**Master Thesis**

Title

# A merge conflict resolution method using source code metrics and development history data

Supervisor

Katsuro Inoue

Author

Mohan Bian

2022/2/2

Software Engineering Laboratory, Department of Computer Science

Graduate School of Information Science and Technology, Osaka University

令和三年度 Master Thesis

# A merge conflict resolution method using source code metrics and development history data

Mohan Bian

## Abstract

In large-scale software development, a version control system is frequently used. However, if multiple persons change the same piece of code in parallel, conflicts may occur. In order to merge the changes successfully, the developer must investigate the cause, re-edit the code. This can take hours or even days, delaying the project ’ s development schedule while the developer repeatedly reviews to identify the reason for the conflict and find a solution.

In the previous research, a machine learning model was created to determine how to solve merge conflicts from meta information such as the number of lines of merge conflicts, the date and time when commits were created, and the developers who created them. In this research, by adding source code metrics to the model, we aim to investigate the influence on the judging model that suggests how to resolve appropriate merge and which source code metrics contribute more to the model. Also, we examine the adaptability of the model for a different language. 20 Java projects and 7 Python projects were used from the OSS projects published by the Apache Software Foundation. In the case of Java, the average changed from 78.56% to 77.88%. In python, the average increased from 64.79% to 70.28%

**Keywords**

Merge conflict
Machine learning
Source code metrics

## Contents

## 1 Introduction

In large-scale software development projects, it is common for multiple developers to work together. For multi-person development, it is necessary to record information such as “when”, “who”, and “what changes were made” for trouble shooting afterwards. To record these information, a version control system was frequently used. While projects can be efficiently developed by multiple people, parallel development may cause problems. After completing a newly created branch from the mainstream, if the mainstream also edits it, there will be a situation where it cannot be merged when merging it into the mainstream. This is called a merge conflict. Merge conflict is one of the most annoying problems. It has been clarified that it occurs relatively frequently in software development using a version control system. In the research by Brun et al., as a result of investigating the development history of nine open-source software (hereinafter, OSS) , merge conflicts occurred in all projects. It has been shown that the ratio of merge conflicts occurs to all merges is about 19% on average and the maximum is about 42% [1].

The problem with merge conflict is that it takes time and effort to resolve them. If a merge conflict occurs, the developer must investigate the cause, re-edit the code, and debug until the merge is successful. This can take hours or even days, delaying the project to the development schedule while the developer repeatedly reviews to identify the reason for the conflict and find a solution [2].

In some existing studies, the characteristics of merge conflicts and their resolution methods have become clear. It has been clarified that the higher the number of lines where merge conflicts occur, the higher the rate of merge conflicts would be [3]. In Shiraki’s research [4], a model was created using machine learning to determine how to resolve merge conflicts from meta information such as the number of lines of merge conflicts, the date and time when commits were created, and the developers who created them. It became clear that the number of lines of the merge conflict contributes to the method of resolving the merge conflict.

Ahmed et al. have also shown that bugs lead to merge conflicts [5]. Bad code design not only impacts maintainability, it also impacts the day-to-day operations of a project, such as merging contributions. Ahmed et al. indicate that research is needed to identify better ways to support merge conflict resolution to minimize its effect on code quality. In other research, it became clear that the complexity of the program calculated by the source code metrics is also related to the defects of the program [6]. From the above, we realized it is possible that source code metrics are potential indicators of merge conflict. By adding source code metrics to the model, we aim to create a model that suggests how to resolve appropriate merge conflicts with higher accuracy.

In this paper, Chapter 2 describes the background of the research, Chapter 3 introduces the previous research, Chapter 4 proposes methods for model extension and parameter improvement, and Chapter 5 evaluates the improved model. Chapter 6 discusses the limitations of this study. Finally, we summarize the research in Chapter 7.

# 2 Background

## 2.1 Version Control System

For multi-person development, it is necessary to record each version of project as developing history for trouble shooting afterwards. In that case, a version control system is frequently used. There are numerous version control system products, of which Apache SubVersion (commonly known as SVN) and Git are used in many projects.

SVN has spread rapidly because it is free of charge, it is compatible with various operating systems, and it is provided as one of the functions of programming software. However, since there is only one repository on the server, it is not suitable for large-scale development projects because it is difficult for individuals to manage assets under development, and it is not possible to manage versions while offline. So, you can't make a commit until all the work is completely done. Therefore, Git was released to support large-scale development projects, and the biggest feature is that in addition to the remote repository on the server shared by multiple people, it has a copy of the remote repository as a local repository on the client's personal computer [6]. This has made it possible for individuals to manage assets under development in a local repository even when offline. Then, in the case of parallel development using Git, a new derivative branch is created from the branch that is the mainstream of development, and after the deliverable is completed, it is integrated (merged) into the mainstream branch. As a result, multiple developers can proceed with development at the same time, and efficient software development can be realized.

Therefore, Git has gradually become more popular than SVN because it has the feature of being able to develop using branches, has abundant functions, and has a sophisticated operating system. Thanks to Git, multiple people's development can be easily realized. On the other hand, using Git may cause problems. Merge conflict is a classical problem with a long history while using VCS for parallel development.

In this paper, we aim to improve a model that suggests how to resolve appropriate merge conflicts by adding more useful information.

## 2.2 Merge Conflict

In Git, a repository is created locally and most of the development is done in the local environment, and work is done in units called branches. Merge is a way to bring branches back. The `git merge` command is a command created using git branch to merge multiple independent branches. However, if an edit in the same area is detected in both commits, a merge conflict will occur.

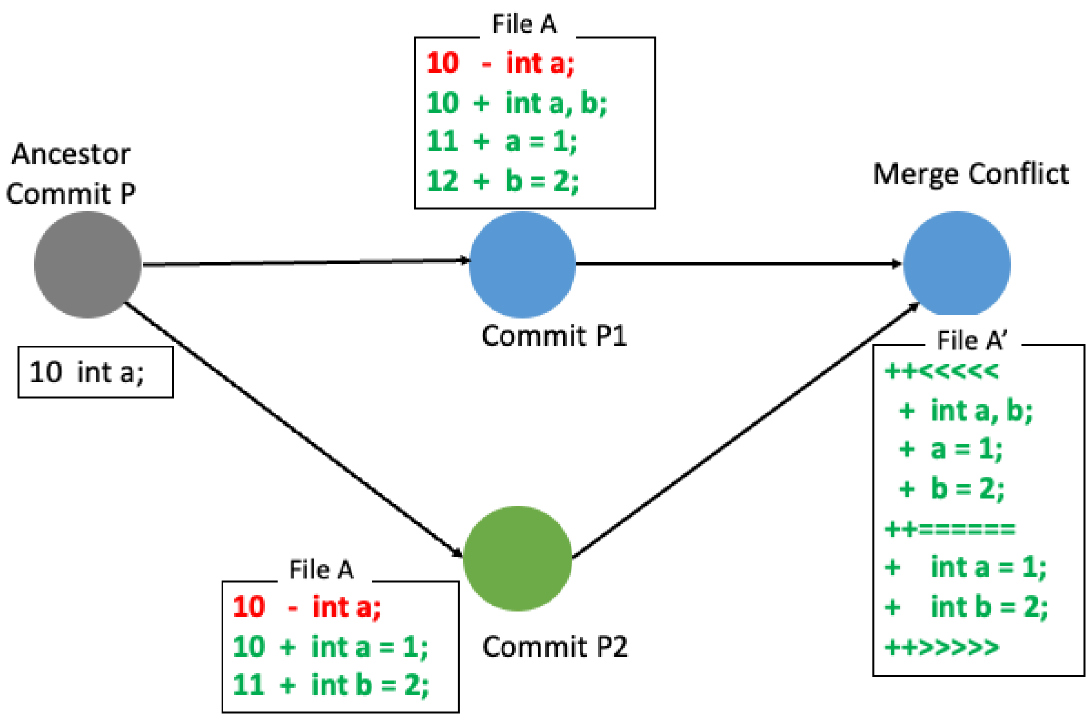
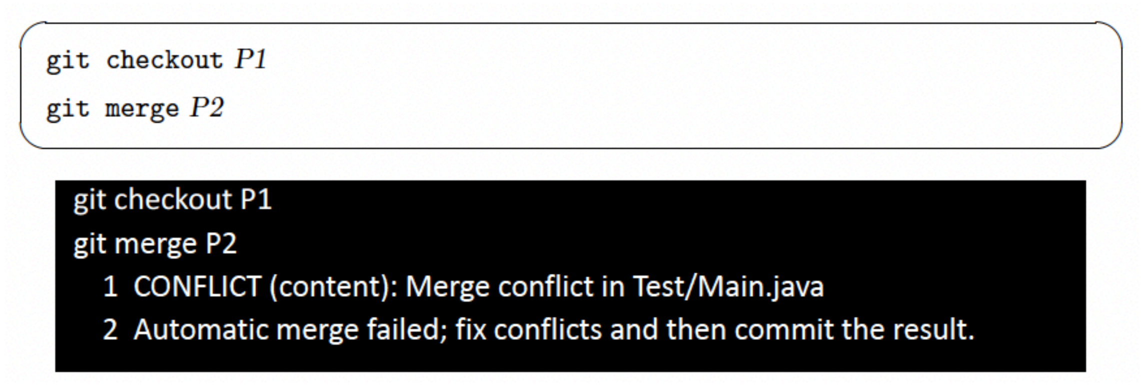


