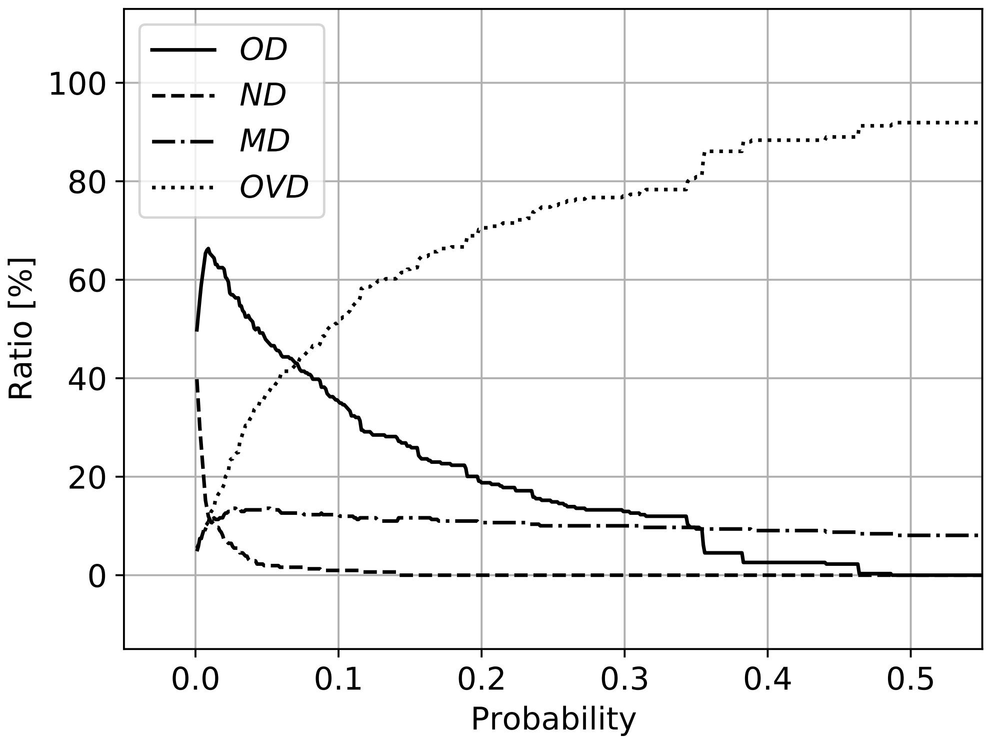


Figure A.35: A result of ITP1_3_B.

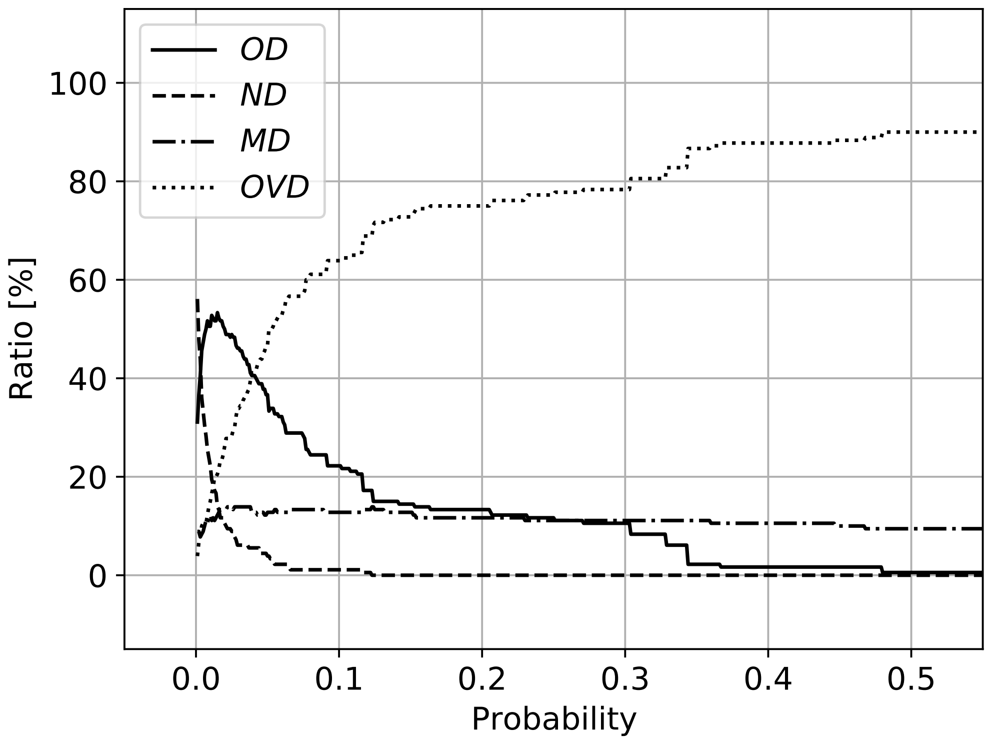


Figure A.36: A result of ITP1_3_C.

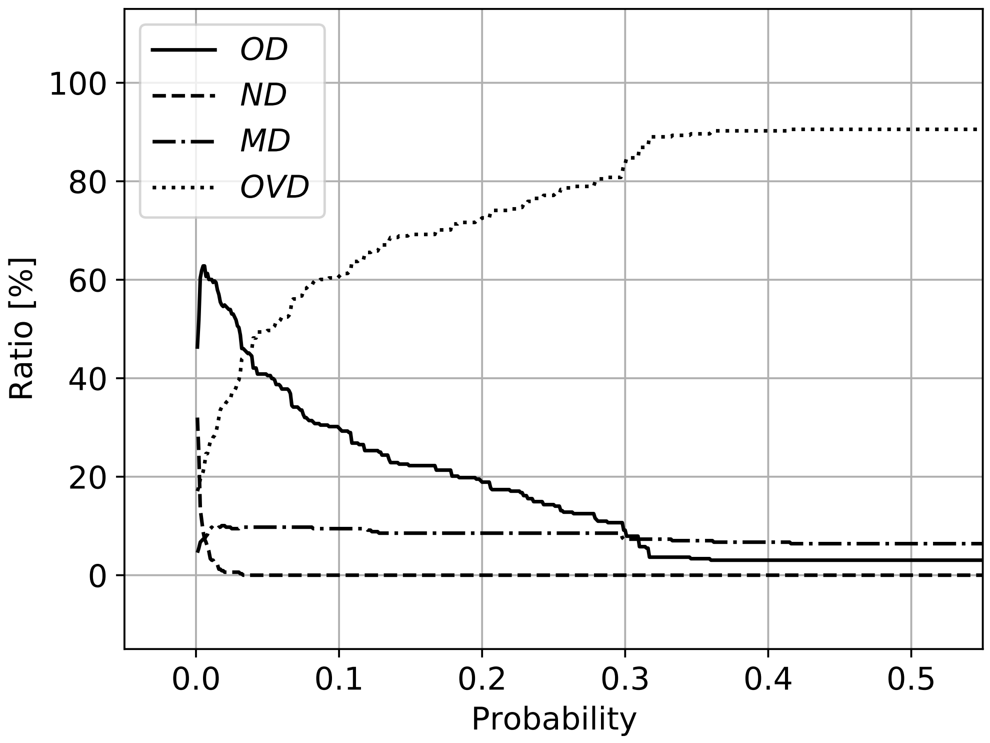


Figure A.37: A result of ITP1_3_D.

