

PHILIPPS UNIVERSITY OF MARBURG

DOCTORAL THESIS

Improving the Accuracy of Refactoring Detection

Author:

Tan Liang
Xi'an of China

Supervisor:

Prof. Christoph Bockisch
Prof. Jianjun Zhao

Programming languages and tools
Mathematics and Computer Science

January 28, 2023

Declaration of Authorship

I, Tan Liang, declare that this thesis titled, "Improving the Accuracy of Refactoring Detection" and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed:

Date:

PHILIPPS UNIVERSITY OF MARBURG

Abstract

Mathematics and Computer Science

Improving the Accuracy of Refactoring Detection

by Tan Liang

With the development of refactoring technology, refactoring detection technology as its reverse technology has also been greatly progressed and applied, the technology has important significance and role for code optimisation, code review and code reliability. Over the past 20 years, the refactoring detection technique has evolved from a theoretical concept to be mature approaches and tools. However, due to the various complexities that arise when refactoring code, there are still some problems with these detection tools at work: selection of tools, detection of nested refactorings, false negatives due to matching algorithms, etc. As the requirements for detection increase, the pursuit of better detection performance (precision and recall) and more generalised detection tools has become a major research goal for my PhD.

The main research components of this paper include:

Firstly, at the beginning of my research I conducted a meta-analysis of refactoring detection and evaluated the detection performance of four common refactoring detection tools under the same benchmark, analysed and compared their strengths and weaknesses, and identified new research questions and research directions.

Secondly, I identified the study and detection of nested refactorings as a research blind spot in existing approaches, so I conducted a demonstration of the feasibility of nesting multiple refactor types with each other; in addition, I created an approach that can detect nested refactorings based on a single refactoring data using manually defined refactoring features combined with a random forest algorithm, thus being able to detect all 35 semantically meaningful nestings of them with 91.4% accuracy.

Then I focused on the features that emerged during the refactoring process, mining the refactoring information in the diff to help RefDiff improve detection performance. During the research I developed Diff Extractor and Diff Encoder for extracting and encoding diffs, transformed diffs into arrays for refactoring information mining, and trained two models: 1. Diff Structure Feature Model, which determines the type of refactoring based on the structural features of the refactored diffs and can be used as a result checker, which improves the overall performance of RefDiff by checking for false positives in the RefDiff detection results. 2. Diff Feature Matching Network, which is trained based on the correspondence of the removed and added parts of the refactored diff, has excellent robustness and can solve the problem of missing matching caused by word frequency matching approach.

Finally, I design an approach that integrates the two models, optimises Diff Extractor and Diff Encoder based on the characteristics of diff features, adds flags to diff based on the refactoring property, designs new encoding approach emphasising token uniqueness, trains better models and builds a model cross-validation mechanism that allows us to obtain detection results with high levels of confidence. We have shown that our approach, called *RefDiff-Model*, not only improves the precision of RefDiff 2.0 to 100% and increases the recall to 96.1%, but also continues to support detection tasks in multiple programming languages.

Keywords:Refactoring Detection, Nested Refactoring, Tools, Model, Machine Learning.

Acknowledgements

This dissertation was completed under the careful guidance of my supervisor, Prof. Dr. Christoph Bockisch, who has been a great help to me in my work and life over the past time, for which I would like to express my deepest gratitude. Over the past five years, the professor has taught me how I should go about my research work, and has taught me how to choose a topic, understand it, and explore new research points so that I can explore new research ideas. The professor's rigorous, meticulous and factual approach to research, as well as his dedication to his work and kindness to others, have left a positive impact on me.

From the design to the end of this project, the professor has helped me not only in terms of academic but also cultural differences. There are some differences between my country and Germany in terms of expressions and ways of thinking, which led to problems in our communication at times. The professor was very patient in exchanging ideas with me and worked hard to bridge this difference, making my English expressions more authentic and playing an important role in the polishing of my papers.

I remember many times the professor revised my papers late at night, and despite being very busy, he would still email me during his work trip to tell me what needed to be improved and enhanced in my papers, which left a deep impression on me. The weekly meetings were indispensable, even during my home working period, and his continued advancement of my research was a major reason why I was able to complete my PhD on time.

I will soon be completing my PhD and would like to thank Pro. Dr. Christoph Bockisch for his guidance and help, as well as my other colleagues for their help.

I would like to thank my family for the support and encouragement I have received.

Thank you!

Contents

Declaration of Authorship	iii
Acknowledgements	ix
1 Summary	1
1.1 State of the Art in Detecting Refactoring	2
1.2 Problem Statement	4
1.2.1 Meta-Analysis and Unified Benchmark Evaluation	5
1.2.2 Detection of Nested Refactoring	6
1.2.3 Matching Problem	7
1.3 Solution Approach and Results	8
1.3.1 Result of Meta-Analysis	8
1.3.2 Detection Approach of Nested Refactoring	10
1.3.3 Solution Approach of Matching Problem	11
1.4 Conclusion	12
1.5 Future Work	13
1.6 Chapter Arrangement	14
2 A Survey of Refactoring Detection Tools	17
2.1 Purpose of Survey	17
2.2 Refactoring detection tool	18
2.2.1 RefactoringCrawler	18
Theory	18
Discussion	20
Conclusion	21
2.2.2 Ref-Finder	21
Theory	21
Discussion	23
Conclusion	23
2.2.3 RefDiff	23
Theory	23
Discussion	25
Conclusion	25
2.2.4 RefactoringMiner	25
Theory	26
Discussion	27
Conclusion	28
2.3 Experimental Comparison of Tools for Detecting Refactorings	29
2.3.1 Experimental results	30
Accuracy	30

	Performance	34
2.3.2	Comparing influence of repository structure on RM	35
2.4	Threats to Validity	36
2.4.1	External Validity	36
2.4.2	Internal Validity	36
2.4.3	Discussion	37
2.4.4	Detection of <i>Move Class</i> and <i>Rename Package</i>	38
2.4.5	Distinguishing <i>Move Class</i> and <i>Rename Package</i>	39
2.5	How to choose a refactoring tool	40
2.6	Conclusion	42
3	Probability Model for Nested Refactoring	45
3.1	Detecting Problems with Nested Refactoring	45
3.1.1	Nested Refactoring	45
3.1.2	Problem Solution	45
3.2	Probability modeling based on random forest	48
3.2.1	Theory of Probability Model	48
3.3	Algorithm	51
3.4	Proof of Concept	53
3.4.1	Training	53
3.4.2	Validation for Single Refactorings	54
3.5	Detecting Nested Refactoring	57
3.5.1	Extract Features and Calculation	58
3.5.2	Results and Analysis	59
3.5.3	Specific Strategy	59
3.6	Evaluation	60
3.6.1	Threats to Validity	60
	Internal Validity	60
	External Validity	61
3.7	Conclusion	63
4	Diff Extractor and Diff Encoder	65
4.1	The Role of Diff in Refactoring Detection	65
4.2	Analysis of Refactoring Diff	65
4.2.1	Classification of Refactoring Diff	65
4.2.2	Analysis of Refactoring Diff	68
4.3	Diff Extractor	69
4.3.1	Basic Algorithm of Diff Extractor	69
4.3.2	Implementation Algorithm of Diff Extractor	69
4.4	Diff Encoder	70
4.4.1	Related Encoding Approach	70
4.4.2	Jigsaw Hypothesis	73
4.4.3	Encoding Approach for Diff	73
	Code To Array	73
	Encoding Tokens	74
	Array To Image	78

5	Training Model Base on Diff	79
5.1	Solution Problem	79
5.1.1	Code similarity algorithm	79
5.1.2	Analysis	80
5.2	Diff Structure Feature Model	81
5.2.1	Approach Overview	81
5.2.2	Training Process	82
5.2.3	Model Evaluation	84
5.3	Evaluation	86
5.3.1	Approach Evaluation	86
	Performance of Result Checker	87
	Result Analysis	88
5.3.2	Threats to Validity	90
5.3.3	Challenges and limitations	90
5.4	Diff Feature Matching Network	91
5.4.1	Approach Overview	91
5.4.2	Training Process	92
	Data preparation	93
	Training Process	93
5.4.3	Model Evaluation	95
5.4.4	Approach Evaluation	97
	Deployment	97
	Performance of Diff Feature Matching Network	98
	Result Analysis	99
5.4.5	Threats to Validity	100
5.4.6	Challenges and Limitations	101
6	RefDiff-Model	103
6.1	Problem Analysis	103
6.1.1	Problems with existing models	103
	Problem with Diff Structure Feature Model	103
	Problem with Diff Feature Matching Network	104
6.1.2	Problems with Diff Tool	104
	Problem with Diff Extractor	104
	Problem with Diff Encoder	105
6.2	Approach Overview	105
6.2.1	Approach Workflow	105
6.2.2	Core Mechanism	106
6.3	Optimised Solutions	107
6.3.1	Optimised Diff Extractor	107
6.3.2	Optimised Diff Encoder	109
	Encoding Approach For Diff Features Network Network	109
	Encoding Approach For Diff Structure Features Model	110
6.4	Training	111
6.4.1	Optimised Diff Features Marching Network	111
	Model Training	111
	Model Evaluation	113

6.4.2	Optimised Diff Structure Features Model	115
	Model Training	115
	Model Evaluation	115
	Generalized Applications	117
6.5	Evaluation	118
6.5.1	Result Analysis	119
6.5.2	Threats to Validity	121
6.5.3	Challenges and Limitations	121
6.6	RefDiff-Model with Random Forest Model	121
6.6.1	Implementation	122
6.6.2	Train Abstract Features	123
6.6.3	Application in Nested Refactoring	125
6.6.4	Model Embedment	126
	Bibliography	129

List of Figures

1.1	An example of refactoring and detection	2
1.2	Nested Refactoring	6
1.3	Processing Node Information	8
2.1	Boxplot of Execution Time	35
2.2	Example diff for a <i>Move Class</i> refactoring.	40
2.3	Example diff for a <i>Rename Package</i> refactoring.	40
3.1	Normal Refactoring	46
3.2	Nested Refactoring	46
3.3	Diff	47
3.4	Algorithms Idea	48
3.5	Manual Extracting	49
3.6	Extracting by refactoring function	49
3.7	Example of Rename Package	54
3.8	Example of Extract Method	55
3.9	Traditional Idea of Refactoring Detection	56
3.10	New Idea of Refactoring Detection	56
3.11	Process	57
3.12	Variable relationship	61
4.1	Move statement Type	66
4.2	Rename Type	66
4.3	Remove Type	66
4.4	Add Type	67
4.5	Fixed transformation Type	67
4.6	Combine Type	68
4.7	Refactoring objects and Code elements	68
4.8	Prase and Match	70
4.9	Example of the ExtractIntersection algorithm	71
4.10	Putting the code into an array	74
4.11	Example of ExtractMethod	74
4.12	Example of encode	76
4.13	Encoding to arraying	76
4.14	Example of Move Type	76
4.15	Example of Rename Type	77
4.16	Example of C	77
4.17	Example of JavaScript	77
5.1	Approach Flow chart	82

5.2	Accuracy Chart	85
5.3	Loss Value Result	86
5.4	Accuracy distribution	87
5.5	Distribution of predicted values for test data	89
5.6	Abstract Illustration	91
5.7	Approach Flow chart	92
5.8	Structure of Diff feature Matching Network	94
5.9	Train Result	96
5.10	Testing Example of Comparison	96
5.11	Filter Layer	99
6.1	Approach Flow chart	105
6.2	Node information processing concept	106
6.3	Diff Change After Optimization	108
6.4	Example of uniqueness encoding	110
6.5	Optimisation Result	111
6.6	Matching Network	113
6.7	Matching Network	114
6.8	Matching Network	114
6.9	Accuracy Chart	116
6.10	Loss Chart	116
6.11	Diff Convert to Array	118
6.12	Process of Features Extraction	122
6.13	Model Fusion	127

List of Tables

2.1	RefactoringCrawler(1.0.0)	30
2.2	Ref-Finder(1.0.4)	30
2.3	RefactoringMiner(1.0.0)	30
2.4	RefDiff	31
2.5	Detection Results for Accuracy(supported refactoring types)	32
2.6	RefactoringMiner(1.0.0)	33
2.7	RefDiff	33
2.8	Execution Time Comparison	35
2.9	Detailed accuracy results for RefactoringMiner and the <i>Move Class</i> refactoring type.	38
2.10	Detailed accuracy results for RefactoringMiner and the <i>Rename Package</i> refactoring type.	39
3.1	Count statistics and Probability calculations	52
3.2	Probability distributions	53
3.3	Example	53
3.4	Feature summary.	55
3.5	The trained probability model.	56
3.6	Test result of a single refactoring	57
3.7	Test result of a nested refactoring	58
3.8	Framework of Programming Language	62
4.1	Element Operations Table	69
5.1	Data Distribution	83
5.2	Training Set and Validation Set	83
5.3	Parameters of Diff Feature Matching Network	84
5.4	Test Result	88
5.5	Data Distribution	93
5.6	Parameters of Diff Feature Matching Network	94
5.7	Testing Result	97
5.8	Test Result	99
5.9	Comparison in JavaScript	99
5.10	Comparison in C	100
6.1	Data Distribution	112
6.2	Evaluation of experimental results	117
6.3	Test Result	119
6.4	Comparison in C	119
6.5	Comparison in JavaScript	120
6.6	Abstract Feature Assignment	124

6.7	Feature Arrays Chart	124
6.8	Feature Arrays Chart	125
6.9	Nested Refactoring Probability	126

Chapter 1

Summary

Refactoring is a disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behavior.