Figure 1: An example of merge conflict
(source:Shiraki, 開発履歴のメタ情報を用いたマージコンフリクト解消支援手法, 2020, p5)


```
git checkout P1
git merge P2
```

```
git checkout P1
git merge P2
   1  CONFLICT (content): Merge conflict in Test/Main.java
   2  Automatic merge failed; fix conflicts and then commit the result.
```

Figure 2: When a merge conflict occurs
(source:Shiraki, 開発履歴のメタ情報を用いたマージコンフリクト解消支援手法, 2020, p7)

In merging, 3-way merging is common. Figure 1 shows an example of a 3-way merge conflict. A branch is created in ancestor commit P, and commit P1 is created with edits to line 10 of file A. Next, another branch is created from commit P, and commit P2 is created with another compilation for the line 10 of file A. Commit P1 and P2 are a commit pair to merge. Then, commits P1 and P2 try to merge their respective revisions. Currently, there are duplicates in the edited part. A merge conflict occurs in file A'.

In addition to textual conflicts, there are merge conflicts called build conflicts and test conflicts [1, 7]. Build conflicts and test conflicts appear to be successfully merged in the text, When you actually run a program build or test, you get an error due to a merge. Building conflicts and test conflicts have been found to be less than half as likely as textual conflicts [7]. Therefore, this study does not deal with build conflicts, and hereafter, merge conflicts refer to textual conflicts.

## 2.3 Detection and Solution of Merge Conflict

When investigating a merge conflict from the development history, we should check whether a merge conflict has occurred in a merge commit with two commits P1 and P2. If the following command is executed and a merge conflict occurs in commits P1 and P2, CONFLICT is output as shown in the first line of the output result in Figure 2.

In addition, regarding the specific method of resolving merge conflicts, there are many cases where the merge conflict is resolved by adopting the editing of one of the commit pair and deleting the other one as shown in Figure 1. This is called one-sided adoption. A study by Yuzuki reveals that about 98% of all merge conflicts that occur within methods in Java projects are resolved by one-sided adoption [8].

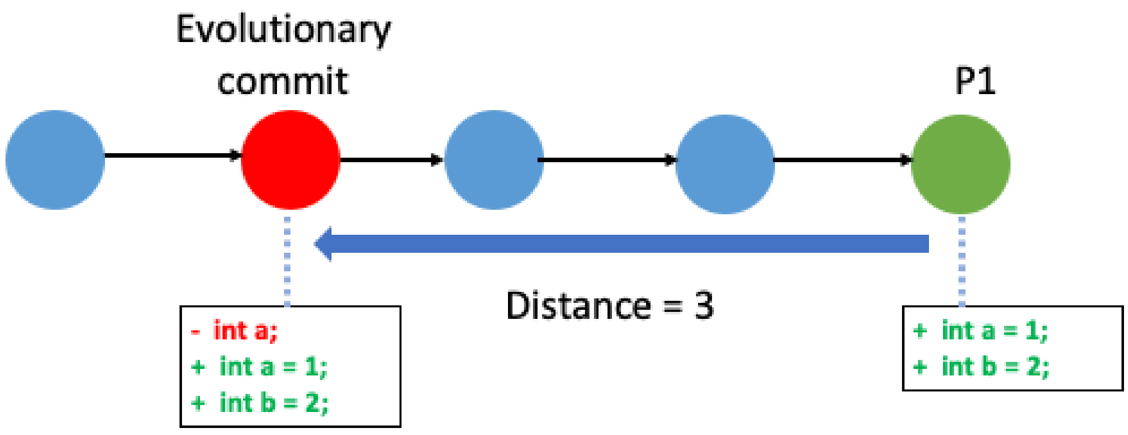


Figure 3: Evolutionary commit and the distance
(source:Shiraki, 開発履歴のメタ情報を用いたマージコンフリクト解消支援手法, 2020, p12)

### 2.4 Evolutionary Commit and Distance

In each of the commits P1 and P2 where the merge conflict occurred, there is a commit edited in the branch from the common ancestor to P1 and P2 where the merge conflict occurred. Of all the edited locations where merge conflicts occur, the commits closest to P1 and P2 are called Evolutionary Commits [9]. In the previous research, the distance between the Evolutionary Commit and the commit in which the merge conflict occurred was obtained and used to create the model. The number between green and red commit in Figure 3 is the distance explained above.

### 2.5 Source Code Metrics

Source code metrics are measurements of various aspects of software code. Some metrics are at an high level, spanning the entire code, while others are at a lower level, covering classes, methods, or even smaller blocks of code. For example, the number of the lines of code, the number of comments in the code, the number of variables, functions, developers, and so on. In these metrics, a lot of valuable information about the program is hidden. For example, in Meirelles ’ s study, the relationship between source code metrics and the attractiveness in free software projects has become clear [10]. They suggest software projects with higher structural complexity have lower attractiveness. On the other hand, projects with more lines of code have higher attractiveness. Lanubile et al. [6] shows that source code metrics are related to the defects of the program. Since the source code metrics can tell us so many stories, it is quite possible that the metrics we get from the source code in the development history would give us useful information related to merge

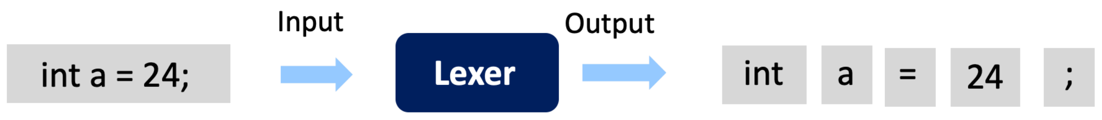


Figure 4: Flow of the input and output of Lexer

conflict.

## 2.6 ANTLR

ANTLR[1] or ANother Tool for Language Recognition is a Lexer and parser generator aimed at building and walking parse trees. It makes it effortless to parse nontrivial text inputs such as a programming language syntax. It is a tool that automatically generates a so-called analyzer, which is a function required for parsing an abstract syntax tree, using a grammar file (g4 file) as input. Since the grammar file does not depend on the programming language of the analyzer, it is possible to generate an analyzer for multiple programming languages. The analyzer consists of three parts: Lexer, Parser, and Listener. In the experiment we only used Lexer, Figure 4 shows the flow of how Lexer works.

Lexer is a procedure that analyzes character strings such as natural language sentences and programming language source code to obtain a sequence of "tokens", which is the smallest unit in parsing(shown in Figure 4). Parser is a process that reads the text to be analyzed and decomposes it into a syntax tree. Listener is an API for users to create their own analyzer.

In this study, in order to obtain the source code metrics, Lexer is used to decompose it into the smallest unit "token", and the source code metrics are obtained from there. In this paper, we will analyze Java projects and Python projects. Since a grammar file is required for each language, it is necessary to generate Lexer using the grammar files for Java language and Python language, respectively.