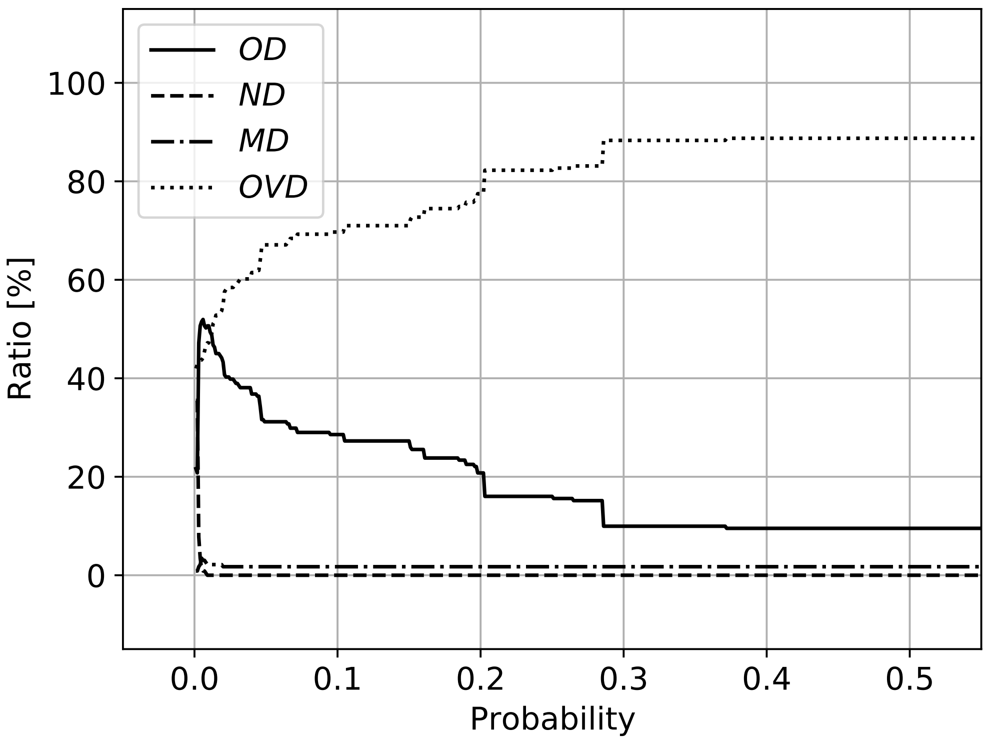


Figure A.38: A result of ITP1_4_A.

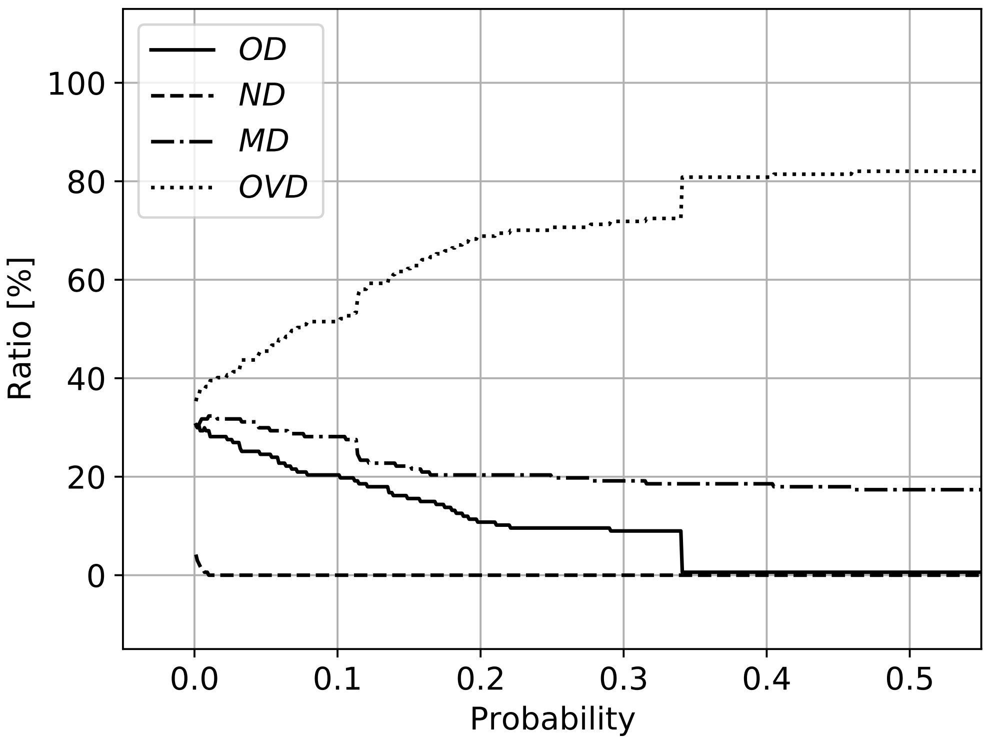


Figure A.39: A result of ITP1_4_B.

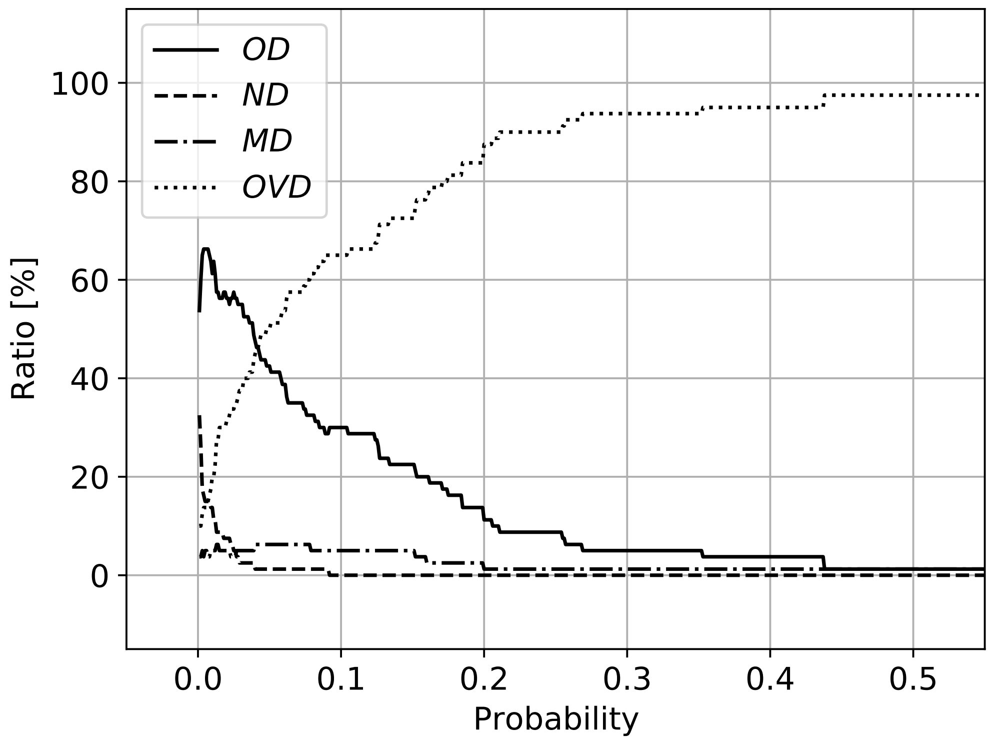


Figure A.40: A result of ITP1_4_C.

