Vol.2022-SE-210 No.21
2022/3/11

Investigating the impact of source code metrics on merge conflict resolution judgement model

Mohan Bian[1,a)] Tetsuya Kanda[1,b)] Kazumasa Shimari[1,c)] Katsuro Inoue[1,d)]

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 create a model that suggest how to resolve appropriate merge conflicts with higher accuracy.

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. Version control systems enable us to access the revision history of each branch, which make it much easier for parallel developing. While projects can be efficiently developed by multiple people, parallel development may cause problems. After editing a newly created branch from the mainstream, if the mainstream also edits the same piece of code, 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]. Shiraki et al. [4] proposed a machine learning model 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 measured 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 machine learning model proposed by Shiraki et al., we aim to create a model that suggests how to resolve merge conflicts with higher accuracy. In addition, we investigate the importance of source code metrics contributes to the machine learning model and compare the importance of features in case of Java and Python projects.

In this paper, Section 2 describes the background of the

1 Osaka University
a) hen-bk@ist.osaka-u.ac.jp
b) t-kanda@ist.osaka-u.ac.jp
c) k-simari@ist.osaka-u.ac.jp
d) inoue@ist.osaka-u.ac.jp

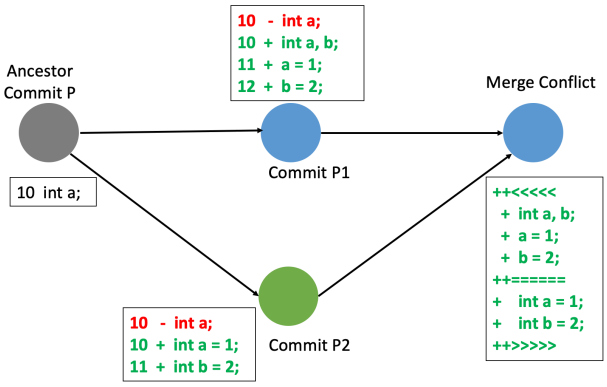

Fig. 1 An example of merge conflict.

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

2. Background

2.1 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 different edits in the same piece of code are detected in both commits, a merge conflict will occur.

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. Next, another branch is created from commit P, and commit P2 is created with another compilation for the line 10. 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, so that a merge conflict occurs when merging those two branches systematically.

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.

2.2 Solution of Merge Conflict

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. A study by Yuzuki reveals that about 98% of all merge conflicts that occur within methods in Java projects are resolved in this way [8].

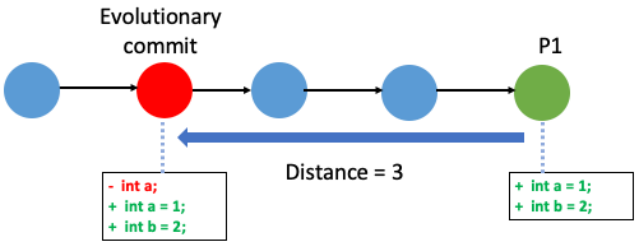

Fig. 2 Evolutionary commit and the distance.

2.3 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. 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 2 is the distance explained above.

2.4 Source Code Metrics

Source code metrics are measurements of various aspects of software code. Some metrics are at an higher 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 valuable information, 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 conflict.

2.5 ANTLR

ANTLR[*1] is a lexer and parser generator aimed at building and walking parse trees. ANTLR 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 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

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

Fig. 3 Flow of the input and output of Lexer.

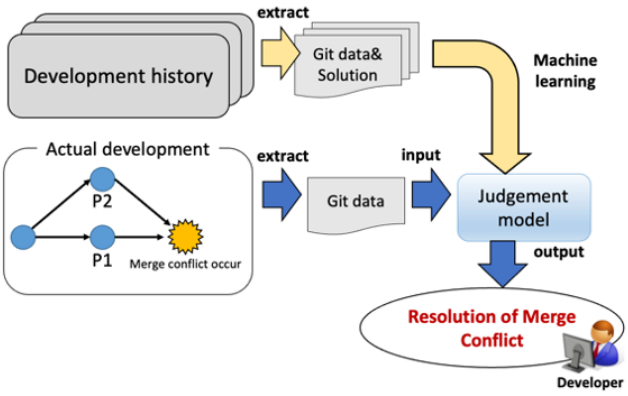

Fig. 4 Merge conflict resolution model proposed in previous research*1.

of three parts: Lexer, Parser, and Listener. In the experiment we only used Lexer, Figure 3 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 3). 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 source code 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.