[1]https://www.antlr.org/

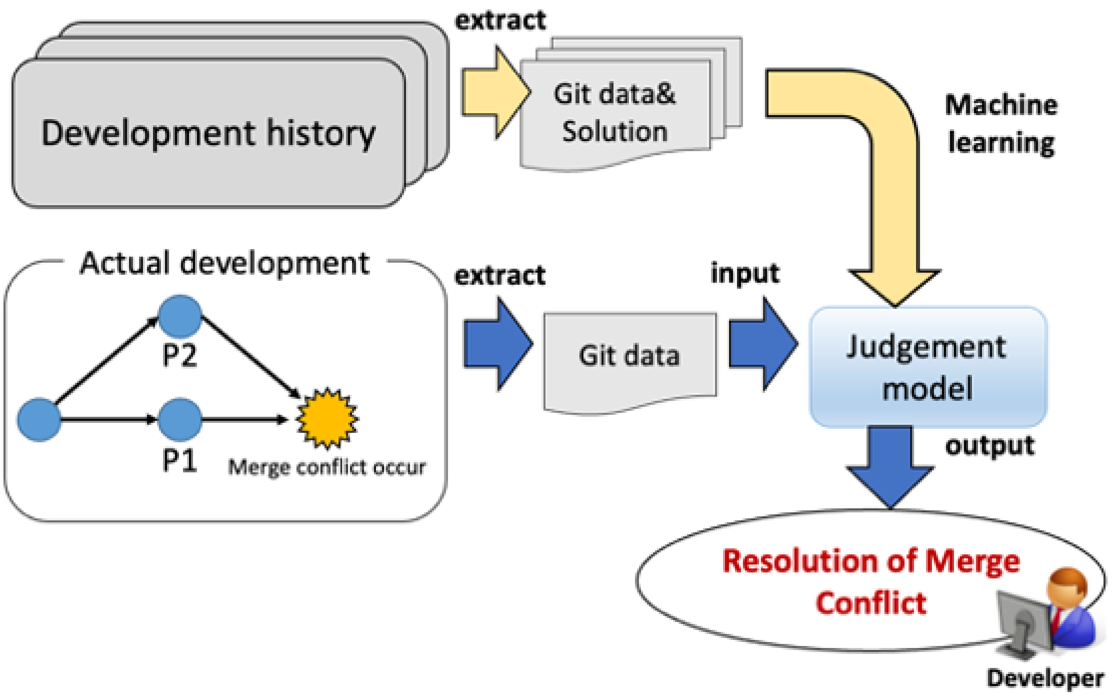


Figure 5: Merge conflict resolution model proposed in previous research
(source:Shiraki, 開発履歴のメタ情報を用いたマージコンフリクト解消支援手法, 2020, p9)

## 3 Previous research

In this research, we extend the model created by Shiraki. In this chapter, we will explain the previous research by Shiraki and his results.

### 3.1 Experimental Approach

They aimed to suggest to developers how to resolve merge conflicts when they occur. According to the research by Yuzuki [8], it is often resolved by adopting one and deleting the other for the commit pair when the merge conflict occurred. Based on this result, Shiraki proposed a method for resolving merge conflicts using machine learning. They created a judgement model (shown in Figure 5) for determining the method for resolving merge conflicts from development history information related to merge conflicts that occurred in the past. To build the model, random forest is used as the learning algorithm.

Table 1: Project used in the experiment

| | | |
|---|---|---|
| beam | camel | cassandra |
| cordova-android | curator | dubbo |
| flink | geode | groovy |
| hbase | hive | ignite |
| incubator-heron | jmeter | lucene-solr |
| mahout | maven | nifi |
| nutch | rocketmq | |

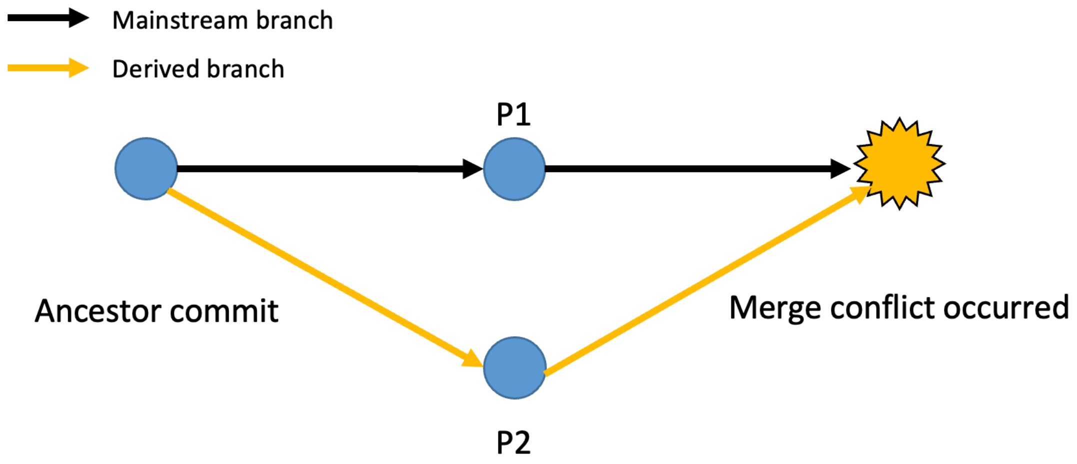


Figure 6: Distinguishing Commit Pairs with Merge Conflicts (source:Shiraki, 開発履歴のメタ情報を用いたマージコンフリクト解消支援手法, 2020, p10)

## 3.2 Data Collection

In the previous research, 20 Java projects were shown in table one collected from the OSS provided by Apache.

The determination of the resolution method by the judgment model is performed for each commit pair in which a merge conflict has occurred. As shown in Figure 6, to distinguish each commit of the commit pair, P1 is the commit on the mainstream branch and P2 is the commit on the newly derived branch.

The resolution method proposed to developers is a set of resolution methods for each commit (P1 resolution method / P2 resolution method). Table 2 defines the list of resolution methods for each commit.

Table 2: Definition of merge conflict resolution method in Shiraki's study

| | |
|---|---|
| ADOPT | Adopt all edits |
| DELETE | Delete all edits |
| EDIT | Make additional edits, adopt part of the edits, or both |
| ZERO | 0 lines to edit |

Table 3: Parameters used to build the model

| | Parameter | Parameter meaning |
|---|---|---|
| P1 | linenum(P1) | Number of lines in P1 where merge conflicts occur |
| | time(P1) | P1 commit creation date and time |
| | author ratio(P1) | ratio of commit made by P1 creater to total commit number |
| | distance(P1) | Distance between P1 and its Evolutionary Commit |
| P2 | linenum(P2) | Number of lines in P2 where merge conflicts occur |
| | time(P2) | P2 commit creation date and time |
| | author ratio(P2) | ratio of commit made by P2 creater to total commit number |
| | distance(P2) | Distance between P2 and its Evolutionary Commit |
| Difference | linenum(d) | linenum(P2) - linenum(P1) |
| | time(d) | time(P2) - time(P1) |
| | author ratio(d) | author ratio(P2) - author ratio(P1) |
| | distance(d) | distance(P2) - distance(P1) |

Development history information was acquired from the repository for each commit P1 and P2 in which a merge conflict occurred. Table 3 is the list of parameters used to create the machine learning model.

### 3.3 Experiment result

20 Java projects collected from the OSS provided by Apache were used for the experiment. The correct answer rate is a result of cross-validation by dividing the merged conflict data into five for each project. The correct answer rate was 66.41% on average and 94.43% at maximum. It became clear that the solution method can be determined with high accuracy.

To find out which parameter the created model determines the resolution method, an index called importance is used. Importance is the percentage of each parameter contributing to the

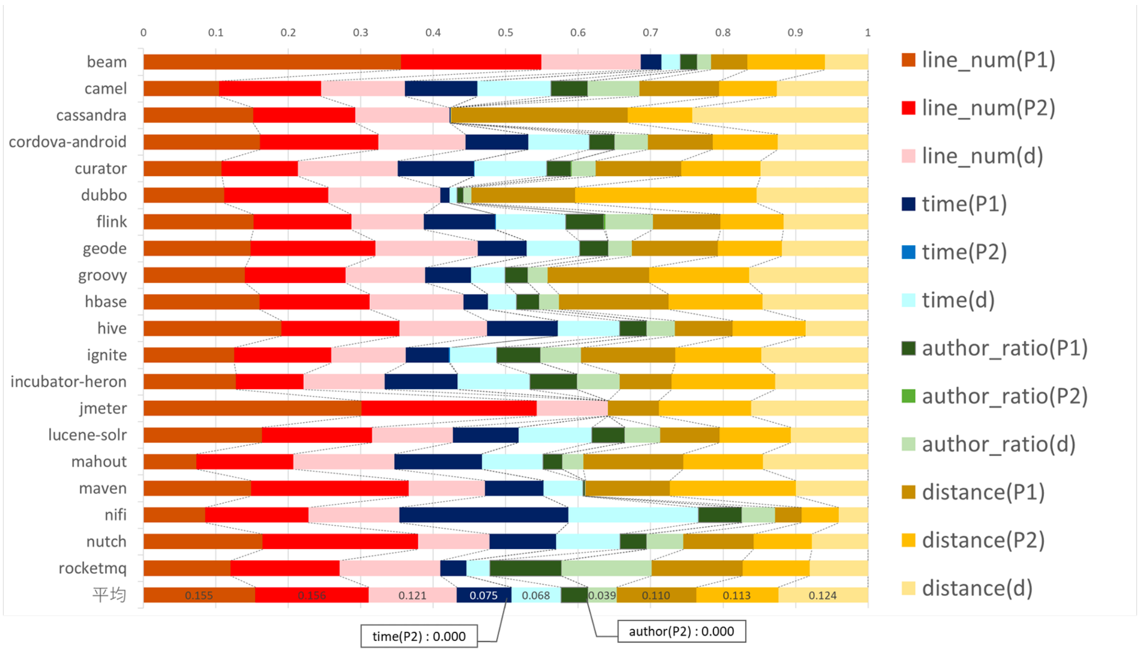


Figure 7: The Parameter importance in previous research (source:Shiraki, 開発履歴のメタ情報を用いたマージコンフリクト解消支援手法, 2020, p21)

classification in the model, and the sum of the importance of all parameters is 1. Figure 7 shows the importance of the parameter in each project. The number of lines line_num (P1) and line_num (P2) where merge conflicts occur are both as large as 0.15 or more. Also, from the figure, there are many projects whose importance is over 0.1 even for lines (d). So, it can be said that the number of lines where conflicts occur greatly contributes to the determination of how to resolve merge conflicts in any project.