Refactoring is usually motivated by noticing a code smell. The technique of refactoring a software system has been around for a long time already, however, the term was probably coined in 1989 by Bill Opdyke and Ralph Johnson. It has gained wide adoption especially in the agile software development community, marked by the popular book by Fowler et al. (Fowler, 1999). Refactoring is intended to improve the design, structure, and/or implementation of the software (its non-functional attributes), while preserving its functionality. Potential advantages of refactoring may include improved code readability and reduced complexity; these can improve the source code's maintainability and create a simpler, cleaner, or more expressive internal architecture or object model to improve extensibility. Another potential goal for refactoring is improved performance; software engineers face an ongoing challenge to write programs that perform faster or use less memory (Martin, 2009).

The turnover of teams implies missing or inaccurate knowledge of the current state of a system and about design decisions made by departing developers. Further code refactoring activities may require additional effort to regain this knowledge (Nassif Matthieu, 2017). Refactoring activities generate architectural modifications that deteriorate the structural architecture of a software system. Such deterioration affects architectural properties such as maintainability and comprehensibility which can lead to a complete re-development of software systems. (Van Gorp Jilles, 2002)

Code refactoring activities should have been recorded in the software activity logger when using tools and techniques providing data about algorithms and sequences of code execution (Hassan Ahmed E., 2010). Providing a comprehensible format for the inner-state of software system structure, data models, and intra-components dependencies is a critical element to form a high-level understanding and then refined views of what needs to be modified, and how. (Novais Renato, 2017) However, refactoring activities are rarely documented.

Refactoring detection is the analysis of refactoring activities that have occurred during the evolution of the software system. This analysis can be helpful for improving the software engineering process. For example, detected refactorings can help the programmer to understand the program code

again after being absent for a certain time. Refactorings detected in software archives can also be used to identify errors that have occurred during the software development. Furthermore, the refactoring information provides new insights how software projects evolve over time. Refactoring detection techniques are very relevant, they are used for empirically studying (Giuliano Antoniol, 2004; Danilo Silva, 2016; Gabriele Bavota, 2012; Gabriele Bavota, 2015; Miryung Kim, 2011; Fabio Palomba, 2017; Napol Rachatasumrit, 2012; Gustavo Soares, 2013) software evolution, and to support other software engineering tasks, such as library adaptation (Ittai Balaban, 2005; Danny Dig, 2006; Johannes Henkel, 2005; Zhenchang Xing, 2007), software merging (Danny Dig, 2008), code completion (Stephen R. Foster, 2012; Xi Ge, 2012), and code review (Everton L. G. Alves, 2014; Xi Ge, 2014; Xi Ge, 2017).

1.1 State of the Art in Detecting Refactoring

Reverse engineering can be used to detect the application of refactorings (Serge Demeyer, 2000). Current research focuses on refactoring detection algorithms that detect a (likely) set of refactorings that developers applied to the source code (Nikolaos Tsantalis, 2018). Figure 1.1 shows an example, namely applying and detecting the *Move Class* refactoring, which we will consider in more depth in the remainder of this dissertation.

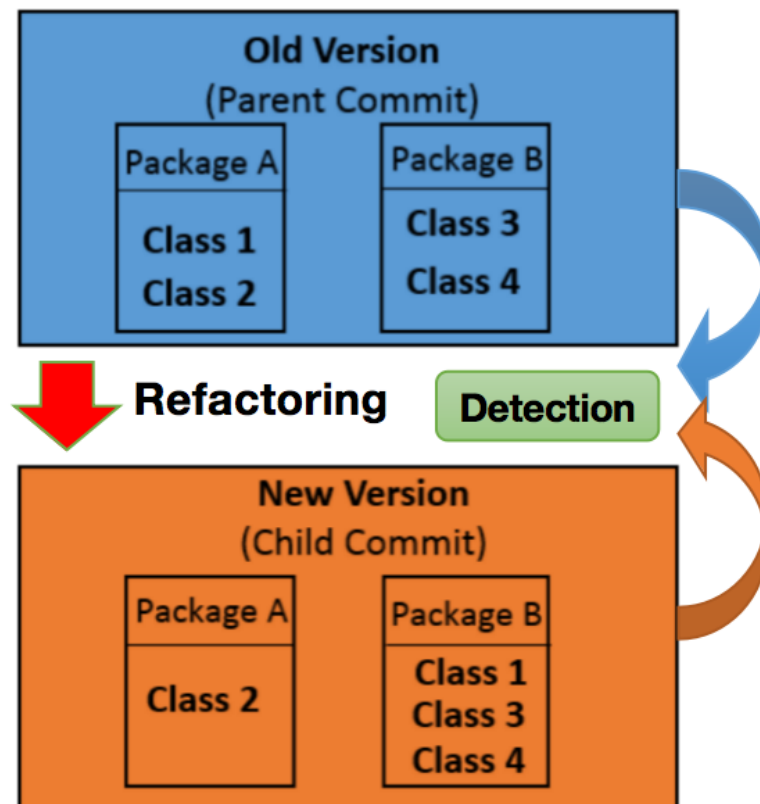


FIGURE 1.1: An example of refactoring and detection

The history of detection mechanisms for refactorings starts with theoretical discussions of ways to recognize a limited set of simple refactoring. Today we have automated tools, which are capable of recognizing several different and more complex refactorings.

In 2000, Demeyer et al. (Serge Demeyer, 2000) proposed reverse engineering of reconstructed code and proposed a detection heuristics. Instead of a refactoring-detection algorithm, a heuristic approach was used to measure the available versions of each software system and compare the results to determine the existence and nature of the refactoring. Another relevant development is the increasing availability of technologies, which enable the detection of refactorings. For example, Kim et al. (T. Kamiya, 2002) developed CCfinder, a new clone detection technique that consists of the transformation of input source text and a token-by-token comparison. Since then, methods for the detection of refactorings and related technologies have entered a stage of rapid development.

The first refactoring-detection approaches focused on specific refactorings that are relatively simple. Van Rysselberghe and Demeyer (F. Van Rysselberghe, 2003)—inspired by palaeontological fossil research—proposed the “software palaeontology” heuristic in 2003. This method paid special attention to the evaluation of the “move method” refactoring. Their work consisted of comparing different releases of existing source code and analyzing differences and reconstructing past evolution processes. In the same year, Malpohl et al. (Miryung Kim, 2010) proposed the algorithm “Renaming detection”, which is equally applicable to data description languages such as XML. The detector works with multiple file pairs, also finding renamings that span several files. It is also part of a suite of intelligent tools for merging programs exploiting the semantics of programming languages.

After the first concrete approaches to detecting specific refactorings, more holistic methods have been developed in theory. Antoniol et al. (Giuliano Antoniol, 2004) and his group presented an approach, based on Vector Space cosine similarity of class identifiers, to automatically identify Class-level refactorings between two subsequent releases. The approach was useful to identify some replacement, merge and split during the evolution of *dnsjava*. This approach uses a calibration threshold to influence the precision and recall of the detection method. An object-oriented design structure difference algorithm (“UMLDiff”) was proposed by Xing and Stroulia (Zhenchang Xing, 2005) in 2005. UMLDiff can detect additions, removals, moves, renamings of packages, classes, interfaces, fields methods, changes of attributes and changes of the dependencies among these entities. Based on the UMLDiff algorithm, JDEvAn (Java Design Evolution Analysis) can automatically detect the design changes between two models corresponding to two versions of a system. Comparing the software versions at the design level makes it have a direct impact on the evolutionary development process of the software. Domain-specific semantics make the comparison results more intuitive than other structured differentiation algorithms. Early researchers used visualisation approaches to describe code changes that required the subjective involvement of users. With the advent of some excellent algorithms, the design

of algorithmic automation based on these algorithms have made it possible to rely less on subjective interpretation than visualisation approaches and can instead provide the basis for subsequent analysis. The detection approaches and algorithms proposed by some early researchers laid a solid theoretical foundation for later refactoring detection tools. Some researchers used the above algorithms and ideas to develop complete refactoring detection systems, i.e., tools that automatically detect the application of different types of refactorings in the code. These systems will be the subject to the study presented later in this dissertation.

Automatic refactoring detection tools. The earliest detection tool developed was RefactoringCrawler by Danny Dig et al. (Danny Dig, 2006) in 2006. This tool is implemented as a plugin for Eclipse and can detect seven types of refactorings in Java components, focusing on rename and move refactoring. In 2010, the tool Ref-Finder, developed by Kyle Prete and his team, was proposed (Kyle Prete, 2010). This tool is based on a Program the tool LSdiff (Logical Structural Diff) released by Kim et al. (M. Kim, 2009) to compute the delta between two versions of the source code. Prete and his team used a logic meta-programming approach to identify complex refactorings from two program versions and claimed that the tool can support the 63 refactoring types. RefactoringMiner was proposed in 2013 by Tsantalis (N. Tsantalis, 2013; Nikolaos Tsantalis, 2020) and his research group. It implements a lightweight version of the UMLDiff algorithm for computing the differences between object-oriented models independent on the IDE. It can detect ten kinds of refactorings. In 2017, RefDiff was created by Danilo Silva and Marco Tulio Valente (Danilo Silva, 2017; Danilo Silva, 2020), an automated approach that identifies 13 different refactoring types by inspecting two code revisions in a git repository. The key feature of this tool is the use of a similarity index.

1.2 Problem Statement

Each automated refactoring detection tool, in its own test data, indicates a good performance, so in order to better understand the detection capabilities of existing automated refactoring detection tools, they need to be evaluated using test data from a unified benchmark. Furthermore, the study of existing refactoring detection principles helped us to identify the research directions presented in this thesis.

Currently, the best refactoring detection tools are RefactoringMiner (Nikolaos Tsantalis, 2020) and RefDiff (Danilo Silva, 2020), both of which have reached version 2.0 after continuous development, and their performance is excellent. But the comprehensive performance of both tools has some shortcomings. Both tools are designed along similar lines, first parsing the code to be inspected into an abstract syntax tree of nodes, both of which focus only on nodes related to diffs. The syntax-tree nodes representing the code in the diff before and after the change are then matched to form refactoring

candidates using a matching algorithm. Finally, the candidates are assigned refactoring types based on the characteristics of each refactoring.

However, their studies have all been on the detection of single refactoring type, whereas in reality developers often apply multiple refactorings to the same code element at the same time, but only few refactoring combinations have received attention, as mentioned in the RefactoringMiner paper.

RefactoringMiner 2.0 is a programming-language-dependent detection tool with a precision of 99.7 % and a recall of 94.2 %. It achieves this for two main reasons: 1. it is based entirely on an abstract syntax tree, which gives access to all syntax information. 2. its matching algorithm consists of statement matching and syntax-aware replacements. RefactoringMiner supports most of the well-known refactoring types. Its biggest limitation is that it can only detect java refactorings.

RefDiff 2.0 is a programming-language-independent detection tool with better generality. RefDiff has a precision of 96.4 % and a recall of 80.4 %. Although its performance is weaker than RefactoringMiner, it can cover a wide range of programming languages, which is a significant advantage, and it still supports the most common refactoring types. This is due to two things: 1. It is based on an abstract syntax tree that focuses on coarse-grained code structure elements and can be referred to as Code Structure Tree (CST). 2. Its matching algorithm relies heavily on text similarity of node code, a design that expresses similarity using the word frequency in the node code. These two advantages allow RefDiff to support multiple programming languages under one matching algorithm, as it does not need to take into account syntactic differences between programming languages. On the downside, this design sacrifices some of the detection performance, because the algorithm for computing code similarity fails to match the correct refactoring candidate in some cases. In particular RefDiff's recall needs to be improved.