3. Previous research

In this research, we extend the model created in previous research. In this section, we will explain the previous research by Shiraki et al and their 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 et al. proposed a method for resolving merge conflicts using machine learning. They created a judgement model (shown in Figure 4) 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

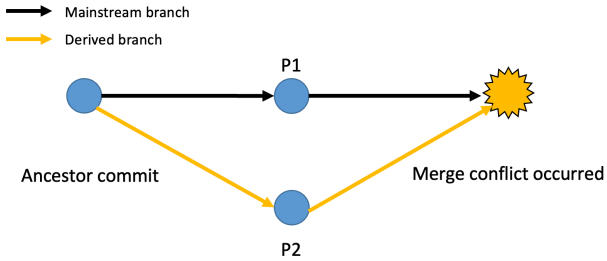

Fig. 5 Distinguishing Commit Pairs with Merge Conflicts*1.

Table 1 Definition of merge conflict resolution instruction in Shiraki et al.'s study.

ADOPT	Adopt all edits
DELETE	Delete all edits
EDIT	Make edits
ZERO	0 lines to edit

Table 2 Parameters used to build the model.

	Parameter
P1	linenum(P1) time(P1) author ratio(P1) distance(P1)
P2	linenum(P2) time(P2) author ratio(P2) distance(P2)
Difference	linenum(d) time(d) author ratio(d) distance(d)

is used as the learning algorithm.

3.2 Data Collection

In the previous research, 20 Java projects were collected from the OSS provided by Apache. The determination of the resolution method by the judgement model is performed for each commit pair in which a merge conflict has occurred. As shown in Figure 5, 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 instructions for each commit (P1 resolution instruction / P2 resolution instruction). Table 1 defines the list of resolution methods for each commit.

Development history information was acquired from the repository for each commit P1 and P2 in which a merge conflict occurred. Table 2 is the list of parameters used to create the machine learning model. `linenum` is the number of lines where merge conflicts occur, `time` is commit creation date and time, `author ratio` is the ratio of commit creator to total commit number, `distance` is the distance between current commit and its Evolutionary Commit. P1 is the parameter from mainstream, P2 is the parameter from branch, and difference is the subtraction of P1 and P2.

*1 Shiraki et al.,Judgment Model of Merge Conflict Resolution Pattern Using Machine Learning Meta-Information,IEICE Technical Report,2020

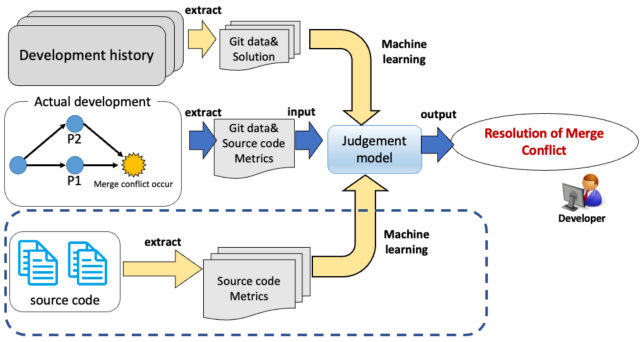

Fig. 6 An overview of the model proposed in this research.

3.3 Experiment result

20 Java projects collected from the OSS provided by Apache were used for the experiment. The accuracy is a result of cross-validation by dividing the merged conflict data into five for each project. The accuracy 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 classification in the model, and the sum of the importance of all parameters is 1. The number of lines line_num (P1) and line_num (P2) where merge conflicts occur are both as large as 0.15 or more. 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.

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 6.

Shiraki et al.'s method [4] used only language-independent features. However, as introduced in Section 2, source code metrics also possibly have an effect on software quality. Therefore, we add source code metrics to this model to see how the accuracy changes. We also aim to verify whether a judgement 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 judgement 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 3 Source code metrics to get.

P1	Name_Num(P1) Operator_Num(P1) Keyword_Num(P1)
P2	Name_Num(P2) Operator_Num(P2) Keyword_Num(P2)
Difference	Name_Num(d) Operator_Num(d) Keyword_Num(d)

There are many source code metrics show the complexity of the program. For example, 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. Considering the cost of extracting the source code metrics, we decided to use source code metrics that can be easily get. Metrics that represent the amount of information 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 3 shows the new metrics to get for the judgement model. `Name_Num` represents the number of variables and functions, `Operator_Num` is the number of operators and `Keyword_Num` is the number of keywords. Same with previous research, P1 is the parameter from mainstream, P2 is the parameter from branch, and diff is the subtraction of P1 and P2.