I will add source code metrics to this model to see how the accuracy changes. And I would like to investigate whether the model can be used in Python, which is also an object language. Obtaining metrics for each language may reveal language-specific characteristics. I attempt to improve the accuracy of the model by using both development history data and source code metrics for each language.

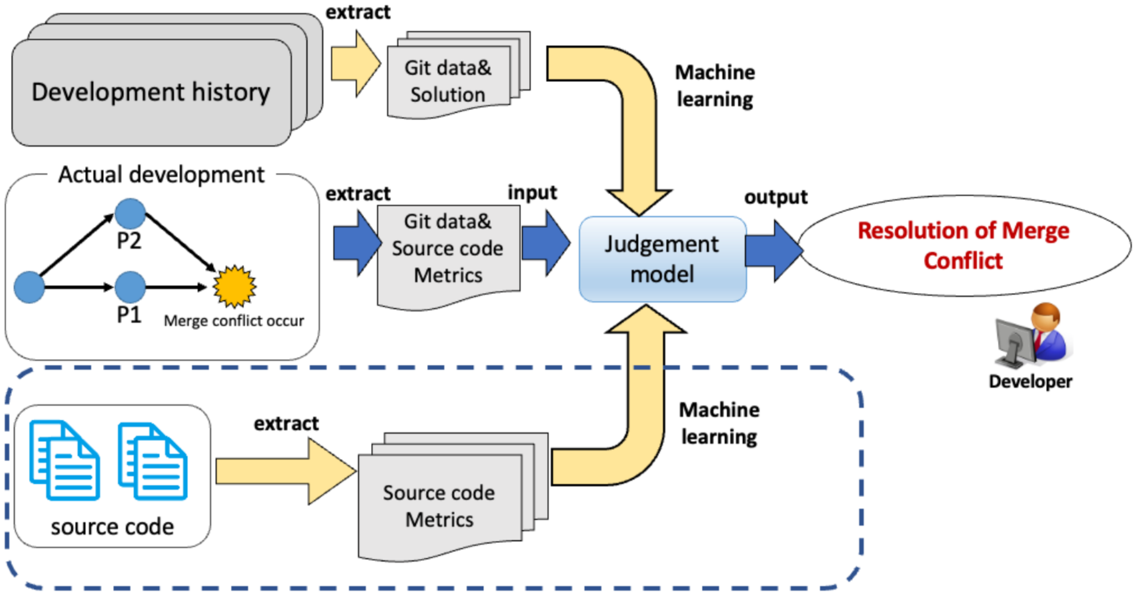


Figure 8: An overview of the model proposed in this research

## 4 Model Expansion

The goal of this research is to improve the prediction accuracy by improving the existing feature and adding new metrics extracted from source code. An overview of the model is shown in Figure 8.

Shiraki's method [4] used only language-independent features. However, as introduced in Chapter 2, source code metrics also possibly have an effect on software quality. So, we add source code metrics to this model to see how the accuracy changes. We also aim to verify whether a judgment model should be created for each language or regardless of the language. Since merge conflicts often occur in parallel development using Git regardless of programming language, another goal of this research is to help resolve merge conflicts not only in Java language projects but also in other languages.

### 4.1 New judgment model by adding code metrics

Some studies have also shown that bugs lead to merge conflicts [5]. In other research, it has become clear that the complexity of the program calculated by the source code metrics is also related to the defects of the program [6]. Therefore, it is quite possible that there is a connection between merge conflict and the complexity of the program calculated by the source code metrics.

Table 4: Source code metrics to get

| | | |
|---|---:|---|
| P1 | Name_Num(P1) | the number of variables and functions in P1 |
| | Operator_Num(P1) | the number of operators in P1 |
| | Keyword_Num(P1) | the number of keywords in P1 |
| P2 | Name_Num(P2) | the number of variables and functions in P2 |
| | Operator_Num(P2) | the number of operators in P2 |
| | Keyword_Num(P2) | the number of keywords in P2 |
| Difference | Name_Num(d) | Name_Num(P2) - Name_Num(P1)) |
| | Operator_Num(d) | Operator_Num(P2) - Operator_Num(P1) |
| | Keyword_Num(d) | Keyword_Num(P2) - Keyword_Num(P1) |

There are many metrics that describe the complexity of a program. Cyclomatic complexity, number of lines of code, number of methods, number of function calls, and so on. It is conceivable that complicated programs tend to have a large amount of information. Metrics that represent the amount of information that can be easily obtained including the number of lines in the program in which the merge conflict occurred, the number of variables and functions, the number of operators, and the number of keywords. And since the number of lines of each commit has already been acquired in previous research, this time we will add the number of variables and functions, the number of operators, and the number of keywords words as source code metrics for building the new model. Table 4 below shows the new metrics to get for the judgment model.

## 4.2 The method of extracting code metrics

The method of getting the code metrics from development history can be described into several steps:

1. Get the development history file contents corresponding to file path and commit
2. Generate the Lexer of Java or Python according to the project language
3. Pass the file contents as strings to Lexer to obtain a sequence of “tokens ”
4. Implement a program to stat the number of the metrics in need

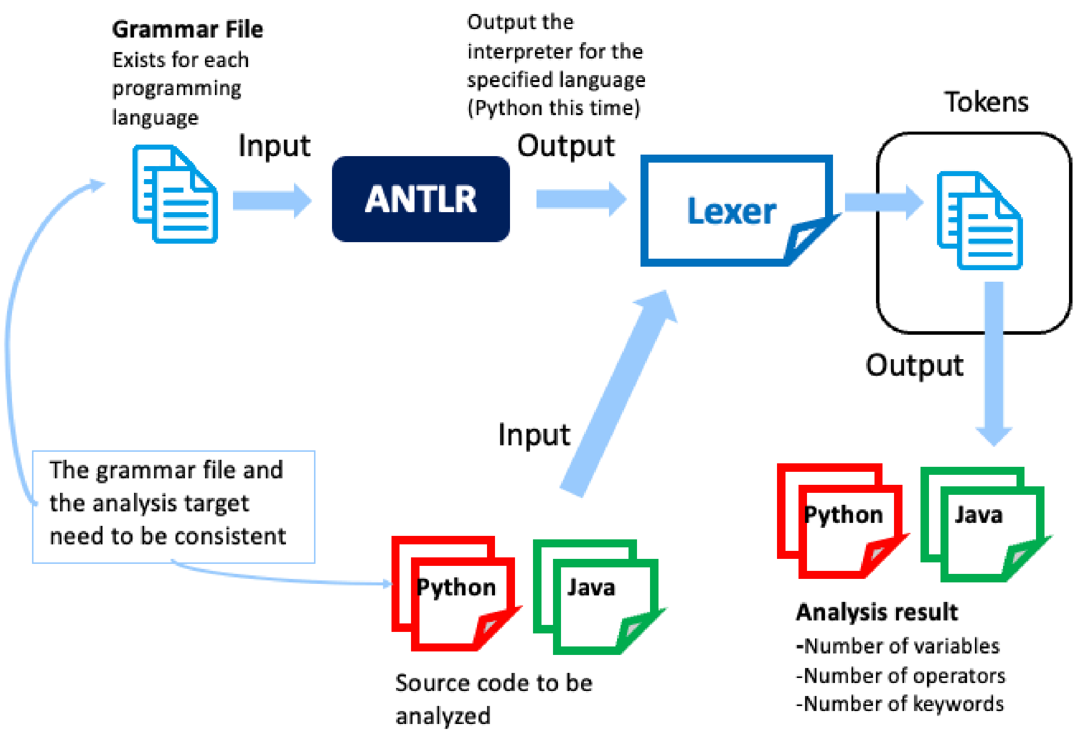

Grammar File
Exists for each programming language
Input
ANTLR
Output the interpreter for the specified language (Python this time)
Output
Lexer
Tokens
Output
Input
The grammar file and the analysis target need to be consistent
Python
Java
Source code to be analyzed
Python
Java
Analysis result
-Number of variables
-Number of operators
-Number of keywords

Figure 9: The flow of source code metrics extraction

First of all, I used git show command: "git show commitHash:/path/to/file" to get the developing history file contents corresponding to file path and commit hash. The way of getting the source code metric is by using a language recognizer. I used ANTLR introduced in chapter 2.7 to generate a Lexer from Java and Python grammar file respectively. Then I used Lexer of each language to analyze projects in each programming language source code to obtain a sequence of "tokens", which is the smallest unit in parsing. Figure 9 describes the flow of source code metrics extraction.