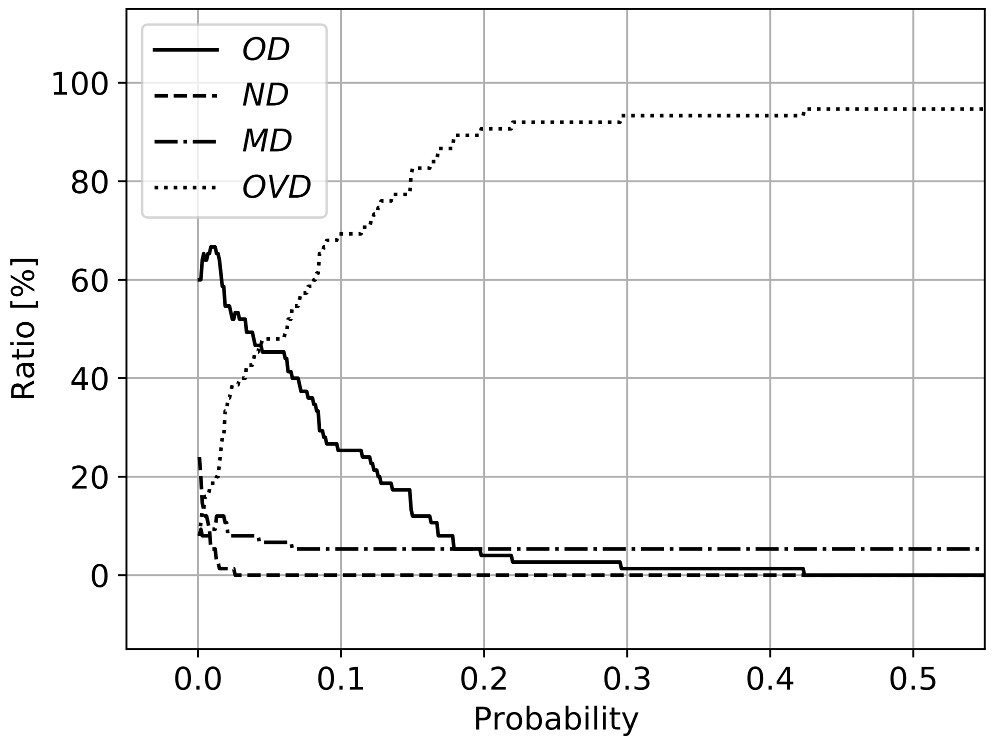


Figure A.41: A result of ITP1_4_D.

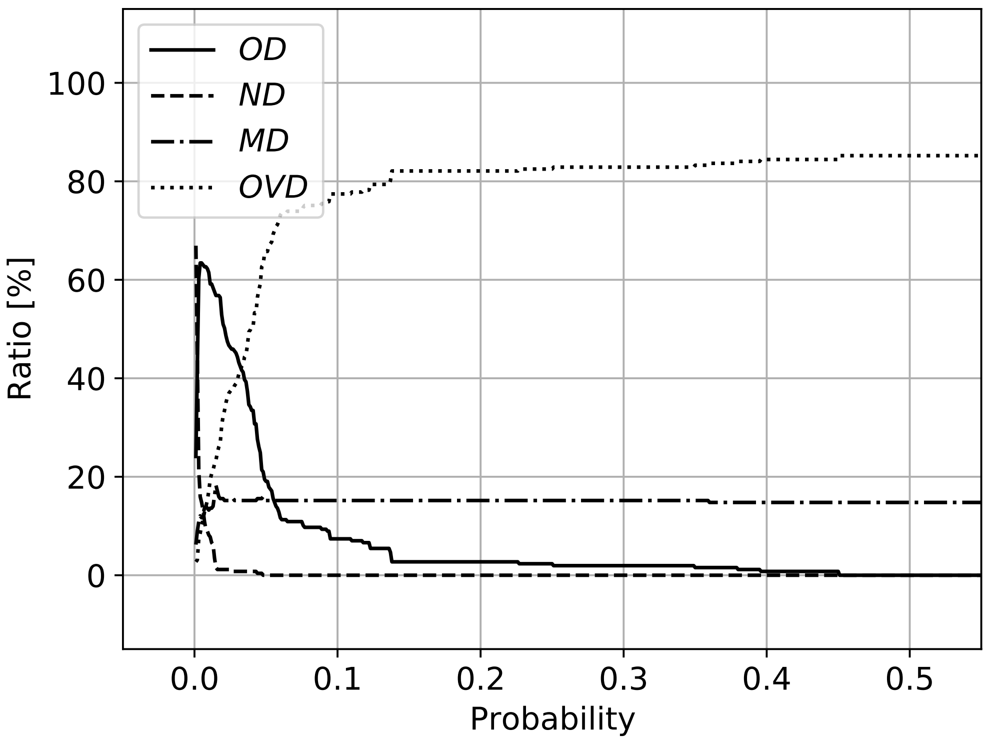


Figure A.42: A result of ITP1_5_A.

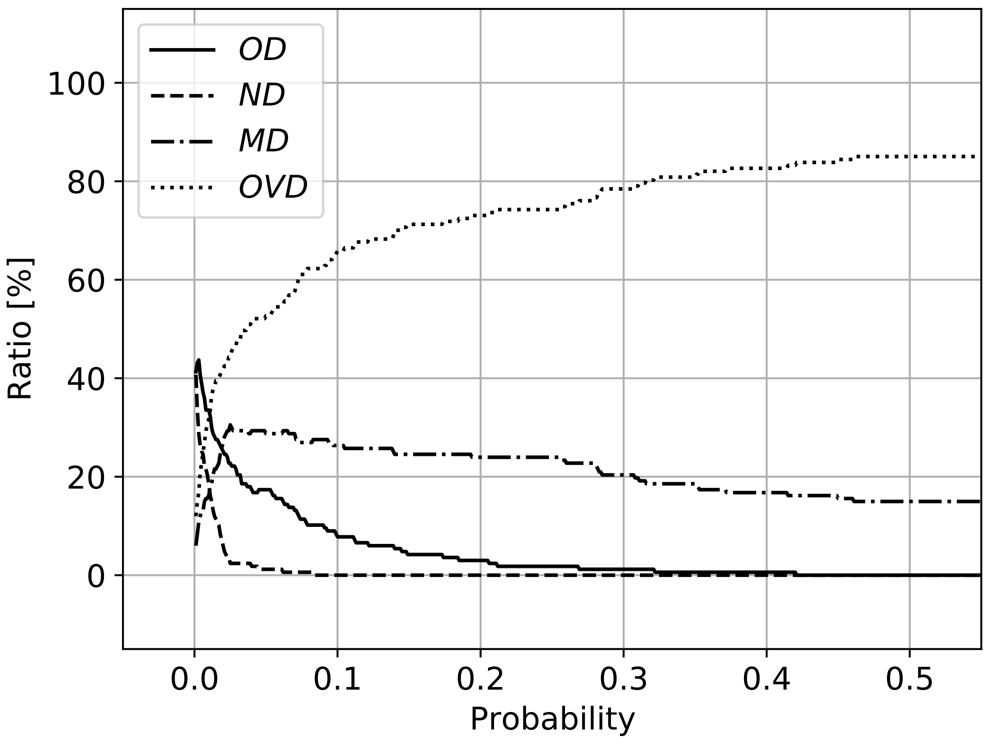


Figure A.43: A result of ITP1_5_B.