RefDiff, as a tool for detecting refactoring in multiple programming languages, has excellent detection generality, which fits well with the trend towards generality detection tools. Therefore, we have made RefDiff to be the main object of research in the PhD phase. In summary, we have three goals:

1. Meta-analysing detection tools and testing them in a common benchmark;
2. Implementing detection of nested refactoring;
3. Developing an approach and tool to reach higher precision/recall without being language-specific by improving RefDiff.

1.2.1 Meta-Analysis and Unified Benchmark Evaluation

In meta-analysis and evaluation, we did study introducing four detection refactoring techniques, and comparing the precision, recall, execution time, verification of some previous researchers' conclusions, which revealed many new problems. All in all, we found the following problems: 1. Calibrating matching thresholds play an important role, but they are considered to be

unstable factors affecting accuracy in some tools. 2. Each tool has a very high precision and recall, but in some comparison experiments there was a large gap between the experimental results and the published values.

In the process of studying refactoring detection approaches, we identified two research questions:

1. Are approaches, which rely on a calibrated detection threshold (Ref-Finder, RefactoringCrawler, RefDiff), superior to methods that do not require such calibration (RefactoringMiner)?
2. Can the experimental results of the authors be repeated by a different researcher? In the case of Ref-Finder, the results reported by different authors are contradictory - which ones are the most likely to be representative?

The main purpose of this research can be to understand the advantages and disadvantages of the existing detection approaches, as well as the principles, and to find the direction and objectives of the next phase of research.

1.2.2 Detection of Nested Refactoring

Nested refactoring, also known as *compound refactoring*, is a derivative subject of refactoring development. After the first refactoring, the refactored program elements (such as method, field, or class) are directly refactored for the second time, and then committed as one code change to Git, such as *Extract Method + Move Method*, as shown in Figure 1.2. As far as Martin Fowler's seminal work (Fowler, 1999) is concerned, *single refactoring* is a general refactoring.

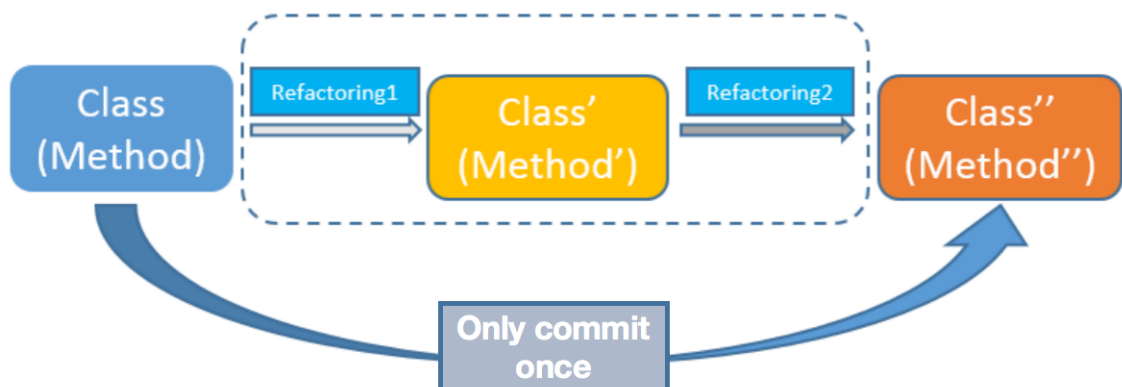


FIGURE 1.2: Nested Refactoring

Through the research of RefactoringMiner, Ref-Diff and other approaches, we found that all current refactoring detection approaches basically consist of two main parts (S. Kusumoto T. Kamiya, 2002): 1. Find refactoring candidates; 2. Detect refactoring types. In the second part, the detection tool sets very strict judgment conditions for each refactoring type. This is the key reason for the high-precision detection result and why the nested refactoring

is difficult to detect. Because researchers need to define judgment conditions for all possible nested refactoring types, however, the refactoring catalog maintained by Martin Fowler¹ currently lists 91 single refactoring types, and supporting nesting combinations would lead to a combinatorial explosion, having to define a huge number of judgement conditions.

The emergence of nested refactoring brings two new research questions to the research of refactoring detection:

- The feasibility study of nesting refactoring. Not all refactoring types can be nested with each other, regardless of the actual meaning and the legitimacy of semantics, the study of nested refactoring is a study of the mutual composability of refactoring types. Not only that, the order of nesting is also very important, the two refactoring types change in their nesting order, will affect the results of nesting, there will also be unable to be nested or illegal nested caused by changing the order. Thereby, the order of applying two or more refactorings also has to be taken into account.
- Nested refactoring is an important test of existing refactoring detection tools. Because the definition and judgment rules given by traditional detection tools for refactoring types are not applicable when detecting nested refactorings. Since the performance of nested refactoring is not simply superimposing these two types of refactoring on each other, but a new composite structure, researchers need to define new judgement rules for these nested refactoring types, which is a huge workload.

Summary The main objective of this study was to understand how refactorings are nested and to develop an approach to detect nested refactorings given some of the findings of the nesting process.

1.2.3 Matching Problem

RefDiff's *Similarity* expresses the degree of similarity of all codes contained in two nodes, based on Term Frequency-Inverse Document Frequency (TF-IDF) (G. Salton, 1986) (TF-IDF) and Jaccard coefficient (F. Chierichetti, 2010), which, however, in some cases is not suitable as a key condition for matching refactoring candidates, because they are designed for natural language texts. RefDiff's 80.4 % recall rate already indicates that word frequency similarity as an important condition for determining matches misses about one fifth of the refactoring candidates. In addition, changes in similarity threshold directly affect detection performance. The threshold is proportional to precision and inversely proportional to recall. There are two most important threats to word frequency similarity.

¹see: <https://refactoring.com/catalog/>

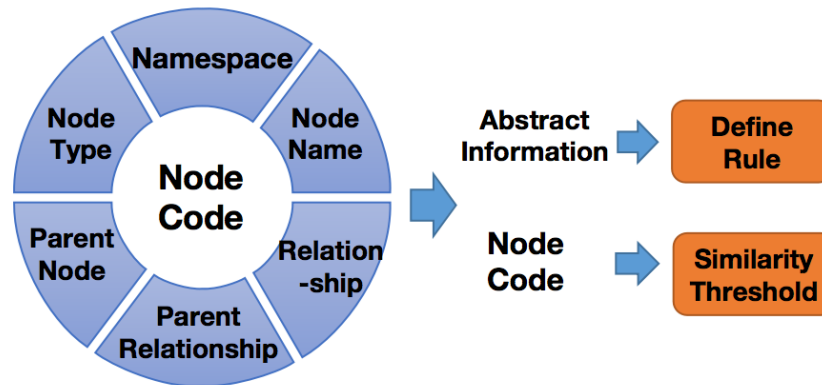


FIGURE 1.3: Processing Node Information

- Threat 1: Non-refactoring code changes. Refactoring is usually accompanied by non-refactoring code changes, where the node being refactored has code changes other than refactoring. RefDiff defines a similarity threshold of 0.5. However, if a refactored node contains a large number of non-refactoring code changes, the similarity of node code can easily be diluted below the threshold, leading to some false negative results.
- Threat 2: Noise nodes associated with diff. Non-refactoring changes exist not only in the nodes being refactored, but also in other nodes unrelated to refactoring. These noisy nodes also participate in matching, and if there are noisy nodes with similar code composition, then it increases the probability of false matches, which is the main reason why RefDiff will produce false positives.

Summary As shown in Figure 1.3, a node consists of six pieces of abstract information (the outer ring) and **NodeCode** (in the center). RefDiff uses the abstract information in the outer ring to define decision rules for each refactoring type and uses the similarity of the node code to determine whether the node matches, however it is clear that RefDiff's use of the information contained in the **NodeCode** is not sufficient.

Therefore, the **core problem** to be addressed in this part is: how to more fully mine the information contained in the nodes for better detection performance.

1.3 Solution Approach and Results

1.3.1 Result of Meta-Analysis

This dissertation studies and introduces four existing refactoring-detection techniques, and compares their precision, recall, execution time. In this way some previous researchers' conclusions are verified and new problems are identified. This study consists of 3 parts:

- The development history of refactoring detection and the algorithm structure of the main detection tools are briefly introduced. Through the study of these methods, we identified two research questions: Is there a kind of refactoring detection algorithm superior to others? Can the results of previous studies be reproduced and the expected performance of the tools be validated? We designed two sets of detection experiments for these questions. The experiments were conducted on four presence refactoring detection tools. Experiment 1: We chose the four refactoring types as a common benchmark of test data, 170 in total, to test their detection performance. Experiment 2: We selected the two best performing detection tools from Experiment 1, using 11 refactoring types as test data, a total of 350, for a more comprehensive and detailed evaluation.
- During our experiment, we found two main problems in previous experiments: on the one hand, the most modern tools have only been evaluated using very small code changes including only one or a few refactoring types; on the other hand, for larger code changes used in studies, a lot of actual refactorings were missing in the list of the expected results (i.e., the “gold standard”). For this reason we deliberately selected a more comprehensive data set, and we also improved the gold standard for comparison. For RefactoringMiner, the most advanced of the investigated tools, we compared two ways of operation: either there are just two versions to be analyzed before and after a series of refactorings has been applied; or there are multiple intermediate versions which only include a subset of the refactorings each. Finally, by analyzing the experimental results, we found some refactoring types for which detection needs to be optimized, and provide guidance for our next phase of research.
- At the end of the article we discuss how to choose a refactoring tool, based on the reader’s own needs, to detect the type of refactoring and the performance of the detection tool. Throughout the research, we found that *RefactoringMiner* is the most stable detection tool with higher precision and recall, covering the most common types of refactoring and various methods of connecting repositories. While our completed benchmark largely confirmed the previous results, in particular confirming that RefactoringMiner generally outperforms its competitors, we also identified a weak spot of RefactoringMiner that was not noted before: Refactorings of the type *Move Class* and *Rename Package* are frequently classified falsely. We also discussed the reasons for this wrong classification and outline a possible fix, which potentially boosts the overall precision and recall of RefactoringMiner to over 95%. *RefDiff* is the tool that spends the least amount of execution time. Although being defective for individual refactoring types, RefDiff maintains high precision and recall. But for the outdated tools *RefactoringCrawler* and *Ref-Finder*, there is a big gap between with the two new tools, whether

in precision, recall, and execution time, or in the coverage of common refactoring types.

1.3.2 Detection Approach of Nested Refactoring

In the first part of the study of the state-of-the-art, we found some problems that cannot be solved by the existing detection tools. This part of the research is mainly for the detection of nested refactoring. This phase is composed of three parts:

- Through research and analysis, we first clarified the reasons for the formation of nested refactoring. With the development of refactoring technology, the phenomenon of mutual nesting and compounding between refactoring types has appeared. This phenomenon is mainly due to two reasons: 1. Users frequently do not commit their changes to the version control system immediately after each refactoring. Instead, they perform multiple refactorings sometimes to the same part of the code before they commit, so that a nested refactoring occurs. The code after nesting refactoring cannot be detected by existing refactoring detection tools. 2. The composability between refactoring types is also an important reason for the nesting phenomenon. First of all, not all refactoring types can be nested with each other, and the nesting order between two refactoring types will also affect the results of nesting, so this is also the problem studied in this article.
- According to the analysis and research of the first part, the second part of my study investigates the possibility of nesting between refactoring types. In this part, based on ten common refactoring types, we conducted pairwise nesting experiments on them, and reached two important conclusions: 1. The two nested refactoring types need to conform to the syntax logic of the programming language, and some refactoring types have no connection points for nesting at all. 2. Nesting needs to have practical meaning. Some refactoring types can be nested, but it is not positive for improving the code structure. On the basis of these theories, we have also created some examples of nested refactorings and studied methods to detect them based on these examples.
- In this part, we have studied why the existing refactoring detection approaches cannot detect nested refactorings. Based on the research results, we have created a detection mechanism that can detect nested refactorings.