According to the grammar file, thirty-four kinds of operator and fifty kinds of keyword are defined for Java and thirty-nine kinds of operator, and thirty-five kinds of keyword are defined for Python. Table 5 shows the keyword of Java and Python language.

Finally, I wrote a program to stat the number of variables and functions, the number of operators, and the number of keywords from the tokens, which are the source code metrics needed for building the new model.

Table 5: Keywords of Java and Python language

| keywords of Java language(50 in total) | | | | |
|---|---|---|---|---|
| abstract | continue | for | new | switch |
| assert | default | if | package | synchronized |
| boolean | do | goto | private | this |
| break | double | implements | protected | throw |
| byte | else | import | public | throws |
| case | enum | instanceof | return | transient |
| catch | extends | int | short | try |
| char | final | interface | static | void |
| class | finally | long | strictfp | volatile |
| const | float | native | super | while |

| keywords of Python language(35 in total) | | | | |
|---|---|---|---|---|
| False | None | True | and | as |
| assert | async | await | break | class |
| continue | def | del | elif | else |
| except | finally | for | from | global |
| if | import | in | is | lambda |
| nonlocal | not | or | pass | raise |
| return | try | while | with | yield |

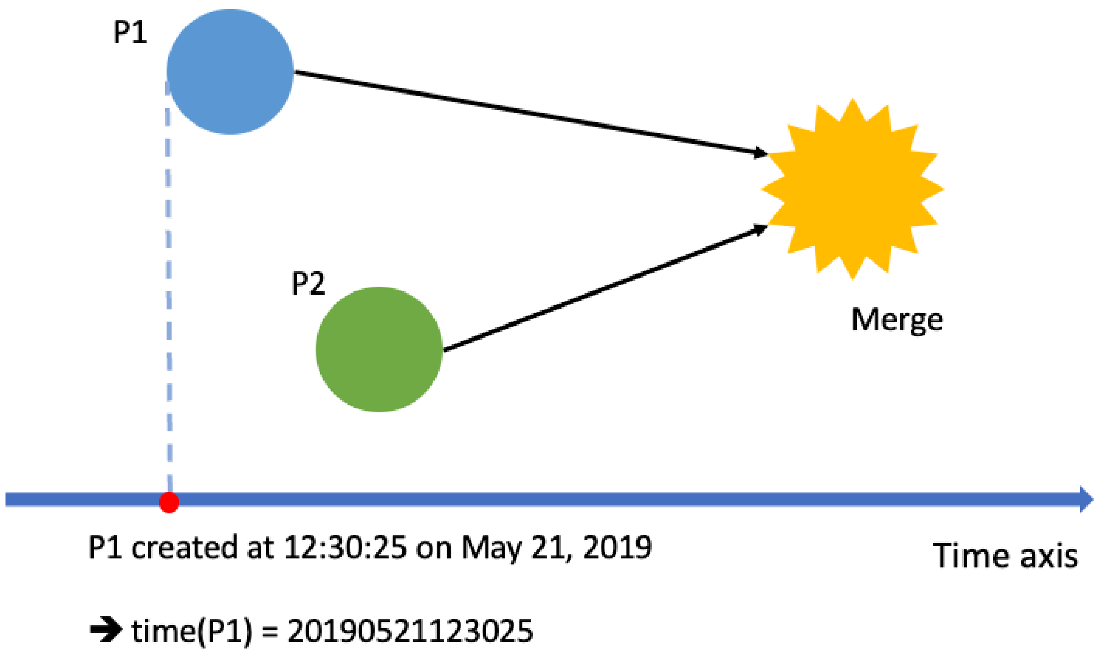

P1
P2
Merge
P1 created at 12:30:25 on May 21, 2019
Time axis
➔ time(P1) = 20190521123025

Figure 10: The time feature used in previous research

## 4.3 Parameter Improvement

In the previous research, the creation dates and times of commits P1 and P2 were acquired and used as features of the learning model. For example, if a commit was created at 12:30:25 on May 21, 2019, it would feature an 8-digit date " 20190521 " and a 6-digit time " 123025 " , a total of 14-digit numbers " 20190521123025 " (shown in Figure 10).

It is a quantity and since the subtraction of the time of P1 and the time of P2 is also subtracted as a decimal number of 14 digits, its usage is a little inconsistent with the concept of time. In Costa et al.'s research [11], factors that contribute to the occurrence of conflicts are listed as number of changed files, number of changed lines, number of commits, number of developers, branching-duration, lack of communication, developer working in several branches and so on. They asked 109 software developers to conduct a survey on factors that they think lead to conflicts. In this question, participants were allowed to mark more than one answer. 76 (69.7%) developers marked the option "branching-duration". From this result, it is considered that the time from each commit's creation time to the merge time is more related to the merge conflict than the commit creation date and time. Therefore, I calculated the duration from each commit's creation time to the merge time as Time1 and Time2 in seconds. In figure 7, Fixed_time1, Fixed_time2 show the time from

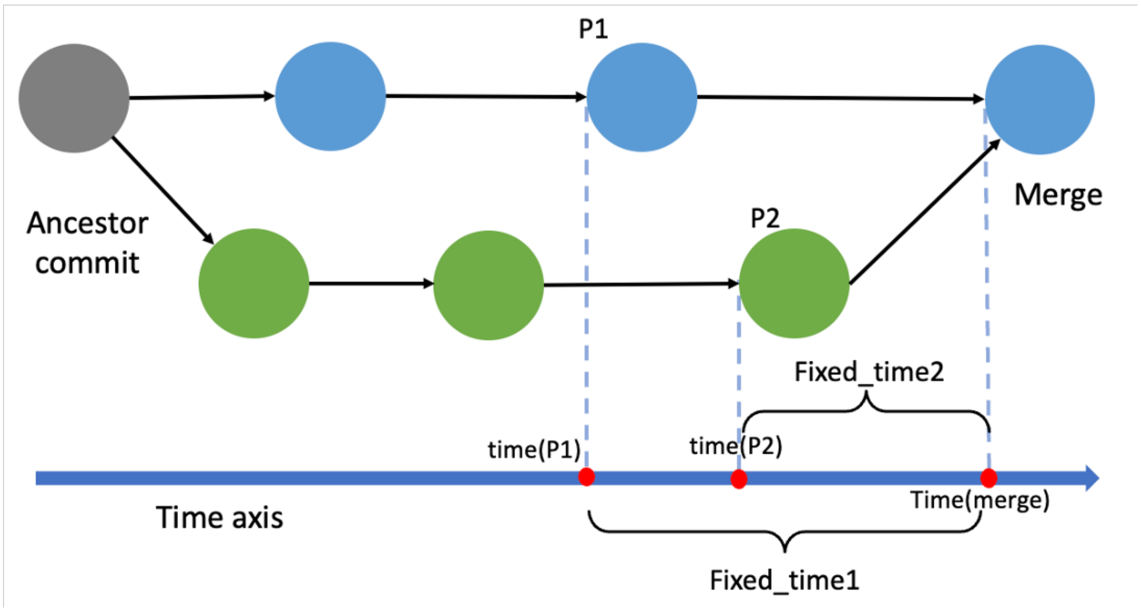


Figure 11: The new time feature `Fixed_time1`, `Fixed_time2` used in this research

each commit's creation time to the merge time, which are used as features for the new model.