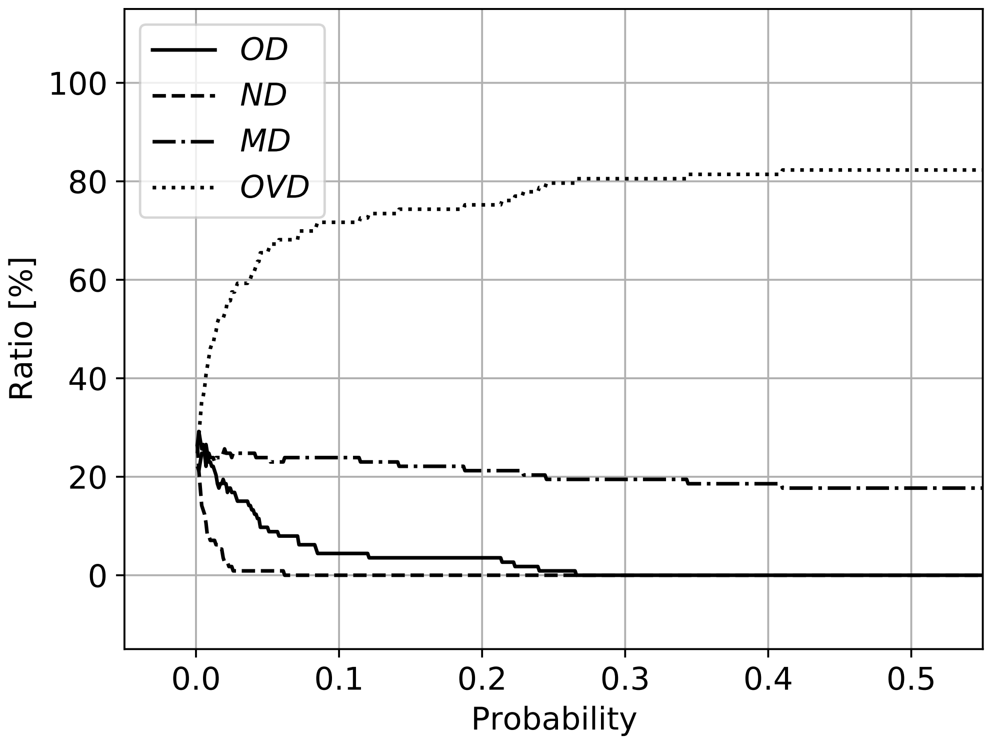


Figure A.44: A result of ITP1_5_C.

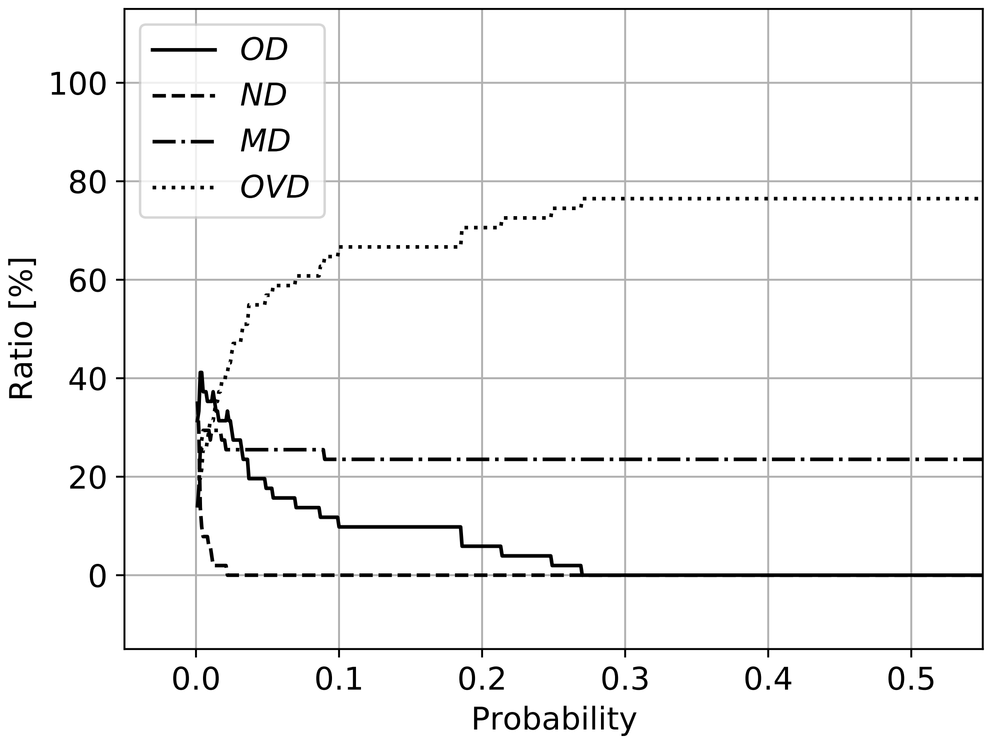


Figure A.45: A result of ITP1_5_D.