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

First of all, we 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. We used ANTLR introduced in Section 2.5 to generate a Lexer from Java and Python grammar file respectively. Then we 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 7 describes the flow of source code metrics extraction.

According to the grammar file, thirty-four kinds of op-

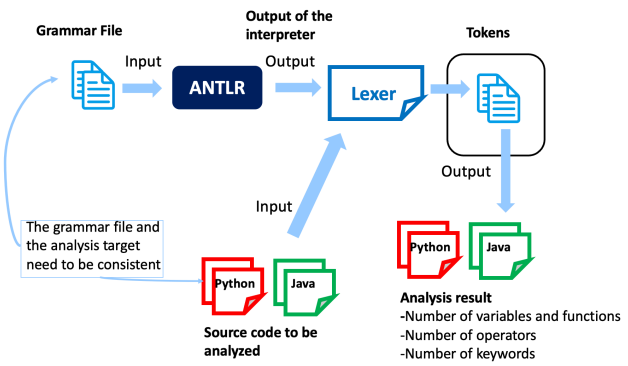

Fig. 7 The flow of source code metrics extraction.

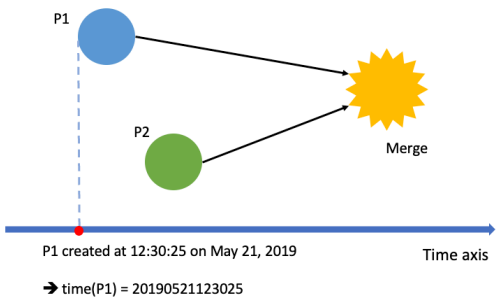

Fig. 8 The time feature used in previous research.

erator 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.

Finally, we implemented a program to count 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.

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 8).

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, we 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 each commit's creation time to the merge time. `Fixed_time1`, `Fixed_time2`, as well as `Fixed_timeD`(subtraction of `Fixed_time1`, `Fixed_time2`) which are updated as features for the new model.

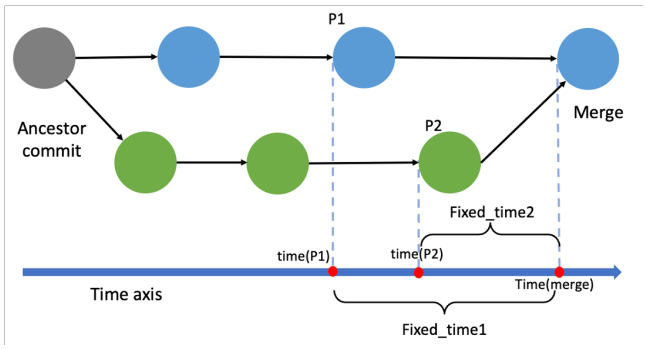

Fig. 9 The new time feature `Fixed_time1`, `Fixed_time2` used in this research.

5. Evaluation

In order to investigate the effectiveness of the source code metrics for the judgement model, we conducted evaluation experiments on Java projects and Python projects with and without the source code metrics. Same with Shiraki et al.'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 4 shows the list of projects used in the experiment and the number of merge conflict.

20 java projects and 7 python projects were used in the experiment. For each project, the accuracy 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 et al.'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.

5.1 RQ1: How does the accuracy of the model change if both developing history data and language-specific source code metrics are used in the model?

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

Figure 10 shows the ratio of source code metrics(the grey area pulled out by the line) to all features. The blue zone represents the parameter importance of mainstream, the red zone represents that of branch, the green zone represents that of the subtraction of mainstream and brunch. Source