Table 6: Parameters (source code metrics) used in the experiment

| | parameter | meaning |
|---|---|---|
| P1 | linenum(P1) | Number of lines in P1 where merge conflicts occur |
| | Fixed_time(P1) | duration from P1 creation time to merge time |
| | author ratio(P1) | ratio of commit made by P1 creater to total commit number |
| | distance(P1) | Distance between P1 and its Evolutionary Commit |
| P2 | linenum(P2) | Number of lines in P2 where merge conflicts occur |
| | Fixed_time(P2) | duration from P1 creation time to merge time |
| | author ratio(P2) | ratio of commit made by P2 creater to total commit number |
| | distance(P2) | Distance between P2 and its Evolutionary Commit |
| Difference | linenum(d) | linenum(P2) - linenum(P1) |
| | Fixed_time(d) | Fixed_time(P2) - Fixed_time(P1) |
| | author ratio(d) | author ratio(P2) - author ratio(P1) |
| | distance(d) | distance(P2) - distance(P1) |

## 5 Evaluation

In order to investigate the effectiveness of the source code metrics for the judgment model, we conducted evaluation experiments on Java projects and Python projects with and without the source code metrics. Same with Shiraki's research, 20 Java projects were used from the OSS projects published by the Apache Software Foundation. For the Python project, 15 new OSS projects published by the Apache Software Foundation are selected in order of popularity. Projects with no conflicts and projects with extremely few conflicts (less than 10) are excluded, and 8 projects remained. table 6 shows modified parameter used for judgement model and table 7 shows the list of projects used in the experiment and the number of merge conflict.

Twenty java projects and seven python projects were used in the experiment. For each project, the correct answer rate is a result of cross-validation by dividing the merged conflict data into five. The learning algorithm Random Forest was used.

After modifying the parameter in Shiraki's research, the accuracy of the model increased from 66.41% to 74.99% in case of Java project, and 55.32% to 61.87% in case of Python project. From the result, it is clear that the new parameter `Fixed_time1, Fixed_time2` defined contribute more to the model.

Table 7: List of Java and Python projects used in the experiment

| Java project(20 in total) | |
|---|---|
| project name | # merge conflict |
| beam | 1449 |
| camel | 44 |
| cassandra | 17837 |
| cordova-android | 973 |
| curator | 287 |
| dubbo | 401 |
| flink | 3454 |
| geode | 573 |
| groovy | 353 |
| hbase | 187 |
| hive | 3246 |
| ignite | 2850 |
| incubator-heron | 61 |
| jmeter | 693 |
| lucene-solr | 2562 |
| mahout | 195 |
| maven | 251 |
| nifi | 157 |
| nutch | 658 |
| rocketmq | 56 |

| Python project(8 in total) | |
|---|---|
| project name | # merge conflict |
| airavata | 72 |
| airflow | 61 |
| allura | 28 |
| cassandra | 200 |
| incubator-datalab | 449 |
| incubator-spot | 10 |
| libcloud | 2082 |
| predictionio | 14 |

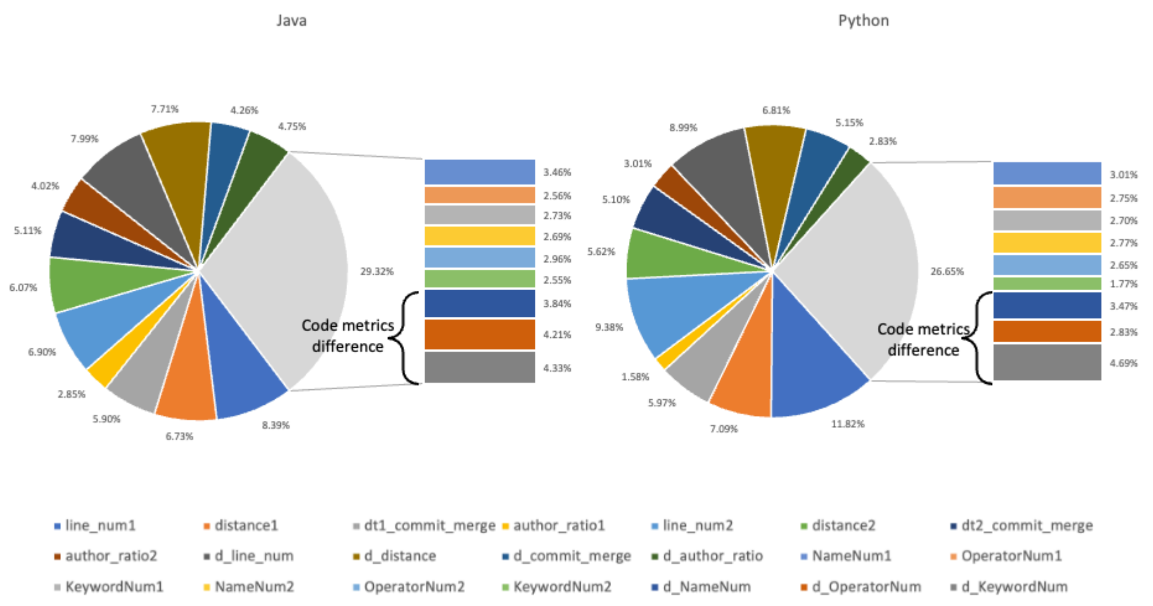


Figure 12: Importance of source code metrics in case of Python and Java

## 5.1 RQ1: How will the results change if both developing history data and language-specific source code metrics are used?

In the case of Java, the average changed from 74.99% to 75.54%. In Python, the average changed from 61.87% to 61.76%, which were almost unchanged.

Figure 12 shows the ratio of source code metrics to all features. Source code metrics (9 in total) are 29.32% for Java and 26.65% for python (the grey area pulled out by the line). Among all the 9 source code metrics, `d_NameNum`, `d_OperatorNum`, and `d_KeywordNum` came to the top in both Java and Python projects. The rankings are different, but the top three are exactly the same. Therefore, from the source code metrics extracted from each commit pair, it was found that the difference between the source code metrics of the two commits contributes more to the result of the judgment model than the metrics extracted from each commit. It is conceivable that source code metrics are related to the amount of information in a program, and programs with a large amount of information tend to have merge conflicts. However, it became clear that the difference between the code metrics of two commits contributed to the result.

Since the difference of code metrics from two commit tend to have stronger relevance, we made another experiment of the model only consist of the difference of code metrics and parameters used in Shiraki's model. Table 8 shows the parameter and the accuracy of each model. Figure 13 and Figure14 show the result of 3 different models in the case of Java and Python project.

Table 8: Parameters used in each model

| parameter | improved Shiraki's model | expanded model | expanded model (difference only) |
|---|---|---|---|
| linenum(P1) | ✓ | ✓ | ✓ |
| Fixed_time(P1) | ✓ | ✓ | ✓ |
| author ratio(P1) | ✓ | ✓ | ✓ |
| distance(P1) | ✓ | ✓ | ✓ |
| linenum(P2) | ✓ | ✓ | ✓ |
| Fixed_time(P2) | ✓ | ✓ | ✓ |
| author ratio(P2) | ✓ | ✓ | ✓ |
| distance(P2) | ✓ | ✓ | ✓ |
| linenum(d) | ✓ | ✓ | ✓ |
| Time(d) | ✓ | ✓ | ✓ |
| author ratio(d) | ✓ | ✓ | ✓ |
| distance(d) | ✓ | ✓ | ✓ |
| Name_Num(P1) | | ✓ | |
| Operator_Num(P1) | | ✓ | |
| Keyword_Num(P1) | | ✓ | |
| Name_Num(P2) | | ✓ | |
| Operator_Num(P2) | | ✓ | |
| Keyword_Num(P2) | | ✓ | |
| Name_Num(d) | | ✓ | ✓ |
| Operator_Num(d) | | ✓ | ✓ |
| Keyword_Num(d) | | ✓ | ✓ |
| Java Accuracy | 74.99% | 75.54% | 75.01% |
| Python Accuracy | 61.87% | 61.76% | 65.77% |