The existing traditional refactoring detection approaches will firstly find the refactoring candidates, and then compare them with the candidates one by one according to the established refactoring type rules, and finally determine the refactoring type. The refactoring type rules are formal definitions. To detect nested refactorings, you continue to need to define each nested refactoring type. However, with currently 91 java refactoring types described by Fowler, the number of types of nesting

refactorings is very large. Thus traditional detection methods of defining detection rules are not suitable for nested refactoring detection. We found that the diff of nested refactorings has some similarities with the two diffs of the corresponding single refactorings. Due to the nesting, the content of some diffs will be hidden, but the nested diff contains enough features to point to the two nested refactoring types. Therefore, we used statistical probability to detect nested refactorings based on features of single refactorings. We calculate the probability relationship between each feature and each refactoring type, and recognize diffs which show a very high probability for two refactoring types as applications of nested refactorings.

We have trained our approach for 10 common single refactoring types, resulting in the ability to detect all 35 semantically meaningful nestings of them with a precision of 91.4%, while RefactoringMiner only provides detection rules for three nested refactorings and other tools none at all. The precision of our approach for detecting single refactorings is at the same level as that of existing tools.

1.3.3 Solution Approach of Matching Problem

Our research in the second phase showed us that probabilistic, feature-based approaches are suitable for refactoring detection. Therefore, we continued to follow this direction by developing an approach based on machine learning algorithms for refactoring detection. Our main contributions at this stage are as follows:

- To demonstrate that machine learning can be applied to the diff between source code files, we analyzed and studied all existing refactoring types in the java programming language and invented an approach to encode the code text according to the characteristics of the programming language. The encoding uses numerical abstraction to represent the diff and lays the groundwork for mining the refactoring information contained in the diff using machine learning algorithms.
- Based on the encoded diff data and the structural features contained in the refactoring diff, we trained the Diff structure feature model to check the results reported by RefDiff 2.0 to improve its detection result precision to 99.7% and recall to 82.8% without changing the RefDiff threshold. When the candidate detection threshold is lowered, our result checker helps RefDiff to increase recall to 95.2% also at the small cost of a 0.2 percentage point loss, resulting in precision of 99.5%. RefDiff's algorithm is designed to work with a wide range of programming languages and our result checker is not syntax dependent, so our checker can also be used with other programming languages supported by RefDiff by replacing the feature model in the checker trained by diff and keyword assignment for other programming languages. Thus, the checker can support new languages or support more new types of refactorings by providing enough samples to train the diff feature model.

- The Diff structural feature model boosts the performance of RefDiff at the cost of significant checking time, as lowering the threshold generates a large number of false positive candidates that need to be checked by the checker one by one. Based on the principle that the added and removed parts of a refactored diff are almost identical, we trained a diff feature matching network for RefDiff 2.0 that solves the match missing problem caused by the word frequency similarity matching algorithm without changing the RefDiff design architecture. The matching network improves the overall detection performance of RefDiff to 98.6% precision and 93.2% recall, and works with all programming languages supported by RefDiff. The *Diff Feature Matching Network* is a neural network that features text relationships, text structure and the order of text expression to match the similarity of two pieces of text with high robustness. The network is suitable for texts with strict structural and syntactic rules, such as programming languages, and can be used to mine the structural features contained in the text, with far-reaching implications for the wider implementation of reconstruction detection in programming languages.
- Combining the role of the previous models, we have created an optimisation solution (**RefDiff-Model**) that uses deep learning algorithms for deep data mining of node codes. This solution is a good solution to the problem of missed matches and mismatches due to similarity thresholds, as well as the problem of single model dependencies. The solution is a RefDiff-based parsing matching algorithm that obtains the abstract information contained in the node and then combines it with the structural and corresponding information contained in the node's diff code. The solution uses this information to determine the refactoring type of the candidate, which can effectively circumvent the drawbacks of a single model, and the cross-validation mechanism guarantees 100% precision with a recall of 96.1%. The algorithm is not dependent on the syntax of a particular programming language and also benefits from the extensive generalisation of RefDiff. Not only this, but we have also incorporated the Random Forest algorithm, which gives the RefDiff-Model the ability to discover and detect nested refactorings.

1.4 Conclusion

During the PhD period, my research can be divided into three phases, and the research results of each phase have met the requirements of my initial plan. In detail, the results are as follows:

1. First, we have done a review study of the refactoring detection approaches that have emerged, including meta-analyses and uniform benchmark evaluation. The meta-analysis focuses on each of the four automated refactoring detection tools, analysing their technical routes and algorithmic ideas, summarising their strengths and weaknesses, and

comparing their algorithmic designs. In addition to this, we compare their detection performance using test data from three open source projects as a unified benchmark. In addition, this comparison is independent in that it has not been performed by the authors of the tools and thus offers the first comprehensive and objective comparison. In this way, the evaluation also revealed some problems which have not been discussed before, in particular the fact that some refactorings cannot be distinguished well by the tools and that sequential refactorings of the same code element, so-called nested refactorings, are not well supported.

2. Second, nested refactorings have been researched in depth. On one hand, the compatibility and composability of refactoring types have been studied and summarized. On the other hand, we have contributed a mechanism for detecting nested refactoring. This mechanism uses statistical probabilities of features occurring in the code changes when applying the different refactoring types. A probability model is learned by using the random forests algorithm. Nested refactorings are recognized by this approach by applying the probability model to a code change, when a high probability is found for multiple refactorings.
3. Finally, the use of deep learning algorithms to support refactoring detection is the most important contribution of this dissertation. Firstly, we have implemented the extraction and encoding of refactoring diffs. secondly, we have trained two models based on the structural features and correspondences of the refactoring diffs and implemented the application of the models. Finally, we have optimise the extraction and encoding approaches, trained better models, optimised model deployment, and established a mechanism for model cross-correction, which helps the RefDiff tool to achieve better detection performance. As the refactoring data continues to accumulate, we believe this approach will also become more and more mature.

1.5 Future Work

In terms of my vision for refactoring detection technology, my ideal refactoring detection technology would be intelligent, just like identifying objects in pictures, where users can use powerful models in code control systems to quickly identify refactorings in code. Although we have now completed some of the theoretical demonstrations in my vision and have implemented the ability to use models for matching and detection, there are still some gaps in performance from the ideal. In the future, we will further optimise the detection algorithm and model in three ways:

- We plan to optimise the training data to get a model that performs better and supports more refactoring types. The dataset used currently contains a relatively limited amount and variety of data, for example there is a severe shortage of nested refactoring data. The optimisation solution is to implement a refactoring diff generator to produce training

data, which is based on the refactoring tools in modern IDEs. The user only needs to manually select the refactoring type, and the generator can output the corresponding refactoring diff, which can be used jointly by multiple people in a team to produce better datasets in batches in a short time.

- We will design specialised neural networks to extract structured language features in order to eliminate some of the drawbacks of feature networks designed on the basis of images. The parameters of existing established neural networks are designed based on extracting image features, and the parameter settings of the convolutional kernel are not very suitable for the encoded diff feature array. For example, a convolutional kernel of size 1*1 is dimensionally degrading in images, but in diff data it can speed up the training process with a slight impact on model performance. So a neural network specifically designed for structured languages can balance model performance and training efficiency, and we believe that with more experimentation a better adapted neural network architecture can be designed.
- We will further optimise the encoding approach by using symbols as encoding objects. In designing the encoding approach, we have only encoded words because they are more feature representative, but also symbols, as an important part of the syntax of a programming language, can have the potential to be used as structural features. For example, declaring packages, classes, methods and statements all use different combinations of symbols, which can be encoded as complementary features to the structural features of keywords.

1.6 Chapter Arrangement

The following chapters present our papers presenting the different contributions of this dissertation in detail. They are organized as follows.

Chapter 2: Tan Liang, Christoph Bockisch. “**A Survey of Refactoring Detection Tools**”. In the processing of: *6th Collaborative Workshop on Evolution and Maintenance of Long-Living Systems ((EMLS))*, Stuttgart, Germany, 2019.

My Contributions: This paper presents a meta-analysis of four refactoring detection tools and completes the testing and analysis of the results under the same benchmark. My contributions included, literature compilation, summary of each tool’s methodology, collection and calibration of data for the same benchmark, completion of the entire test, and analysis of the test data.

Chapter 3: Tan Liang, Christoph Bockisch. “**Using refactoring features to solve the problem of nested refactoring**”.

My Contributions: The main contribution of this paper is to use the features of refactoring to train a probability model describing the relationship between refactoring features and refactoring types as a way to achieve the detection of nested refactoring types. My main contribution is to analyse the difference between the features contained in a large number of single refactorings and nested refactorings, to construct a probability model of refactoring features and refactoring types by drawing on the random forest algorithm, and to complete the evaluation of the model by generating some data manually in the absence of data.

Chapter 4: Tan Liang, Christoph Bockisch. “**Checking Refactoring Detection Results Using Code Changes Encoding for Improved Accuracy**”. In the processing of: *22nd IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM)*, Cyprus, October 3-4, 2022.

My Contributions: The main contribution of this paper is the approach for extracting the diffs from the detection results and encoding them as image data for machine learning processing, as well as the training of the machine learning algorithm. My contributions are: analysing the reasons for the low recall of RefDiff, proposing an approach to solve the problem using diff structural features, analysing the performance of diff for all refactoring types, implementing a diff extractor, summarising the encoding rules, completing the data collection, training the result checker and making an evaluation.

Chapter 5: Tan Liang, Christoph Bockisch. “**Diff Feature Matching Network in Refactoring Detection**”. In the processing of: *29th Asia-Pacific Software Engineering Conference (APSEC)*, Japan, December 6-9, 2022. **(Best Paper Award)**

My Contributions: The main contribution of this paper is an approach for encoding the differences between nodes in a syntax tree as image data for neural network matching, thus enabling a complement to the word frequency similarity approach. My contributions are the discovery of flaws in the RefDiff matching algorithm, the creation of an approach to match using the corresponding features in the removed and added parts of the diff, the implementation of a diff encoder, the collection and processing of a large amount of data, the training of a diff feature matching network drawing on image matching networks, the design of a suitable model deployment structure, and the completion of a final evaluation.

Chapter 6: Tan Liang, Christoph Bockisch. “**RefDiff-Model: Model-based Optimization Solution for Refactoring Detection**”.

My Contributions: The main contribution of this paper is adding flags to diff based on refactoring properties and devising new encoding approach that emphasise the uniqueness of tokens, training a better model and establishing a mechanism for model cross-validation. My main contributions are the identification of flaws in the application of existing models, the analysis of these flaws and deficiencies, the design of an optimised diff extractor for them, and also the tailoring of specific encoding optimisations to the characteristics of each model, as well as the proposal and implementation of a model cross-validation mechanism to maintain high confidence levels using the models, the collection and processing of data, and the training of new models, finally complete the evaluation.

Chapter 2

A Survey of Refactoring Detection Tools

2.1 Purpose of Survey

Through 20 years of technology development, the detection refactoring techniques have evolved from the original reverse engineering concept to an actual theory and detection algorithm. Nevertheless, we have identified *four* tools, which we call *complete refactoring detection tools*, by which we mean that the tools are automated and recognize several different non-trivial refactorings. In this chapter, we mainly explain and compare these four tools in detail, especially focusing on the core algorithms and design ideas, and provide readers with a quick guide to refactoring detection techniques. We analyze the current refactoring detection technologies to assess the results reported by the original authors.

To make the technologies more comparable, we apply the tools to common benchmarks as far as possible. In this context a benchmark consists of two versions of a code base—i.e., before and after refactorings have been applied—and a "gold standard"—i.e., a list of the refactorings that have been applied and are expected to be in the results of the tools. To reach a common benchmark, we have used the individual benchmarks used by the authors of the investigated tools so far, and we organized the code bases in such a way that all tools can use them as input. For technical reasons, this was not possible for all code bases, but we could set up two software projects with 170 known refactorings, which could be use for all the tools. We also revised the gold standards applied by the tool authors so fare and found that they missed out a large number of actual refactorings. As a consequence, past studies that used this benchmark have lead to a wrong and much too high recall metric.

Lastly, for one tool that uses the commit history on a Git repository to search for refacotrings, we compared two different commit styles: multiple small commits with only few refactorings applied versus one large commit with many refactorings. We found out that the tool does not perform better with small commits. When developers are interested in the refactorings applied, e.g., between two versions of the software, analyzing small commits might even be disadvantageous: Elements like methods, classes or fields might be refactored multiple times and, thus, be listed repeatedly. This

makes their inspection by developers more tedious and several refactorings will be inspected which are actually obsolete.

Based on the results of our study we present several new insights about the performance of refactoring detection tools. We also outline an idea for future work, in which the reported results are improved by combining several tools.