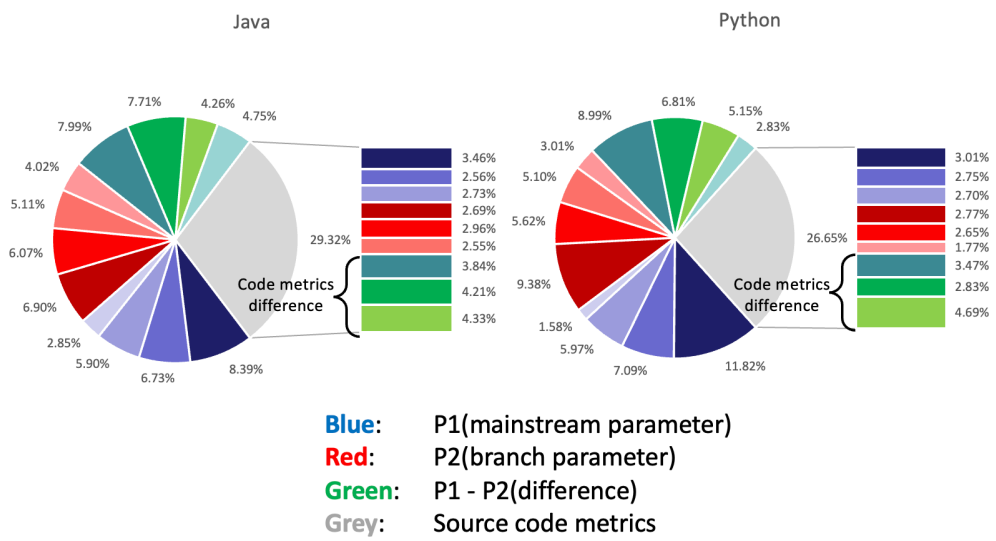

Fig. 10 Importance of source code metrics in case of Python and Java.

Table 4 List of Java and Python projects used in the experiment.

Java project(20 in total)	
project name	# merge conflict
beam	1,449
camel	44
cassandra	17,837
cordova-android	973
curator	287
dubbo	401
flink	3,454
geode	573
groovy	353
hbase	187
hive	3,246
ignite	2,850
incubator-heron	61
jmeter	693
lucene-solr	2,562
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	2,082
predictionio	14

code metrics (9 in total) are 29.32% for Java and 26.65% for python . Among all the 9 source code metrics, `d_NameNum`, `d_OperatorNum`, and `d_KeywordNum` came to the top in both Java and Python projects. These three metrics are all the difference of code metrics for each commit pair. 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 judgement model than the metrics extracted from each commit.

Since the difference of code metrics from two commits 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 et al.'s model. Table 5 shows the parameter and the accuracy of each model.

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 6 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.

When we 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 5 Parameters used in each model.

parameter	improved Shiraki et al.'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)	✓	✓	✓
Fixed_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%

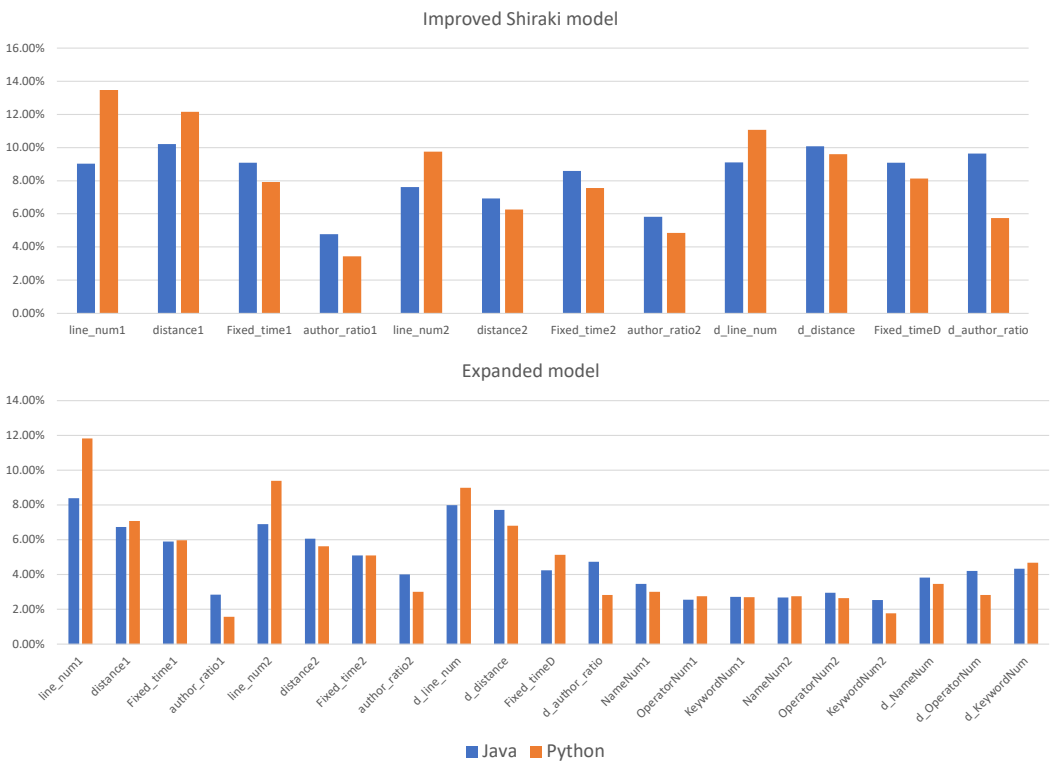

Fig. 11 Feature importance in case of Java and Python.