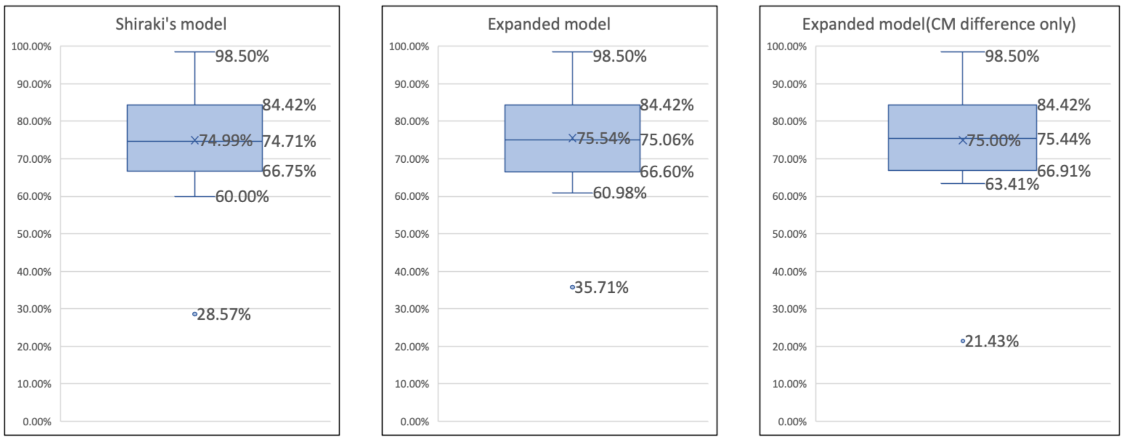


Figure 13: Result of Python with/without code metrics

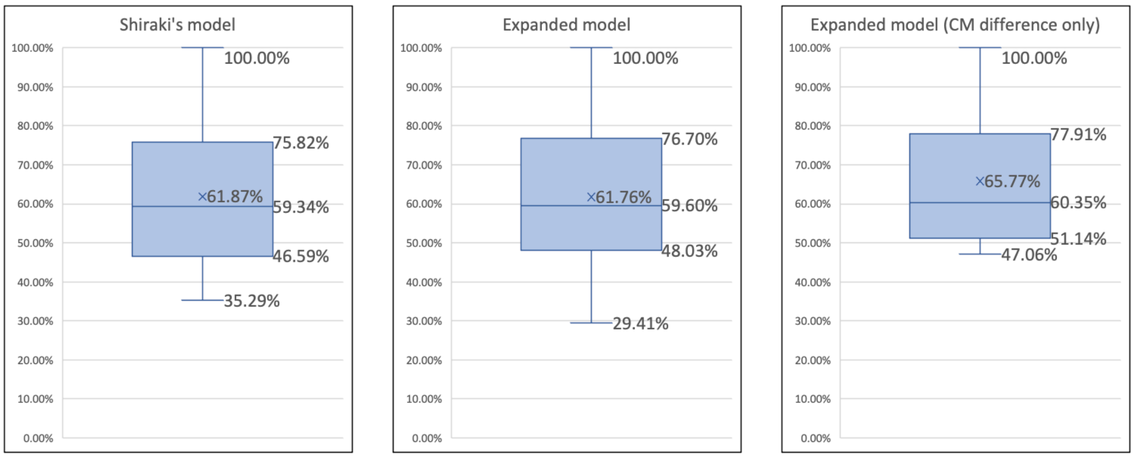


Figure 14: Result of Python with/without code metrics

As a result, source code metrics are about one-third as important in the two languages. However, there was a variation in the total importance of the source code metrics. Table 9 shows the importance of code metrics in each Java project.

In case of Java project, the maximum was 55.37% for project jmeter, compared to only 11.49% for project hbase. In the 20 Java project, 9 project's accuracy increased, 6 stayed the same and 5 decreased. When I visually checked the source code metrics, the difference between the source code metrics obtained from each commit `d_NameNum`, `d_OperatorNum`, and `d_KeywordNum` were large in of some of the projects. These projects' source code metrics tend to have larger importance. On the other hand, when the metrics taken from each commit are almost the same, `d_NameNum`, `d_OperatorNum`, and `d_KeywordNum` would be almost 0. In this case, source code metrics did not make contributions to the decision model.

From the above, it became clear that source code metrics are not useful if each commit does not change the number of names, the number of operators, and the number of keywords. For example, if two developers only change the name of the variable or function, the number of names stays the same. Also, if two developers add a similar piece of code or change the order of the code in a program, the difference between two commits still stays the same as before. In this kind of situation, source code metrics may not contribute as much as we expected.

In conclusion, the source code metrics may not contribute to the model as much as the feature existed, but the difference of the code metrics from each commit pair surely has an influence on the model.

Table 9: Importance of source code metrics (CM) in each Java project

| project name | # conflicts | Shiraki model | expanded model | CM importance |
|---|---|---|---|---|
| beam | 1449 | 96.28% | 95.91% | 31.54% |
| camel | 44 | 83.33% | 83.33% | 37.36% |
| cassandra | 17837 | 65.50% | 66.58% | 35.04% |
| cordova-android | 973 | 70.00% | 71.43% | 21.55% |
| curator | 287 | 69.12% | 66.18% | 37.48% |
| dubbo | 401 | 65.96% | 61.70% | 17.23% |
| flink | 3454 | 76.10% | 76.26% | 29.78% |
| geode | 573 | 73.28% | 72.41% | 24.47% |
| groovy | 353 | 91.04% | 89.55% | 35.71% |
| hbase | 187 | 60.98% | 60.98% | 11.49% |
| hive | 3246 | 71.33% | 70.90% | 23.62% |
| ignite | 2850 | 79.01% | 81.23% | 27.52% |
| incubator-heron | 61 | 28.57% | 35.71% | 28.29% |
| jmeter | 693 | 98.50% | 98.50% | 55.37% |
| lucene-solr | 2562 | 73.32% | 73.87% | 19.04% |
| mahout | 195 | 91.11% | 91.11% | 23.14% |
| maven | 251 | 84.78% | 84.78% | 36.64% |
| nifi | 157 | 60.00% | 66.67% | 29.29% |
| nutch | 658 | 79.31% | 81.38% | 33.78% |
| rocketmq | 56 | 82.35% | 82.35% | 28.10% |
| average | 1814.35 | 74.99% | 75.54% | 29.32% |

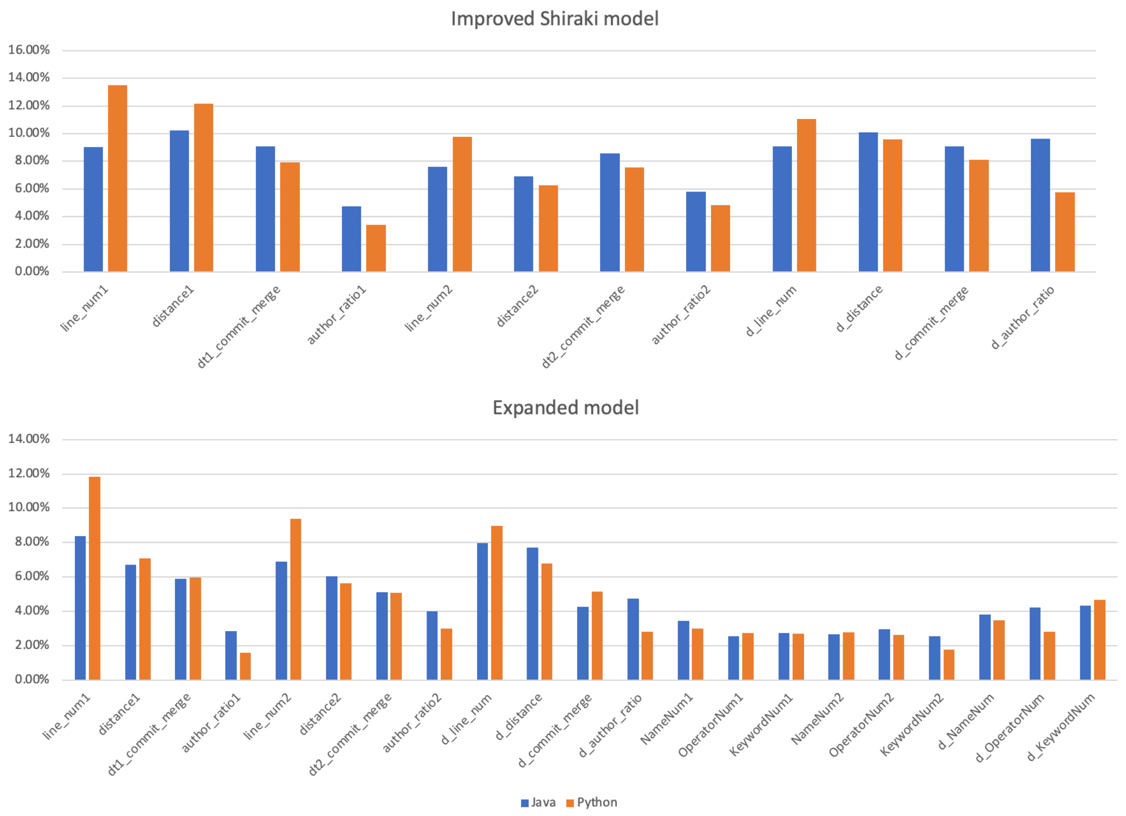


Figure 15: Feature importance in case of Python and Java

## 5.2 RQ2: How is the developing history data that contributes to the judgment different between Java and Python language projects?