The rest of this chapter is structured as follows. Section 2.2 outlines the history of refactoring detection technologies, and focuses on the algorithm ideas of the main technologies. Section 2.3 designs and uses experiments to compare and discuss the performance of the four detecting tools. Section 2.4 evaluates the experiments. Section 2.5 discusses criteria for choosing a refactoring detection tool, and our conclusion is presented in Section 2.6.

2.2 Refactoring detection tool

This section introduces four refactoring detection tools. Based on the research in the previous section, they combined the two algorithms of code matching and judging refactoring types to develop practical tools, and they all announced that they have achieved good accuracy. This section will analyze their approach principles, advantages and disadvantages in detail, analyze and discuss each tool.

2.2.1 RefactoringCrawler

Danny Dig successfully pioneered interactive program transformations in the field of refactoring. He released the world's first open-source refactoring tool. RefactoringCrawler algorithm core idea (Danny Dig, 2006): 1. Fast syntactic analysis to detect refactoring candidates; 2. Precise semantic analysis that finds the actual refactorings.

Theory

Syntactic Analysis In order to identify possible refactoring candidates, the algorithm first determines pairs of similar methods, classes and packages. The whole process requires three steps.

Computing Shingles for Method. The Shingles algorithm (Broder, 1997) takes a series of tokens (method body/comment, not including name/signature) as input and computes a multiple set called Shingles. In order to achieve the setting of the number of shingles in proportion to the length of the method body/note, two parameters were defined: W (the length of the sliding window), and S (the maximum size of the resulting multiset). Given a sequence of tokens, the sliding window is used to find all subsequences of W , the shingle for each subsequence is computed and the minimum shingles are used as the result of multiple sets. The tool uses Rabin's hash function (Rabin, 1981) to calculate shingles. The parameter S acts as an upper bound on the space required to represent shingles: a larger value of S means more expensive, while a smaller S means harder to identify strings.

Computing Shingles for Classes and Packages. The shingles for methods are used to compute shingles for classes and packages. The shingles for a class are the minimum S_{class} values of the union of shingles of the methods in that class. Similarly, the shingles for a package are the minimum $S_{package}$ values of the union of the shingles of the classes in that package (Danny Dig, 2006). In this way, duplicate calculations can be avoided under the premise of algorithm efficiency.

Finding Candidates. Using the shingles to find refactoring candidates (a pair of similar entities from the two versions of the component). Let M_1 and M_2 be the multisets of shingles for two methods, classes, or packages. Let $|M_1 \cap M_2|$ be the cardinality of the intersection of M_1 and M_2 . The algorithm normalizes the similarity between 0 and 1 in order to compare the similarity of different pairs. The average similarity from M_1 to M_2 and from M_2 to M_1 is used to solve the case when M_1 is similar to M_2 but M_2 is not similar to M_1 :

$$(|M_1 \cap M_2|/|M_1| + |M_1 \cap M_2|/|M_2|)/2 \quad (2.1)$$

The similarity value is passed above the user-specified threshold value to the semantic analysis.

Semantic Analysis The algorithm is applied to each detection strategy until it reaches a fixed point, and all the strategies share the same log of detected refactoring (*rlog*). This sharing is crucial for the successful detection of multiple refactoring types in the same entity. The order of detecting refactoring types to achieve standardization of shared logs is determined through the following steps:

- Change Method Signature
- Move Method
- Push Down Method
- Pull Up Method
- Rename Method
- Rename Class
- Rename Package

Shared Log. The strategy compares whether entities in one graph correspond to entities in another graph that have been detected for refactoring, in particular rename type. **References.** The strategies calculate the possibility of refactoring based on references between source code entities in each of the two versions of the component. The reference for each node/entity (method, class, package) is defined separately and $\mu(n', n)$ is written for multiplicity from node n' to node n .

Similarity of References. For a given refactoring log, the algorithm uses

metrics to determine the similarity of references to entities in two versions of the component. First the directional similarity between the two nodes that are refactored is defined. Then the overall similarity between the two nodes n_1, n_2 is taken as the average of the direct similarities from n_1 to n_2 and from n_2 to n_1 . When n_1 is similar to n_2 but n_2 is not similar to n_1 , the average of orientation similarity helps to calculate the fairness level.

Detection Strategies. The steps of this strategy are: First perform a quick syntactic analysis to determine if the pair is related to refactoring, then perform a semantic analysis to determine the likelihood of refactoring. Semantic analysis compares references the similarity to the user-specified threshold τ . The strategy for detecting the similarity of renamed refactoring types is similar (*RenamePackage*, *RenameClass*, and *RenameMethod*). The detection strategy between *PullUpMethod* and *PushDownMethod* is exactly the opposite. When performing the second syntax check on *MoveMethod*, it is required that the parent of the two methods is different, otherwise it is easily misjudged as *RenameMethod*. *ChangeMethodSignature* finds methods with the same fully qualified name but different signature.

Discussion

During the development of RefactoringCrawler(RC), some challenges were encountered:

- The size of the code to be analyzed.
- The noise introduced by preserving backward compatibility in the components.
- Multiple refactorings happening to the same entity or related entities.

Strengths

- RC has higher precision and recall than the method of detection refactoring in the same period.
- Despite a lot of noise and renamed noise, RC was able to detect refactoring. Thus, it demonstrates robust detection capabilities.
- RC is designed to run inexpensive Syntactic detection first, then Semantic detection for refactoring candidates, thus reducing the cost of detecting, improving the scalability.

Limitations

- Poor support for interfaces and fields. In addition to declaring names, RC cannot detect refactorings that occur on fields or interface methods because they do not contain any method bodies.
- Experimental verification is required. Because RC is based on the calibration threshold, setting the threshold value too high will increase the accuracy but will miss some refactorings. If the threshold is too low,

it will detect too many false positives and reduce the accuracy rate. Therefore, the threshold value determination requires a large amount of experimental verification.

Conclusion

The combination of syntactic analysis and semantic analysis effectively avoids their respective drawbacks, meanwhile creating a detecting refactoring tool RC with higher precision and recall. RC can be used on any two versions of the system and automatically detects applications containing 7 refactoring types. The key element refactoring log (*rlog*) helps in the automatic migration of component-based applications. *rlog* can not only improve the way the current configuration management system handles renaming, but refactoring logs can also help developers understand how object-oriented systems evolve from one version to another.

2.2.2 Ref-Finder

Miryung Kim is an associate professor in the Department of Computer Science at the University of California, Los Angeles. Her research focuses on software engineering, specifically on software evolution.

Theory

Logic-based Representation and Refactoring Rules

- Predicate definition. In order to analyze internal content of method bodies, control-structures and variable definitions, based on algorithm LSdiff (M. Kim, 2009) the author adds 6 new predicates (conditional, cast, trycatch, throws, variable definition, and methodbody). The author uses a logical query-based program investigation tool JQuery (D. Janzen, 2003) to extract the logic facts from the old version and the new version respectively. JQuery uses the Eclipse JDT Parser to analyze the Java program structure, and first uses the set-difference to calculate the face-base variation. Then use author's previously proposed tool (M. Kim, 2007) that can automatically infer structural changes to infer the method-header level refactoring is used. Subsequently, the spurious code in/decrease caused by code renaming is removed. The next step is to use four input parameters (M. Kim, 2009) to define the output rules. Finally, post-processing uses the SET-COVER algorithm (E. Balas, 1976) to avoid overlapping matches in previous processing. Throughout the process, comparison thresholds are used to determine whether similar facts are generated. If the similarity between the two candidate methods is higher than the threshold σ , Ref-Finder will generate similar bodies.

- Coding refactoring types as template logic rules. Based on the type of refactoring proposed by Fowler’s catalog (Fowler, 1999), the author defines a refactoring type that is suitable for setting logic rules (including 63 types), and the remaining 9 refactoring types cannot logically define rules for some special reason. Take “Extra Method” as an example, template logic rule:

$$\begin{aligned}
 & \text{addedmethod}(\text{toMethodFullName}, \text{toMethodShortName}, \text{toClassFullName}) \\
 & \wedge \text{aftermethod}(\text{fromMethodFullName}, \text{fromMethodShortName}, \\
 & \text{fromClassFullName}) \wedge \text{similarbody}(\text{toMethodFullName}, \text{toMethodBody}, \\
 & \text{fromMethodFullName}, \text{fromMethodBody}) \wedge \text{aftercalls}(\text{fromMethodFullName}, \\
 & \text{toMethodFullName}) \rightarrow \text{extractmethod}(\text{fromMethodFullName}, \\
 & \text{toMethodFullName}, \text{toMethodBody}, \text{toClassFull} - \text{Name})
 \end{aligned}$$

Refactoring Identification via Logic Queries

- Topological sort (K. Prete, 2010a). These ordering relations are formulated according to the template logic rules. Topological sorting algorithms (T. H. Cormen, 2001) are used to determine the type of refactoring that needs to be inferred first.
- Finding concrete refactoring instances. A concrete refactoring instance is found by converting the premise of each rule to the logical query and by using the Tyruba logic programming engine (Volder, 1998) to invoke the logical fact database query.

Detection steps The Ref-Finder inspection process has six steps:

$$\text{Preparation} \rightarrow \text{Extract } V_1 \rightarrow \text{Extract } V_2 \rightarrow \text{ComputeDifferences} \rightarrow \\
 \text{PerformLSDiff} \rightarrow \text{OutputResult}$$

The above steps can be divided into three parts, first extracting the detectable facts in the two versions (including the derivation part), then computing the difference of the factbase (all the changed codes), and finally converting the atomic changes into LSDiff changes and finding refactorings (this part is the most time consuming).

Discussion

Strengths: Compared to other refactoring detection tools, Ref-Finder (RF) can support 63 refactoring types from the Fowler' refactoring type catalog, while other tools can only detect a few to a dozen different types. Irrespective of Ref-Finder's detection efficiency, RF has the most comprehensive coverage for the index of refactoring type coverage. Moreover, the author confirmed that the overall accuracy of the RF recognition refactoring is 79% and the recall is 95% through some experiments and data.

Limitations

- The effectiveness of LSDiff to extract facts will directly affect the recognition of Ref-Finder on various types of refactoring.
- The complex refactoring consists of a set of atomic refactorings, so the error recognition of the atomic refactoring will directly lead to misjudgment or missed judgment of the complex refactoring.
- The refactoring definitions mentioned by Fowler may have been misinterpreted in the algorithm.

Conclusion

Ref-Finder adds some new predicates to LSDiff's recognition function, and uses logical meta-programming methods to identify complex refactoring types from two program versions. Before it can be renamed, simple refactoring types such as move and basic extraction are promoted to new tools that can cover 63 types of refactorings. Moreover, different types of refactorings are expressed as logic rule templates and then are used by the logic programming engine to infer concrete refactoring instances. After the experiment was verified, the higher precision and recall were achieved.

2.2.3 RefDiff

Marco Tulio Valente focuses on Software Engineering, specifically in the areas of Software Architecture, Software Maintenance and Evolution, and Software Repository Mining.

Theory

RefDiff (Danilo Silva, 2017) uses a heuristic approach based on static analysis and code similarity to detect refactoring between two revisions of the system. The detection algorithm is divided into two main phases: Source Code Analysis and Relationship Analysis. During the first phase, the system source code is parsed, and high-level source code entity models (types, methods, and fields) are constructed. Each piece of source code entity model is divided into two parts (pre-change and post-change), which are the union of corresponding types, methods and fields. Secondly, during Relationship Analysis, the key is to analyze the relationship between pre-change and post-change.

Matching Relationship The key to the matching relationship is that a unique corresponding item with the same qualified name in pre-change code can be found in post-change code. The final step in finding the actual relationship is to select a non-conflicting relationship with a higher similarity index from the list of potential relationships. Examples of matching relationships are Same Type, Move Type, Rename Type, and Pull Up Method.

No-Matching Relationship The pre-change code can find the source code of multiple corresponding codes in post-change code (Non-single correspondence). Examples of matching relationships are *Extract Method*, *Inline Method* and *Extract Supertype*.

Computing Similarity Calculating code entity similarity is the key to using this algorithm to find relationships. The first step is to represent all source code as a multiple set of tokens. Later, the author used the Weighted Jaccard coefficient (F. Chierichetti, 2010):

$$J(X, Y) = \frac{\sum_{i=1}^n \min(X_i, Y_i)}{\sum_{i=1}^n \max(X_i, Y_i)} \quad (2.2)$$

to define the code entity similarity. Let n be the set of all possible tokens, and X_i, Y_i be the weight function of a token for an entity code.