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

There are 21 features in total, including 12 metrics acquired from the development history and 9 extra source code metrics extracted from the source code this time. In Figure 11, 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. 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 differences in the 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 the result, the differ-

Table 6 Importance of source code metrics (CM) in each Java project.

project name	expanded model	CM importance
beam	95.91%	31.54%
camel	83.33%	37.36%
cassandra	66.58%	35.04%
cordova-android	71.43%	21.55%
curator	66.18%	37.48%
dubbo	61.70%	17.23%
flink	76.26%	29.78%
geode	72.41%	24.47%
groovy	89.55%	35.71%
hbase	60.98%	11.49%
hive	70.90%	23.62%
ignite	81.23%	27.52%
incubator-heron	35.71%	28.29%
jmeter	98.50%	55.37%
lucene-solr	73.87%	19.04%
mahout	91.11%	23.14%
maven	84.78%	36.64%
nifi	66.67%	29.29%
nutch	81.38%	33.78%
rocketmq	82.35%	28.10%
average	75.54%	29.32%

ence between the source code metrics extracted from commit pairs contributed more to the judgement 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. In addition, 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, it will be necessary to conduct research not only on Apache but also on various projects in various languages.

Moreover, 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. 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 accuracy 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 judgement model based on the source code metrics. As a result of comparing the judgement 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 metrics in the decision model tend to contribute more. However, if the difference between the modifications of the commit pair is small, the contribution of the source code metrics tend to decrease. From this result, it is considered that the importance in the judgement 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.

Acknowledgment

This work was supported by JSPS KAKENHI Grant Numbers JP18H04094 and JP19K20239.

References

[1] Brun, Y., Holmes, R., Ernst, M. D. and Notkin, D.: Early Detection of Collaboration Conflicts and Risks, *IEEE Transactions on Software Engineering*, Vol.39, No.10, pp. 1358–1374 (2013).

[2] Kasi, B. K. and Sarma, A.: Cassandra: Proactive Conflict Minimization through Optimized Task Scheduling, *International Conference on Software Engineering (ICSE)*, pp. 732–741 (2017).

[3] Dias, K., Borb, P. and Barretoa, M.: Understanding predictive factors for merge conflicts, *Understanding predictive factors for merge conflictsInformation and Software Technology*, Vol. 121, No.106256 (2020).

[4] Shiraki, S., Kanda, T. and Inoue, K.: Judgment Model of Merge Conflict Resolution Pattern Using Machine Learning Meta-Information, *IEICE Technical Report*, Vol. 119, No. 451, pp. 61–66 (2020).

[5] Ahmed, I., Brindescu, C., Mannan, U. A., Jensen, C. and Sarma, A.: An Empirical Examination of the Relationship Between Code Smells and Merge Conflicts, *In International Symposium on Empirical Software Engineering and Measurement(ESEM)*, pp. 58–67 (2017).

[6] Lanubile, F., Ebert, C., Prikladnicki, R. and Vizcaíno, A.: Collaboration Tools for Global Software Engineering, *IEEE Software*, Vol. 27, No. 2, pp. 52–55 (2010).

[7] Brun, Y., Holmes, R., Ernst, M. D. and Notkin., D.: Proactive Detection of Collaboration Conflicts, *European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pp. 168–178 (2011).

[8] Yuzuki, R., Hata, H. and Matsumoto, K.: How we resolve conflict: an empirical study of method-level conflict resolution, *2015 IEEE 1st International Workshop on Software Analytics (SWAN)*, pp. 21–24 (2015).

[9] Mahmoudi, M., Nadi, S. and Tsantalis, N.: Are Refactorings to Blame? An Empirical Study of Refactorings in Merge Conflicts, *International Conference on Software Analysis, Evolution and Reengineering(SANER)*, pp. 151–162 (2019).

[10] Meirelles, P., Jr., C. S., Miranda, J., Kon, F., Terceiro, A. and Chavez, C.: A Study of the Relationships between Source Code Metrics and Attractiveness in Free Software Projects, *Brazilian Symposium on Software Engineering* (2010).

[11] Costa, C., Menezes, J., Trindade, B. and Santos, R.: Factors that Affect Merge Conflicts: A Software Developers' Perspective, *Brazilian Symposium on Software Engineering*, pp. 233–242 (2021).