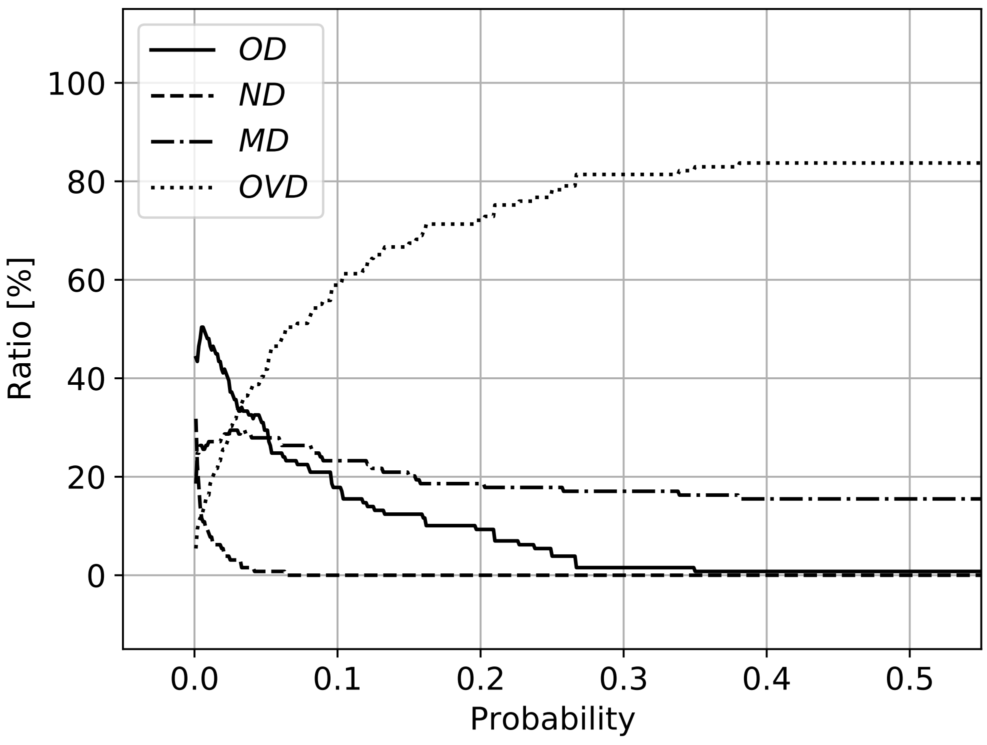


Figure A.46: A result of ITP1_6_A.

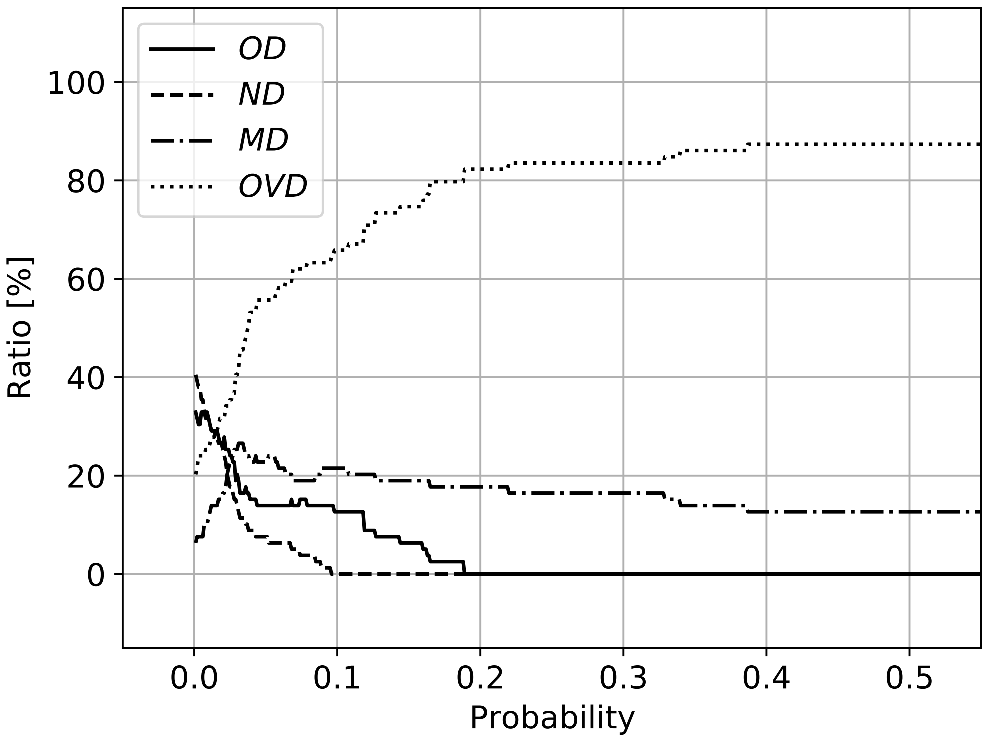


Figure A.47: A result of ITP1_6_B.

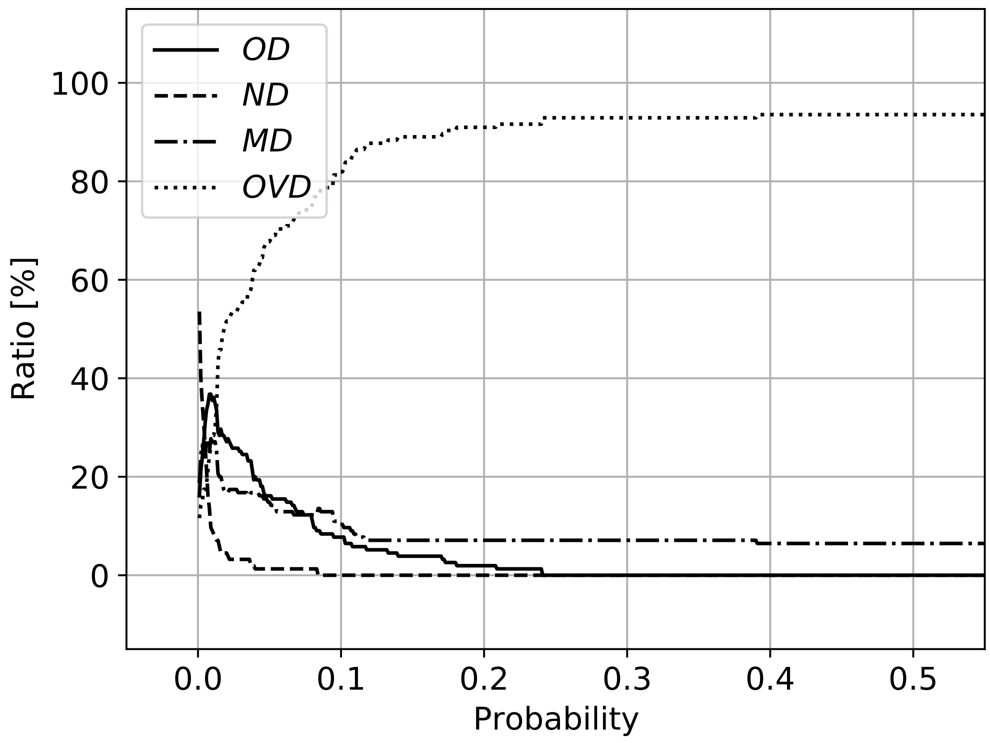


Figure A.48: A result of ITP1_6_C.