- **Weight of a token for a code entity.** Since each different code element is a token, fewer occurrences of tokens are a better indicator of the similarity between computational methods than more tokens. The authors use a well-known variant of information retrieval technology TF-IDF (G. Salton, 1986), which reflects the importance of terms to documents in a collection of documents. In the context of a code entity, the author defines a token as a term and defines the body of a method (or class) as a document. The author obtains the weight function of the token for the code instance through

$$tf - idf_t = tf_t \times idf_t \text{ and } idf_t = \log_a(1 + N/df_t) \quad (2.3)$$

From the formula, idf_t and idf are inversely proportional, explains how with increasing frequency of tokens in the code entity set, the code elements become less important.

- **Similarity of fields.** The similarity detection for types and methods can be directly calculated by the similarity function described above. However, the similarity calculation for fields without bodies is not applicable. Thus, the author defines the concept of a field virtual body, directly using the fields corresponding to each other, before and after the change, to verify the similarity.
- **Similarity for non-matching relationships.** When the source code of two entities has an inclusion relationship, such as *Extract Supertype*, *Extract Method*, and *inline Method*, the author defines a specific version of

the similarity function:

$$J(X, Y) = \sum_{i=1}^n \min(X_i, Y_i) / \sum_{i=1}^n X_i \quad (2.4)$$

The basic principle is that when the multiset of pre-change token is included in the multiset of post-change token, they get the maximum similarity, but the inclusion relationship cannot be reversed.

Calibration of similarity thresholds The RefDiff algorithm relies on thresholds to find relationships between entities. The author defines a threshold λ to determine possible potential relationships between entities. The threshold may affect the algorithm's recall and precision. For this reason, the decision of the threshold needs to take both precision and recall, to ensure that the precision is high and the recall is relatively low, and vice versa. So the choice of threshold must ensure that precision and recall simultaneously reach a relatively high level. After a lot of experimental research, the author determined the calibration threshold corresponding to each refactoring type.

Discussion

Strengths:

- The author compares RefDiff with several other detection methods through experiments, and uses RefDiff detection refactoring to obtain higher precision and recall. The average value of the final precision reached 85.7%, and the recall reached 94.1%.
- Experimental results: RefDiff is not the fastest to perform the detection refactoring, but has a relatively high precision and recall per unit time.

Limitations: Like the RefactoringCrawler, RF also needs a lot of experimentation to determine a threshold, which is only a relative value, and can not detect high precision and recall for all code.

Conclusion

RefDiff uses a heuristic approach based on static analysis and code similarity to detect 13 common types of refactoring. RefDiff leverages existing technology and introduces some novel ideas, such as TF-IDF weighting. In the author's comparative experiment, precision and recall of RefDiff are also superior to all other detection tools, reaching a very high level.

2.2.4 RefactoringMiner

Nikolaos Tsantalis researches software maintenance, empirical software engineering, refactoring recommendation systems, and software quality assurance. Within the context of his research, he has developed some tools, such as

the RefactoringMiner, JDeodorant and Design Pattern Detection tool, which have been researched and used by researchers and developers in related fields. Moreover, RefactoringCrawler creator Danny Dig also joined the development and research of RefactoringMiner.

Theory

Notation The author adopted the notation of Biegel et al. (Benjamin Biegel, 2011) and extended the definition on the basis of the original to indicate that the author used the Eclipse JDT Abstract Syntax Tree (AST) parser to extract the information from each version. This approach has also proven to be a good approach in many applications (PeterWeissgerber, 2006) that require syntactical analysis. In the extraction process, in order to avoid storing AST information into memory, first each statement/ expression is simplified, redundant blanks and multi-line characters are removed and the string is expressed in print format. Second, the AST Visitor is used to extract variable identifiers, method invocations, class instantiations, variable declarations, types, literals, and operators appearing in each statement/ expression and they are stored in the print format of the corresponding statement node.

Statement matching Statement matching between two pieces of code snippets is a core feature in the tool. The biggest feature of the algorithm is that it does not need to define the similarity threshold. The author is inspired by Fluri et al (Beat Fluri, 2007) to develop a matching algorithm that does not require the use of similarity measures. The algorithm uses a bottom-up statement matching method, starting with the basic leaf statement, stepwise matching, and finally to the compound statement. In order to reduce the possibility of mismatching, the author adopts a conservative method, first strict and then loose. The conditions for matching from the first round of matching are successively relaxed, and the matching condition of the previous round is excluded from the matching by the next round. The authors describe these rounds of matches as "safer" matches, in which way the next round will detect fewer combinations of statements. Take the matching leaf statement as an example. In the first round, it matches statements with the same string representation and nesting depth. In the second round, the statements with the same string representation are matched, removing the condition regarding the nesting depth. In the last round, it matches the statement that becomes the same after replacing the different AST nodes between the two statements.

In the first two rounds, the author uses a state matching algorithm with Function *matchNodes*, Function *findBestMatch*. The former can determine all possible pairs of nodes in the two pieces of code and store them. The latter according to special logic sorts the stored node pairs and selects the highest ranked node pair. In the final round, the author used another algorithm that judges Syntax-aware replacements of AST nodes, which takes two statements as input and performs the replacement of the AST node until the statement is identical in text. this does not need to define a similarity threshold,

compared to existing methods that rely on text similarity, and the replacements found can also help infer that other editing operations occur in the refactoring code. Meanwhile Function *compatibleForReplacement* can check the expression used to invoke the method, in the special case of considering the replacement of two method invocations.

In the input state in all rounds, the author introduces two kinds of preprocessing techniques, Abstraction and Argumentation, to handle the specific changes that occur in the code when Extract, Inline, and Move Method are refactored. (Abstraction (Nikolaos Tsantalis, 2018): Some refactoring operations usually introduce or eliminate return statements when extract or inline methods, respectively. Argumentation: Some refactoring operations may replace expressions with arguments and vice versa.)

Refactoring detection The author divides the refactoring detection into two phases. During the first phase, the author uses an algorithm to match code elements from top to bottom, starting from the class, and then to methods and fields. Matching principle: Two code elements have the same signature, so they are relatively inexpensive. During the second phase, the algorithm matches the remaining code elements that are deleted or added from bottom to top, starting with the method, and then to the class, looking for code elements with signature changes or participation in the refactoring operation. Code inside the bodies needs to be checked, so the relative cost will be expensive. **Examination order of refactoring types.** The detection order is an important factor affecting the accuracy of the algorithm. The order of refactoring types: *Change Method Signature, Extract Method, Inline Method, Change Class Signature, Move Method, Move Field, Extract Method and Move Method, Extract Supertype, and Change Package*. The author sorts the refactoring types according to the location of the change (Danny Dig, 2014) and some experimental conclusions (Jim Buckley, 2005) (Stas Negara, 2013). The local refactoring is more frequently used than the global refactoring, and the refactoring possibility is higher. So the detection order starts with the local refactoring type (method/class) and then global refactoring (classes/packages).

Best match selection Since the same piece of code cannot support multiple refactoring operations at the same time. When multiple matches occur, choosing the best match is the key reason to ensure accuracy. The algorithm sorts based on 4 criteria:

- Method pairs with more matching statements are ranked higher.
- Method pairs which have more of the same statements.
- Method pairs with text-like statements.
- Method pairs with text-like names.

Discussion

Strengths

- The author proposes the first refactoring detection algorithm that does not require a code similarity threshold.
- The RefactoringMiner (RM) tool operates on version control submission and provides an externally used API. It takes the commits in the git repository as input for efficient and scalable refactoring detection.
- RM only analyzes additions, deletions and changes in files between the two revisions. The reduction in the number of combinations of code elements makes RM not only more efficient, but also reduces the occurrence of false matches.
- Researchers can use RM to reduce the noise generated by refactoring (Costa, 2017) (Steven Davies, 2014), such as file/directory renaming, which significantly improves the accuracy of other tools.

Limitations

- RM only analyzes the files added, deleted and changed between two revisions. However, a missing context (ie, the unchanged file) may cause RM to report an incorrect refactoring types.
- RM is currently unable to detect nested refactoring operations.
- RM currently only supports 15 detection refactoring types.
- Although the authors did their best to reduce bias by combining the input of the two detection tools and manual verification, but still cannot generate justice.
- An algorithm that judges Syntax-aware replacements of AST nodes, which cannot handle possible changes in the parameter list, since the algorithm can only perform one-to-one AST node replacement.

Conclusion

During detecting refactoring, the author used novel techniques such as abstraction and argumentation to handle changes in code statements. Meanwhile the author claims that RM is by far the most accurate, complete and representative refactoring tool, with 98% precision and 87% recall and with very low computational cost. RM guides the new direction for the ability to submit operations:

- Researchers can create refactored data sets with high precision from the entire commit history of the project and study various software evolution phenomena at a fine-grained level.
- Bug-inducing analysis techniques can use the commit level to gain refactoring information to improve accuracy.
- The refactoring operation can be automatically recorded at the time of submission.

- The submission of visualization difference covers information of the refactoring in order to help code review and evolutionary understanding.

2.3 Experimental Comparison of Tools for Detecting Refactorings

To verify the issues mentioned in Chapter 1 we apply the four tools RM (“RefactoringMiner”), RF (“Ref-Finder”), RD (“RefDiff”) and RC (“RefactoringCrawler”) to the code base of several projects with a version history where the used refactorings are well-known (called the gold-standard). Thus, the effectiveness of the detection tool can be determined by a comparison of precision, recall and execution time.

All tools need as input a version of the code base before the refactoring was applied and a version after refactoring. The refactoring detection tools investigated in this study expect this input in different ways: either as two Eclipse projects in the local file system (RC, RF) or as a version history in a GIT repository (RM, RD).

As the code base used in our experiments, we take a set of GitHub open source repositories for which a gold standard of applied refactorings is published in “Why We Refactor? Confessions of GitHub Contributors” (Danilo Silva, 2016). The sample consists of a number of repositories, including lots of well-known projects (“Projects”). This code base can immediately be used by the GIT-based tools. In addition, we use the projects EclipseUI, Struts and JHotDraw for which the two necessary versions (in form of an Eclipse Java project each) and a gold standard are published by the authors of RefactoringCrawler. These code bases can immediately be used by the file system-based tools.

To make all code bases usable for all tools, we needed to create a GIT repository for the code based where only Eclipse projects were available and vice versa. To start with, we created a new repository and submitted the corresponding versions of the three projects EclipseUI, Struts and JHotDraw. In the case of EclipseUI we were not able to create a repository readable by RM and RD, for reasons we do not currently understand, therefore we can only use Struts and JHotDraw in the further experiments; we leave EclipseUI for future works.

In principle, we could also have created projects in the file system for the code bases from GIT. However, all projects for which a gold standard was available are written in Java 6 or up, which is not supported by RC and RF. Therefore, we did not perform this conversion.

As a consequence, we have not two sets of code bases that can be processed by different groups of refactoring detection tools, leading to two different experiments. First we compare all tools based on their results when applied to Struts and JHotDraw, second we compare RM and RD based on the code bases from GitHub. For all projects used in our experiment the gold

TABLE 2.1: RefactoringCrawler(1.0.0)

Type	True positive	False positive	False negative	Precision	Recall
Rename Method	25	0	18	1.000	0.581
Rename Class	1	0	1	1.000	0.500
Move Method	20	2	13	0.909	0.606
Pull Up Method	1	0	0	1.000	1.000
Total	47	2	32	0.977	0.672

TABLE 2.2: Ref-Finder(1.0.4)

Type	True positive	False positive	False negative	Precision	Recall
Rename Method	29	69	14	0.296	0.674
Extract Method	56	66	35	0.459	0.615
Move Method	6	0	29	1.000	0.171
Pull Up Method	0	0	1	0.000	0.000
Total	91	135	79	0.439	0.365

standard contains for each refactoring that is know to be applies its refactoring type and the location.

2.3.1 Experimental results











Accuracy

To determine the performance of each tool, we determine their precision and recall. The precision calculation is the ratio of all "correctly retrieved items(TP)" to all "actually retrieved (TP+FP)", which means that the retrieved result is accurate, $\text{precision} = \text{TP} / (\text{TP} + \text{FP})$. Recall calculates the ratio of all "properly retrieved items(TP)" to all "item(TP+FN)" that should be retrieved, representing whether the retrieved result retrieves all items, $\text{recall} = \text{TP} / (\text{TP} + \text{FN})$. Therefore, the higher the precision and recall, the higher the performance of the detection tool. Since precision and recall are inversely proportional in reality, high precision and recall are the goal of all researchers. In contrast to other publications, we present for each tool the precision and recall per refactoring type instead of per project the tool is applied to.