There are 20 features in total, including 11 metrics acquired from the development history and 9 extra source code metrics extracted from the source code this time. In Figure 15, each feature is a comparison of importance in Java and python. From this figure, it can be seen that Java project features that are of high importance tend to have high importance in Python as well. The same result came up even when building the model without the source code metrics. Table 10 shows importance of all twenty features in largest order. Features in bold are source code metrics. It became clear that `line_num`, `distance`, `Fixed_time`, and the differences in the source code metrics of commit pair were of high importance. In conclusion, it was found that the source code metrics extracted from each language had no language-dependent characteristics, in other words, Java and Python language don't have obvious difference in number of variables and functions, operators and keywords.

In this research, we got the source code metrics from the entire file for each commit pair. From

Table 10: Ranking of feature importance

| ranking | Java | | Python | |
|---|---|---|---|---|
| 1 | line_num1 | 8.55% | line_num1 | 10.93% |
| 2 | d_line_num | 8.50% | d_line_num | 8.44% |
| 3 | d_distance | 7.61% | line_num2 | 8.40% |
| 4 | distance1 | 7.51% | distance1 | 7.90% |
| 5 | line_num2 | 7.02% | d_distance | 7.59% |
| 6 | distance2 | 6.40% | distance2 | 6.67% |
| 7 | Fixed_time2 | 5.74% | Fixed_time1 | 5.50% |
| 8 | Fixed_time1 | 5.43% | **d_NameNum** | 4.79% |
| 9 | **d_KeywordNum** | 4.98% | Fixed_time2 | 4.57% |
| 10 | author_ratio2 | 4.19% | **d_KeywordNum** | 4.55% |
| 11 | **d_OperatorNum** | 4.06% | **d_OperatorNum** | 4.11% |
| 12 | **d_NameNum** | 3.96% | **NameNum2** | 4.07% |
| 13 | d_author_ratio | 3.90% | author_ratio2 | 3.55% |
| 14 | author_ratio1 | 3.64% | **OperatorNum2** | 3.38% |
| 15 | **NameNum2** | 3.57% | **KeywordNum2** | 3.36% |
| 16 | **KeywordNum1** | 3.43% | d_author_ratio | 3.00% |
| 17 | **OperatorNum1** | 2.98% | **KeywordNum1** | 2.61% |
| 18 | **KeywordNum2** | 2.98% | **OperatorNum1** | 2.46% |
| 19 | **OperatorNum2** | 2.85% | **NameNum1** | 2.27% |
| 20 | **NameNum1** | 2.68% | author_ratio1 | 1.86% |

the result, the difference between the source code metrics extracted from commit pairs contributed more to the judgment result than metrics from each commit. Therefore, instead of the whole file, source code metrics extracted from lines where conflict occurs may be more useful for the model creation.

In conclusion, the features used for building the model don't have obvious differences between Java and Python language.

## 6 Limitations

All the projects used in this study are OSS provided by Apache, and there is no certainty that the results will be similar to the results of this study in other OSS or other commercial projects. Also, among the OSS provided by Apache, the number of python projects is smaller than Java, so it is difficult to evaluate with data of the same scale. In the future, I think it will be necessary to conduct research not only on Apache but also on various projects in various languages.

Also, in this research, as the information used for the parameters of model creation, we traced back to the development history and used the metrics extracted from the source code of the file during the development. However, it is possible that some hidden valuable information that has not been used this time, so the combination of parameters used by the model created in this study is not always optimal. However, from the correct answer rate obtained in this study, it can be said that the model in this study may be useful for resolving merge conflicts.

## 7 Conclusion

Using the information in the development history, we proposed a model that traces back to the file contents during the development phase, extracts the metrics, and contributes to a judgment model based on the source code metrics. As a result of comparing the judgment results of the Java language project and the python language project, it can be seen that the features with high importance of the Java project tend to be of high importance also in Python.

Also, if the difference between each commit pair is significant, the source code metric in the decision model tends to contribute more. However, if the difference between the modifications of the commit pair is small, the contribution of the source code metric tends to decrease. From this result, it is considered that the importance in the judgment model may be higher when the source code metrics are extracted only in the part where the conflict occurs, not in the whole source file.

Since different languages have different grammar, it is necessary to extract source code metrics for each language, but there is also a lot of language-dependent information that remains hidden. Therefore, it is possible that there is room for further improvement in accuracy by increasing the kinds of source code metrics in each language in the future. When a developer encounters a merge conflict, the judgement model can be useful as a reference to determine how to resolve it.

## Acknowledgement

First of all, I would like to express my sincere gratitude to my supervisor Professor Katsuro Inoue (Graduate School of Information Science and Technology, Osaka University Department of Computer Science) for giving me the opportunity to study and work at Inoue Laboratory. He always gave me a lot of valuable advice and insightful comments.

My sincere thanks go to Associate Professor Makoto Matsushita (Graduate School of Information Science and Technology, Osaka University Department of Computer Science). He always gave me appropriate comments and an opportunity to review the validity of my research.

I would like to thank Assistant Professor Tetsuya Kanda (Graduate School of Information Science and Technology, Osaka University Department of Computer Science). He provided me with a lot of insightful comments for implementing my proposed method, so I could conduct my research smoothly. Without his help, it would not have been possible for me to complete my thesis.

I am grateful to Kazumasa Shimari(Graduate School of Information Science and Technology, Osaka University Department of Computer Science). He gave me a lot of valuable comments and suggestions on my research and paper writing.

I am grateful to Secretary Mizuho Karube (Graduate School of Information Science and Technology, Osaka University Department of Computer Science). She is always so nice and kind, making the lab such a cozy atmosphere to study.

I would like to thank all the members in Inoue laboratory for creating such a wonderful environment for studying and researching.

Last but not least, I would like to thank my family members for supporting me spiritually throughout writing this thesis and my life in general.

## References

[1] Yuriy Brun, Reid Holmes, Michael D. Ernst, and David Notkin. Early detection of collaboration conflicts and risks. In *IEEE Transactions on Software Engineering*, Vol.39, No.10, pp. 1358–1374, 2013.

[2] Bakhtiar Khan Kasi and Anita Sarma. Cassandra: Proactive conflict minimization through optimized task scheduling. In *International Conference on Software Engineering (ICSE)*, pp. 732–741, 2017.

[3] Klissiomara Dias, Paulo Borb, and Marcos Barretoa. Understanding predictive factors for merge conflicts. In *Understanding predictive factors for merge conflictsInformation and Software Technology*, Vol. 121, No.106256, 2020.

[4] Shuya Shiraki. 開発履歴のメタ情報を用いたマージコンフリクト解消支援手法. In *Master Thesis, Osaka University*, 2020.

[5] Iftekhar Ahmed, Caius Brindescu, Umme Ayda Mannan, Carlos Jensen, and Anita Sarma. An empirical examination of the relationship between code smells and merge conflicts. In *In International Symposium on Empirical Software Engineering and Measurement(ESEM)*, pp. 58–67, 2017.

[6] Filippo Lanubile, Christof Ebert, Rafael Prikladnicki, and Aurora Vizcaíno. Collaboration tools for global software engineering. *IEEE Software*, Vol. 27, No. 2, pp. 52–55, 2010.

[7] Yuriy Brun, Reid Holmes, Michael D. Ernst, and David Notkin. Proactive detection of collaboration conflicts. In *European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pp. 168–178, 2011.

[8] Ryohei Yuzuki. 開発履歴を用いたメソッドコンフリクトの分析. In *Master Thesis, Nara Institute of Science and Technology*, 2015.

[9] Mehran Mahmoudi, Sarah Nadi, and Nikolaos Tsantalis. Are refactorings to blame? an empirical study of refactorings in merge conflicts. In *International Conference on Software Analysis, Evolution and Reengineering(SANER)*, pp. 151–162, 2019.

[10] Paulo Meirelles, Carlos Santos Jr., João Miranda, Fabio Kon, Antonio Terceiro, and Christina Chavez. A study of the relationships between source code metrics and attractiveness in free software projects. In *Brazilian Symposium on Software Engineering*, 2010.

[11] Catarina Costa, José Menezes, Bruno Trindade, and Rodrigo Santos. Factors that affect merge conflicts: A software developers ’ perspective. In *Brazilian Symposium on Software Engineering*, pp. 233–242, 2021.