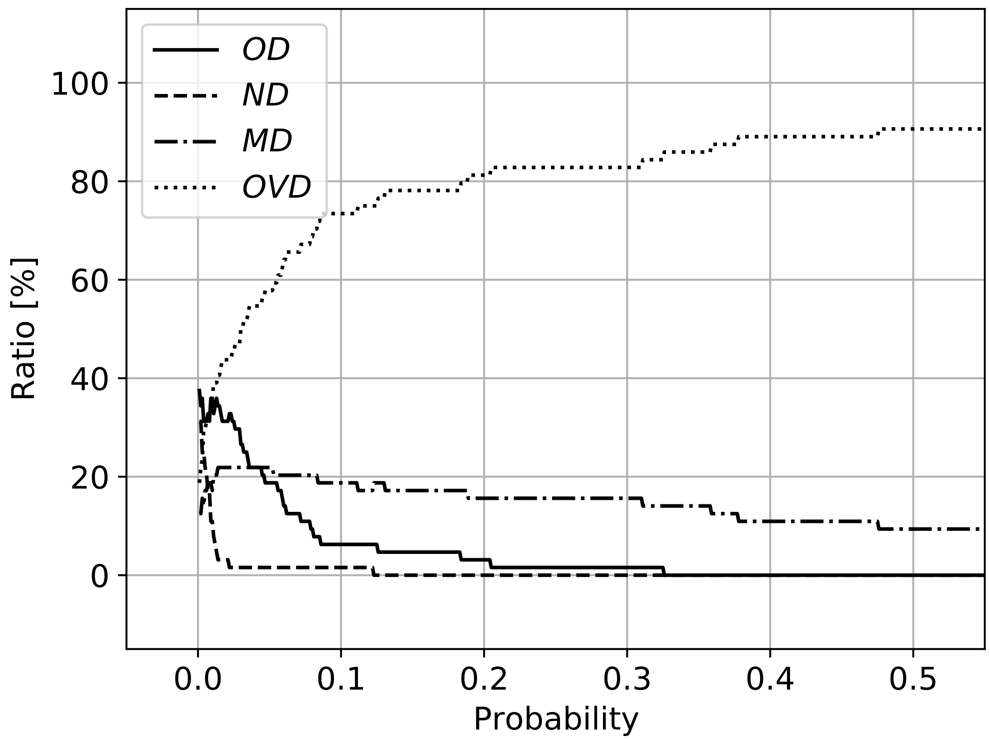


Figure A.49: A result of ITP1_6_D.

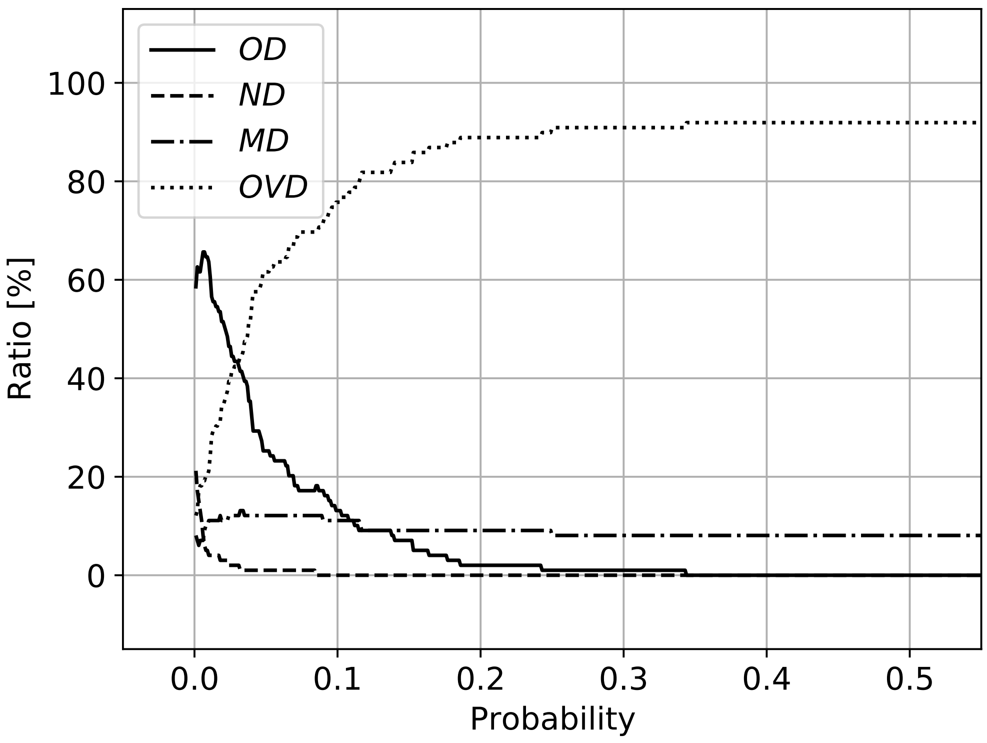


Figure A.50: A result of ITP1_7_A.

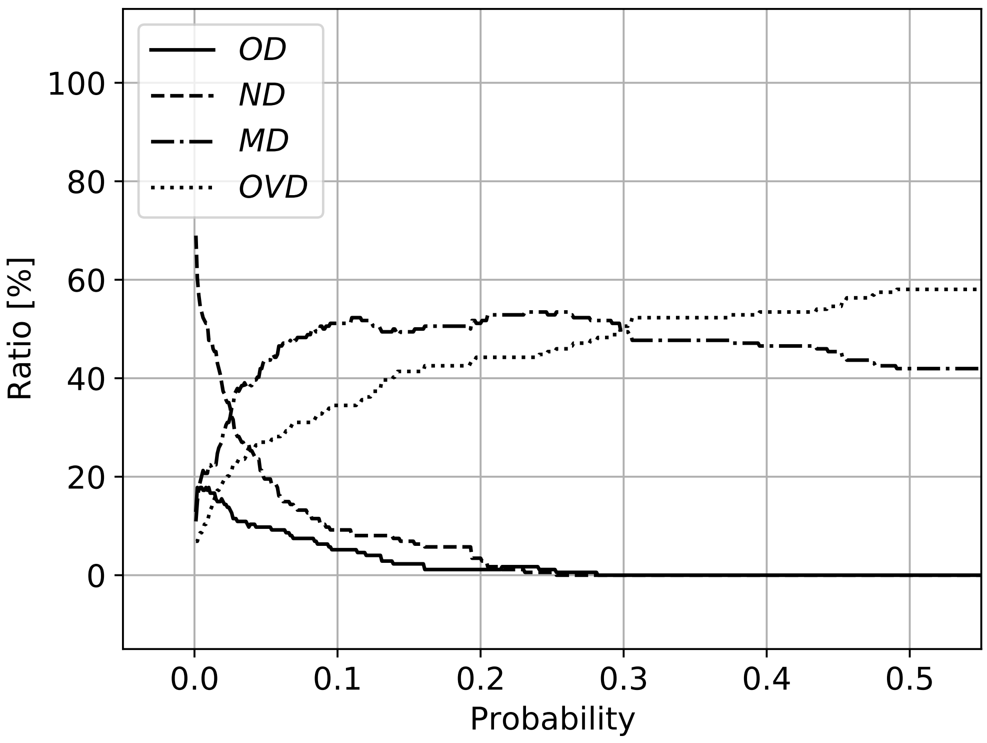


Figure A.51: A result of ITP1_7_B.