TABLE 2.3: RefactoringMiner(1.0.0)

Type	True positive	False positive	False negative	Precision	Recall
Rename Method	38	4	5	0.905	0.884
Extract Method	69	5	22	0.932	0.758
Move Method	16	0	19	1.000	0.457
Pull Up Method	1	0	0	1.000	1.000
Total	124	9	46	0.959	0.775

TABLE 2.4: RefDiff

Type	True positive	False positive	False negative	Precision	Recall
Rename Method	35	2	8	0.946 	0.814 
Extract Method	62	69	29	0.473 	0.681 
Move Method	15	0	20	1.000 	0.429 
Pull Up Method	1	1	0	0.500 	1.000 
Total	113	72	57	0.730 	0.731 

First Group (RC, RF, RM and RD) Because the types of refactoring supported by these four tools are not exactly the same, so we only report results regarding refactoring types supported by all tools. Only for RefactoringCrawler we report results on a different set of refactorings, as we would have too few data for a meaningful analysis otherwise. We discuss this further in the threats to validity.













The comparison of RC, RF, RM and RD is shown in tables 2.1–2.4. During the experiment, we found that the Expected Result (i.e., from our gold standard) corresponding to these sample projects is not complete, and it is not completely correct and many TruePositives are missing. In order to ensure the reliability of the experiment, we review the data’s TruePositive and get a more comprehensive Expected Result, which is used as the standard answer. From the experimental data, we can see that RC has a higher precision and recall than RF.

The four refactoring types detected by RC reached a very high level in precision, but the recall is very unstable in all types. RF can detect 63 refactoring types, but the precision and recall shown in this experiment is very low. During the experiment, it was found that when JHotDraw, Struts was detected, RF detected 358, 378 refactorings respectively. But the data presented here is limited to the detection of the commonly supported refactoring types.

The comparison between RC and RF is quite different from some previous studies. RC has higher precision and recall in the four refactoring types, but the experimental results of RF need to be carefully considered and studied. For RC, as the earliest detecting refactoring tool, although the experimental data results are good, the number of experimental samples is too small. In Danilo Silva and Marco Tulio’s experiments (Danilo Silva, 2017), the RC recall and precision was 58.2% was 35.6%, which is comparatively low. Detection can only cover 7 kinds of Refactoring types, so our results can not be directly representative of RF. Soares and Murphy-Hill (G. Soares, 2013) found that the precision of the RF was 35%, recall was 24%, while False Positives were 65%, which confirms this experiment to some extent. A recent study (Péter Hegedűs, 2017) also showed that the overall average accuracy of RF is only 27%, which is too low for inspection tools that can cover 63 refactoring types.

In contrast, the newer tools RM and RD perform better in this experiment, especially RM provides already provides good results.

TABLE 2.5: Detection Results for Accuracy(supported refactoring types)

Type	True positive	False positive	False negative	Precision	Recall	F1 score
RefactoringCrawler	47	2	32	0.959 	0.595 	0.734 
Ref-Finder	91	135	79	0.403 	0.535 	0.460 
RefactoringMiner	124	9	46	0.932 	0.729 	0.818 
RefDiff	113	72	57	0.611 	0.665 	0.637 

Comparison under common benchmark To compare the four most complete refactoring detection tools discussed above, we created a common benchmark¹ to which we applied all tools. In this way, their ability to detect refactorings is determined using the same code bases, change sets, and expected results. The benchmark combines code bases and corresponding changes used in previous studies to evaluate refactoring detection tools. We improved it to be applicable to all tools by making it accommodate for the different forms of input required by the different tools: either by means of a Git repository or by means of two Eclipse projects reflecting the code before and after the change.

In Table 2.5, we show the results for this combined benchmark and for all tools, but limited to only the refactoring types commonly detected by all four tools (which is only four types). This confirms the previous results from the literature, namely that RefactoringMiner performs best among the available tools. While RefactoringCrawler has a slightly higher precision (96% rather than 93%) than RefactoringMiner, the latter has a higher recall (73% compared to 60%). The F1-score computes a combined measure for precision and recall, and (as can be seen in the table) RefactoringMiner has the highest ranking in the F1-score.

As said before, this comparison of all tools only considered the common subset of refactoring types. Because RefactoringMiner produced the best results in this comparison (which is consistent with the literature), we further focused on this tool and also determined the precision and recall for all refactoring types supported by RefactoringMiner (using the same benchmark applications as before). In this more detailed benchmark—results are shown in Table 2.6—the results remain very good, but diverge slightly from the benchmark presented above, which was limited to only four refactoring types. Considering all code bases in the benchmark and all supported refactorings, RefactoringMiner has a total precision of 94% and a total recall of 75% (Table 2.6), which is in line with the results reported by the original authors.

Although still high, the precision in our detailed benchmark is lower than in the limited benchmark, while the recall is higher. Taking a closer look, we identified that RefactoringMiner has problems with the two refactoring types *Move Class* and *Rename Package*, as shown in Table 2.6. For these refactoring types, the recall drops to 34% for *Move Class*, respectively to 11% for *Rename Package*. This is inconsistent with the results previously presented in the

¹The benchmark can be accessed online:
<https://bitbucket.org/tanliang11/struts/src>
<https://bitbucket.org/bockisch/jhotdraw/src> and <https://umrplt.bitbucket.io/>

TABLE 2.6: RefactoringMiner(1.0.0)

Type	True positive	False positive	False negative	Precision	Recall
Extract Method	28	1	1	0.966	0.960
Inline Method	26	2	0	0.929	1.000
Pull Up Method	19	0	0	1.000	1.000
Push Down Method	10	1	0	0.833	1.000
Move Method	44	4	4	0.820	0.886
Move Class	21	0	44	0.600	0.403
Extract Superclass	9	0	0	1.000	1.000
Extract Interface	5	0	0	1.000	1.000
Move Attribute	85	8	0	0.934	1.000
Rename Package	5	0	39	0.600	0.172
Push Down Attribute	10	0	0	1.000	1.000
Total	262	16	88	0.880	0.856

TABLE 2.7: RefDiff

Type	True positive	False positive	False negative	Precision	Recall
Extract Method	29	2	0	0.935	1.000
Inline Method	27	0	0	1.000	1.000
Pull up Method	19	0	0	1.000	1.000
Push Down Method	2	1	8	0.500	0.667
Move Method	45	38	4	0.620	0.800
Move Class	60	5	5	0.943	0.900
Extract Superclass	3	0	6	0.600	0.500
Extract Interface	5	0	0	1.000	1.000
Move Attribute	13	4	27	0.765	0.325
Rename Package					
Push Down Attribute	10	0	0	1.000	1.000
Total	213	50	50	0.836	0.819

literature. The reason is that the benchmarks previously used were less complete and even missed out several actually applied refactorings in the set of expected results.

Second Group (RefactoringMiner and RefDiff) In the previous experiment we compared all four tools, however with a smaller number of code bases. For RefactoringMiner and RefDiff we can use a larger code base with more known refactorings to get more reliable estimated of their precision and recall.

The comparison of RM and RD is shown in Table 2.6 and Table 2.7. As the latest refactoring tools, RM and RD have reached a very high level of precision and recall, reaching almost 100% in many types of detecting. But in terms of overall stability, RM is more advantageous than RF. Except for *MoveClass*, other types of detection have reached a relatively high level. The

precision and recall of RD in *MoveMethod*, *PushDownMethod* and *ExtractSuperclass* are not high, which is inconsistent with the experimental results of RD developers, but we can in total confirm the precision and recall of RD for these refactoring types.

The comparison between RM and RD is consistent with previous research results, although both RM and RD reports said that they are better than the other tool. However, the experimental results show that both RM and RD are capable refactoring detection tools with their own advantages and disadvantages. RM as a typical representative without calibration threshold can cover more refactoring types, with a overall average precision (88%) and recall (85.6%) are higher than RD. RD is the most advanced detection tool for calibration thresholds. RefDiff can achieve higher recall in refactoring types with respect to code similarity, but RD cannot handle token changes caused by refactoring itself and overlap refactoring. Both RM and RD use API as a detection tool for connecting external code repositories. Moreover, both RD and RM source code use Gradle as a project management tool. All in all, RM and RD are very good, and they can complement each other in many aspects. At the application level, using RM and RD to jointly detect can greatly improve the precision and recall. At the design and code level, method integration may result in more efficient detection tools, as we will discuss further in Section 2.5.

Comparing the results of RM and RD in the first and the second group of experiments, we can see that RM has a similar average precision in both cases and a slightly better precision in the first experiment group. The performance of RD in the second group is better than in the first. The larger number of false positives in RD mainly comes from a large number of false positive *ExtractMethod* being detected. A large part of them are derived falsely detected method extractions from one to a different class. Therefore, we expect that the number of false positives could be significantly lowered if the reported results for the extract method refactoring are limited to cases were the extracted method is placed in the same class.

Performance

In terms of execution time, the machine is equipped with Intel® Core(TM) i7 – 6700CPU@3.40GHz, 16GB memory, Windows 7, and Java 1.8.0, *Eclipse x4*.

RefactoringCrawler, Ref-Finder, RefactoringMiner and RefDiff As shown in Table 2.8, RD is the most outstanding tool in terms of execution time, followed by RM, which is not only time-consuming, but also has excellent detection results. In contrast, the new tools have also completely surpassed the old tools in terms of execution time.

RefactoringMiner and RefDiff In general, as shown in the Boxplot of Execution in Figure 2.1, the execution time of the two tools is not much different

TABLE 2.8: Execution Time Comparison

Code Files	JHotDraw5.2—JHotDraw5.3 160—195	Struts1.1—Struts1.2.4 460—469
RefactoringCrawler	0.5 minutes	5.0 minutes
Ref-Finder	14.0 minutes	20.0 minutes
RefactoringMiner	12.0 seconds	159.0 seconds
RefDiff	9.0 seconds	61.0 seconds

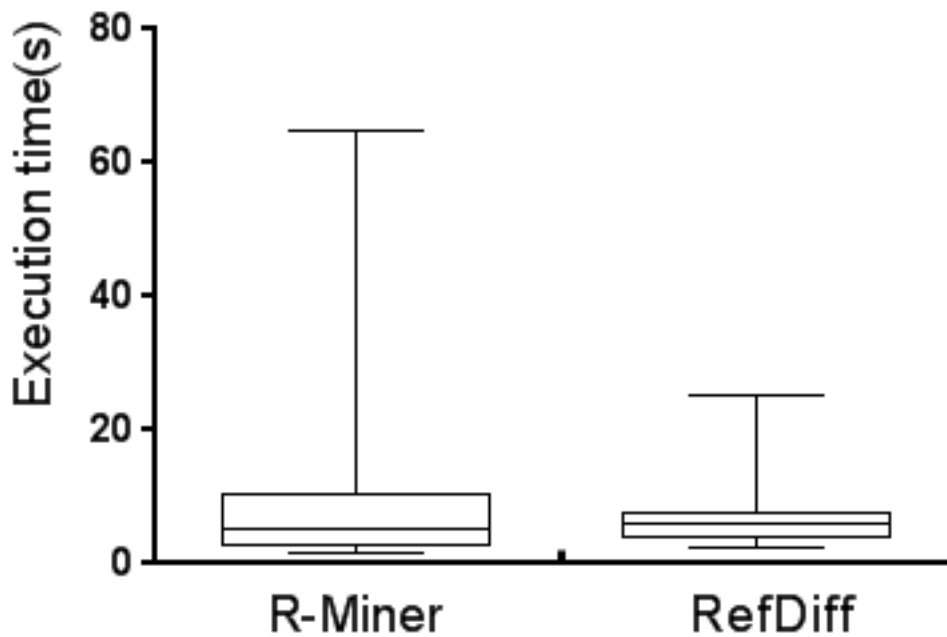


FIGURE 2.1: Boxplot of Execution Time

during the experiment, but from the data point of view, in the case of eliminating the outliers, the time difference of RD is smaller, the average time is relatively shorter, and RD is slightly better than RM in the samples of this experiment.