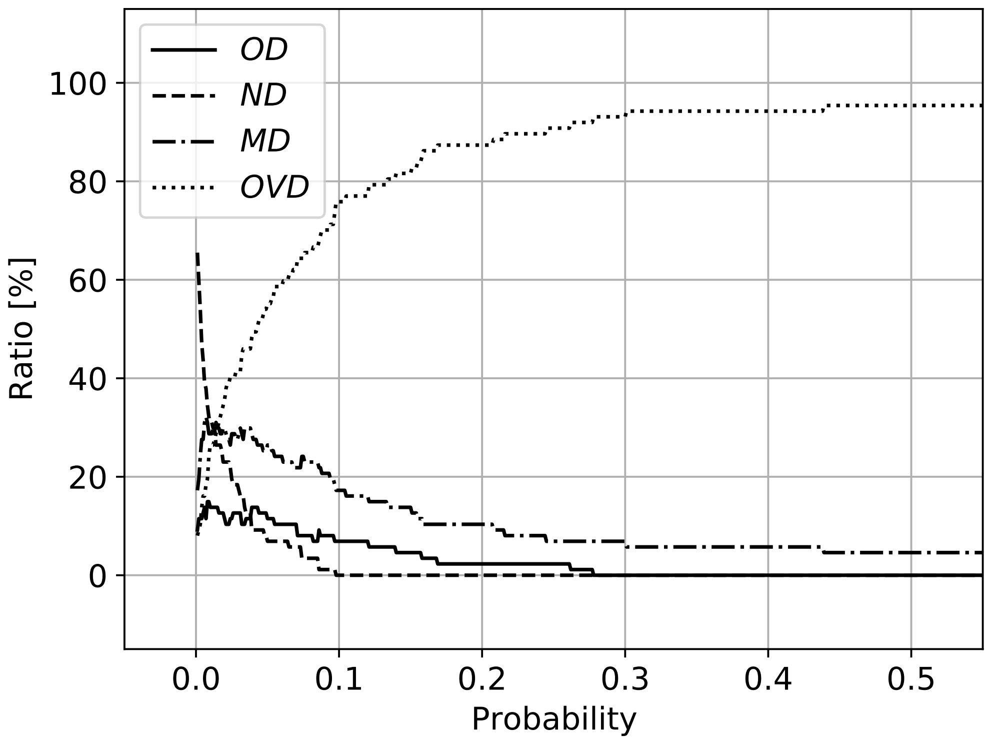


Figure A.52: A result of ITP1_7_C.

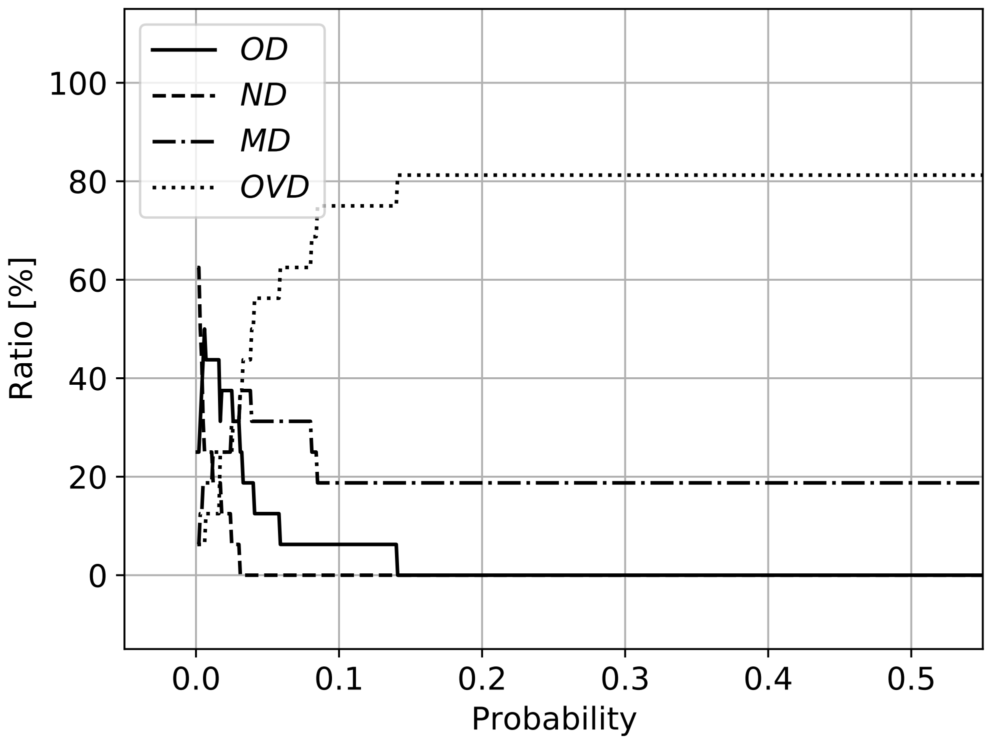


Figure A.53: A result of ITP1_7_D.

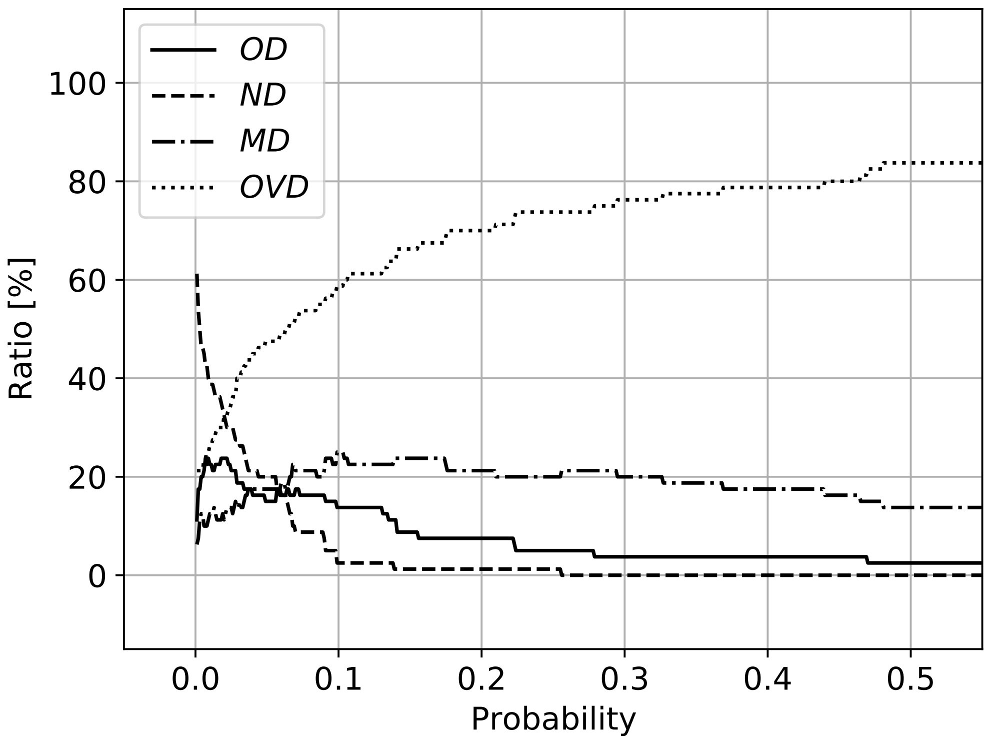


Figure A.54: A result of ITP1_8_A.

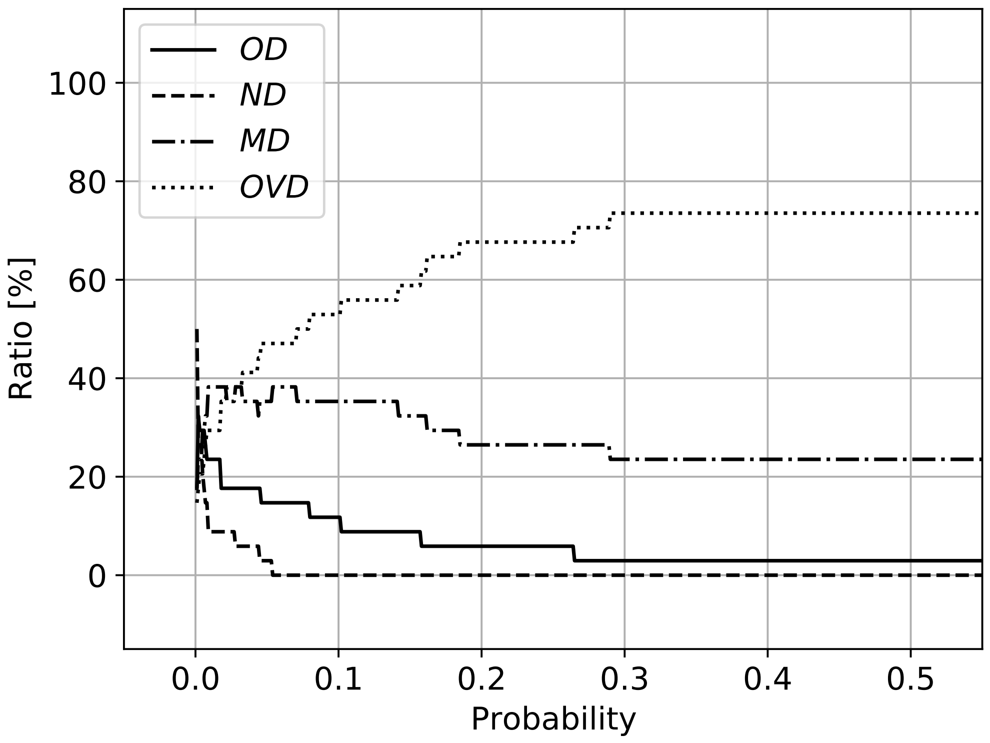


Figure A.55: A result of ITP1_8_B.

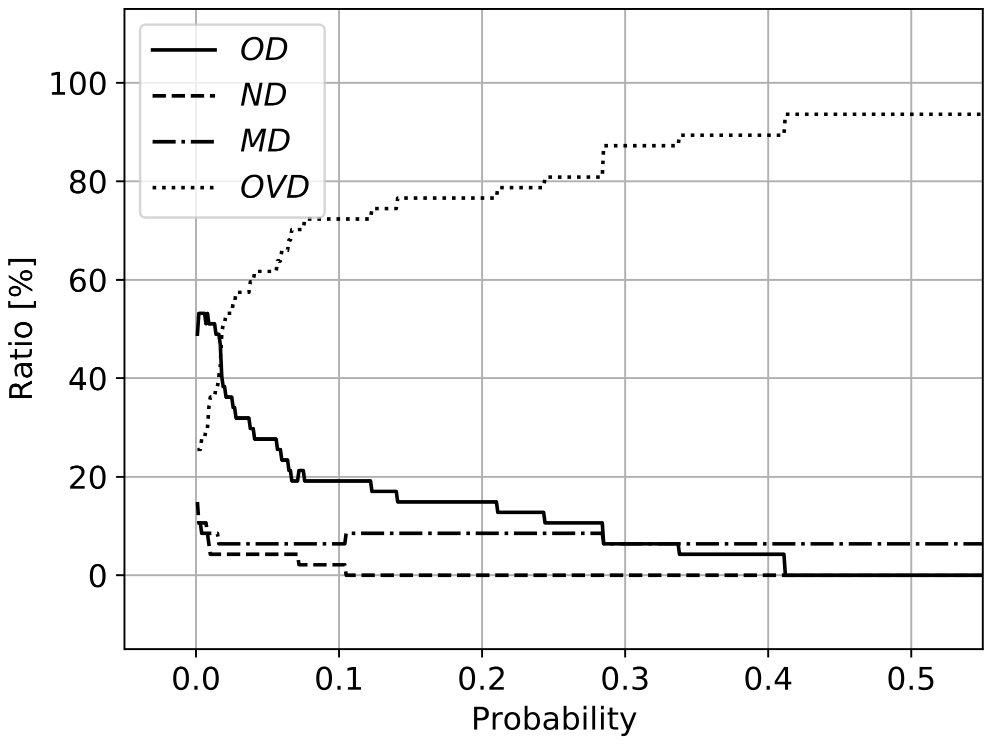


Figure A.56: A result of ITP1_8_C.

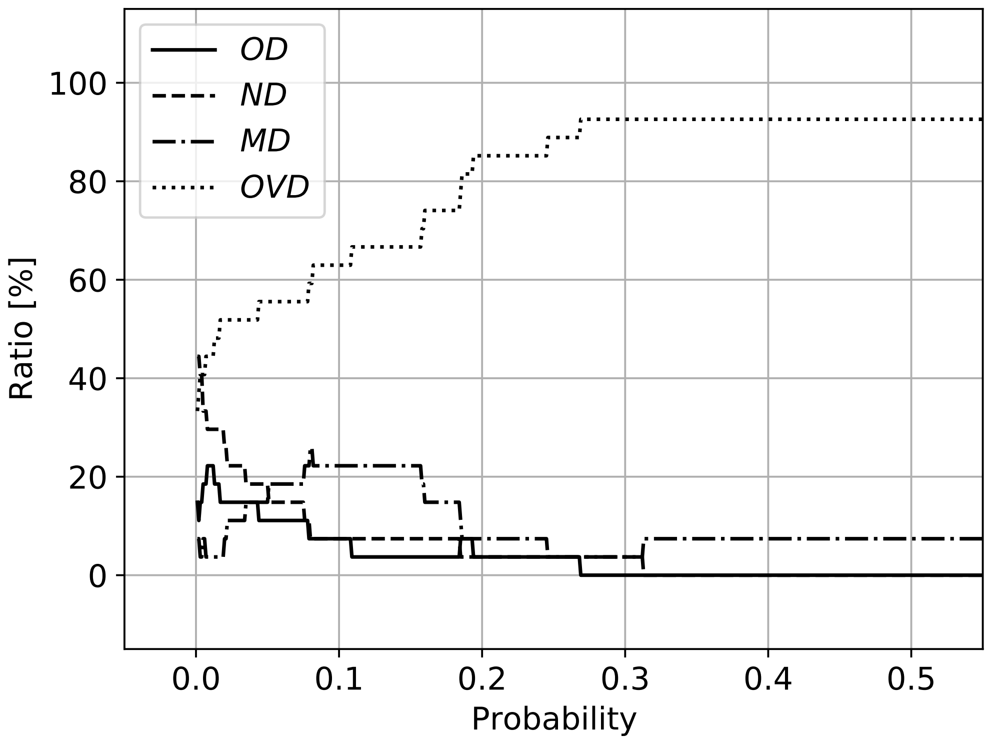


Figure A.57: A result of ITP1_8_D.