2.3.2 Comparing influence of repository structure on RM

During the experiment, the RM has the detection function (`detectBetweenCommits` or `detectBetweenTags`) that detects two specified commits. The working principle is: According to the relationship between the two versions on the branch tag tree, the relationship lines connecting the two versions are determined. There are multiple commits on the line, and the multiple commits are divided into a series of related parent and child commit pairs (child commit of each pair is the parent commit of the next pair). Step-by-step detection of refactorings that occur between each commit pairs. At the same time, we also compared another approach, create a new repository, upload

the required two version commit numbers, and directly use the functions in the First Group for detection. The two approaches are analyzed as follows: 1. Since RM needs to implement statement matching through hierarchy (*Package* → *Class* → *Method*). If the package or class changes in previous commits (rename or move), then skip the intermediate commit, directly detect the commits of two versions will get a lot of mismatched results, and also miss the package and class level refactoring. 2. Direct comparisons will also increase the FalseNegative of some method level refactorings that occur at the relevant position. The recall becomes very low, and the corresponding TruePositive and FalsePositive will also decrease, which will also affect the overall precision (but the refactoring quantity is needed). 3. However, in the step-by-step detection process, the normalization of the matching process can greatly reduce the appearance of FalsePositive, and also reduces the overlap and repetition caused by multiple commits, thereby improving the overall accuracy. Both of these detection methods have advantages and unavoidable disadvantages, and require more extensive experiments and data to quantify the impact of their respective strengths and weaknesses on the results.

2.4 Threats to Validity

2.4.1 External Validity

In the comparative experiment of RM and RD, we used an average of 5 repositories for each refactoring type, and the samples of RM and RD are the same. When an experimental sample cannot be detected simultaneously by RM and RD, we have also set up the corresponding plan to use the alternative sample to guarantee the experiment. For experimental samples of RC, RF, RM and RD, we use two versions of two projects, each: Struts and JHotDraw. While the number of projects is low, the number of individual refactorings to be detected of 170 is sufficiently high. Nevertheless, we cannot tell if the coding and commit style of Struts and JHotDraw have an influence on the investigated tools' performance; a wider spread of projects would be needed.

Since we are not affiliated with any investigative tool developer, we don't need specific experimental results, expect to have no researcher bias. In order to avoid prejudice problems, we carefully compare the expectation reconstruction and the reality refactoring from quantity, position and type, and guarantee the objectivity of the research results. Therefore, the threat of researcher bias does not apply.

2.4.2 Internal Validity

In the RC and RF test, because these two tools are based IDE plug-ins, such as to record the time of execution, calculate precision and recall, some of the results was manually and confirmed repeatedly as to ensure the validity of the experiment. We still cannot guarantee that the list of expected results is

complete. If we miss any, the number of false negatives and thus the recall would be too low.

2.4.3 Discussion

1. Are calibration threshold detection methods (RF, RC, RD) superior to methods that do not require such calibration (RM)?

The calibration similarity threshold is a key indicator for detecting refactoring similarity. For the low threshold, the number of candidate pairs matched is large, the false positives increase, and the detection execution time becomes longer, but giving more opportunities for candidates also means lowering the precision in order to improve the recall. For high thresholds, the number of matched candidate pairs is small, meaning that only almost identical candidate pairs are considered. Although the running time is shorter, some TP refactoring options will be missed, which means that precision is guaranteed and the recall is reduced. So many researchers are working on how to choose a universal threshold that can take both precision and recall. As a developer of RM, through the threshold problem of UMLdiff: the derived threshold may have been over-fitting to the characteristics of the item being inspected, and therefore cannot be sufficiently general. Calibrating the threshold is a very tedious process as different software and architectures have different precision and recall at the same threshold. To ensure accuracy, constant calibration thresholds are needed to cope with new code repositories.

In order to avoid such problems, RM uses novel techniques such as abstraction and argumentation to deal with code statement changes, avoiding the complicated calibration process while achieving matching similarity effects, and achieving very high precision and recall. However, the Section 3.1 experimental data shows that the difference between the two methods for processing similarity matching is not very obvious. The operation of the RD experiment uses the default threshold and does not require repeated calibration after the default threshold is obtained. So in practice, we need more research and experimentation to verify.

2. Can the experimental results of the authors be repeated under realistic conditions? In the case of RF, the results reported by different authors are contradictory - which ones are the most likely to be representative?

Precision and recall are important factors to determine the performance of the four inspection tools. The authors of the four tools in this article have released the following metrics: RM (Precision: 98% and recall: 87%), RD (Precision: 100% and recall: 88%), RC (Precision: 85% and recall: 85%), RF (Precision: 79% and recall: 95%). Through experiments, it can be found that RM, RD and RC are very close to the previously declared data, but combined with some previous experiments, only RM

TABLE 2.9: Detailed accuracy results for RefactoringMiner and the *Move Class* refactoring type.

Number	commit	Expect Result	True positive	False positive	False negative	Precision	Recall
1102923	eclipse-themes 72f61ec	3	3	0	0	1.000	1.000
1107905	elasticsearch f77804d	4	0	0	4	N/A	0.000
1116663	buck 1c7c03d	10	7	2	3	0.778	0.700
1118645	okhttp c753d2e	29	0	0	29	N/A	0.000
1130125	WordPress-Android 9dc3cbd	3	3	0	0	1.000	1.000
1132674	orientdb f50f234	12	4	1	8	0.800	0.333
1134151	gradle 36ccb0f	7	7	0	0	1.000	1.000
1139721	liferay-plugins 78b5475	5	5	0	0	1.000	1.000
1140071	docx4j e29924b	184	184	0	0	1.000	1.000
1147092	neo4j 4beba7b	6	6	0	0	1.000	1.000
1147835	jersey ee5aa50	8	8	0	0	1.000	1.000
1150594	hazelcast f1e26fa	13	13	0	0	1.000	1.000
1152530	hydra 7fea4c9	7	4	0	3	1.000	0.571
1159198	jedis 6c3dde4	26	26	0	0	1.000	1.000
Total		317	270	3	47	0.989	0.852

















and RD achieve high stability and accuracy. That is, in the face of different experiments and objects, always maintain high efficiency. RC and RF represent two extremes in this experiment. Combined with some related experiments and data, it can be found that RC and RF have detection functions. However, the precision and recall rate published by the inventors is higher than reality, so it is necessary to improve the backward algorithms and ideas.

2.4.4 Detection of *Move Class* and *Rename Package*

After discovering this behavior of RefactoringMiner, we wanted to understand it further and conducted a more comprehensive and in-depth test for the refactorings *Move Class* and *Rename Package* respectively. Again, we used the original benchmarks with our extended set of expected results. Since we specifically focused on the two refactorings, we only considered samples that contain at least two instances of the refactoring we are investigating. The test results are shown in the Tables 2.9 and 2.10. The "Number" column in the table refers to the test samples in the code base to make the experiment repeatable and the results traceable. A benchmark sample consists of two versions of a code base, where the second version is based on the first one but with the refactorings applied. In the tables, we refer to the two versions by giving the Git commit number, thus, the base and commit form the two versions. We also provide the expected number of *Move Class* or *Rename Package* refactorings to be found, as well as the true and false positives and the false negatives in the results of RefactoringMiner. Lastly, we present the precision and recall calculated from these figures.

The data in Table 2.9, presenting the results for *Move Class*, show that of four data points the results are significantly different from the results of the other items in terms of recall. The excellent results in precision, cannot cover the instability of recall. The number of undetected *Move Class* refactorings in these four abnormal data points, with a recall of less than 60%, is 44, accounting for 14% of the total number of refactorings. (Note that the original authors reported 100% precision and 96.24% recall for *Move Class* .) In two

TABLE 2.10: Detailed accuracy results for RefactoringMiner and the *Rename Package* refactoring type.

Number	commit	Expect Result	True positive	False positive	False negative	Precision	Recall
1101310	sonarqube abbf325	2	1	0	0	1 1.000 	0.500 
1101296	sonarqube 4a2247c	2	0	0	0	2 N/A	0.000 
1123966	spring-data-neo4j 071588a	27	2	0	0	25 1.000 	0.074 
1125333	facebook-android-sdke 813a0b	8	0	0	0	8 N/A	0.000 
1134096	hibernate-orm 44a02e5	3	2	0	0	1 1.000 	0.667 
1136729	reactor 669b96c	2	0	0	0	2 N/A	0.000 
1140316	aws-sdk-java 14593c6	3	0	0	0	3 N/A	0.000 
1142116	infinispan 8f446b6	7	2	0	0	5 1.000 	0.286 
1157300	android c976598	35	1	0	0	34 1.000 	0.029 
Total		89	8	0	0	81 1.000 	0.090 

out of the 14 commits, RefactoringMiner did not report any results for *Move Class*, which means that we cannot compute the individual precision for these cases.

This is similar for the results for *Rename Package*, found in Table 2.10. In four out of nine cases, no results were reported at all and we could not compute the precision. For the remaining commits, the precision was at 100%. In terms of recall, from the total 9 samples, RefactoringMiner has a recall below 50% in 7 cases and even could not find any true positives in 4 cases. The number of false negatives is in total at 91%. In the original evaluation of the RefactoringMiner authors, a precision of 85% and a recall of 100% had been reported for *Rename Package*.

When investigating the results further, we found that false negatives for *Move Class* were often falsely classified as *Rename Package* and vice versa. Therefore, we believe that RefactoringMiner easily confuses these two refactorings.

2.4.5 Distinguishing *Move Class* and *Rename Package*

To understand this, let us start by looking at the definition of the two types of refactoring.

Move Class: Move a class to another package. The class's simple name is not changed, but the file is moved to a different path and the package statement is changed. The contents of the class are unchanged.

Rename Package: Rename a package. All Java-files contained in the path of the original package are moved to the path corresponding to the new package name. The simple class names stay the same, i.e., the contents of the package does not change, except for the package statement which now uses the changed package name.

The descriptions of the two refactoring types are very similar, namely files are moved to a different path, package statements change and classes stay otherwise the same. An essential difference is that in one case only one Java-file is moved and the rest of the classes in a package stay untouched, and in the other case, all Java-files and resources are moved to a new path. Apparently, RefactoringMiner has difficulties recognizing this difference.

We examined the inner workings of the RefactoringMiner to further understand this. It simply analyzes the difference between two code versions

using the diff-feature of Git. Thus, it does not need to perform a comprehensive matching screening for all the code of the two project versions.

Figures 2.2 and 2.3 show the diff output for two samples in our benchmark in the side-by-side view of Bitbucket. The refactoring in Figure 2.2 is reported as *Move Class* by RefactoringMiner and actually really shows a *Move Class*. The refactoring in Figure 2.3 is also reported to be *Move Class*, however, this time actually a *Rename Package* refactoring had been performed. We can see that the diff only shows the changed file names and package statements. In both cases, the structure of the diff is identical and it is impossible to judge which of the refactorings has been applied by using only this data.

```
2 ■■■■■ ...oifs/filesystem/DocumentOutputStream.java → ...oifs/filesystem/DocumentOutputStream.java
18 - package org.apache.poi.poifs.filesystem; 18 + package org.docx4j.org.apache.poi.poifs.filesystem;
```

FIGURE 2.2: Example diff for a *Move Class* refactoring.

```
2 ■■■■■ .../squareup/okhttp/internal/spdy/Hpack.java → ...squareup/okhttp/internal/framed/Hpack.java
16 - package com.squareup.okhttp.internal.spdy; 16 + package com.squareup.okhttp.internal.framed;
```

FIGURE 2.3: Example diff for a *Rename Package* refactoring.

This is not surprising if we recall the definition of the two refactorings, which both include the step of moving a Java-file and changing its package statement while leaving the file otherwise untouched. The difference lies within the context in which the moved file appears: For *Move Class*, the class is moved to a package (and thus, the Java-file is moved to a path), which already existed before. Except for this one class, the contents of the original package is left unchanged, and the package will not disappear after the refactoring. For *Rename Package*, the target package (and this path) did not exist before and the old package disappears.

However, this required context information is not visible in the diff, which is focused on showing differences in file contents. Whether a directory was newly created or disappeared in a commit, is not reflected. Likewise, unchanged contents are not reported, i.e., it cannot be seen if a directory contains other files than the ones that changed.

2.5 How to choose a refactoring tool

There are three aspects to consider when choosing a suitable detecting tool:

1. Choose the connection method. (Choose an API repositories database or a two-version comparison based on the IDE plugin)
2. Consider the type of refactoring. Determine whether the detection tool covers the refactoring type that needs to be detected.
3. Choose tools with high precision and recall and, if the test items are large, consider the execution time.

The way to use API to connect repository is more suitable for refactoring detection. Github's presence allows APIs to be used in conjunction with repositories, allowing for quick downloads and detection, effectively reducing operating costs. IDE-based plug-ins are the easiest to understand when there is no code management platform, but importing the detected items into the IDE requires a lot of time (especially for large projects). However, a study (Michele Tufano, 2017) showed that with two fully constructed software system versions as input detection refactoring, only 38% of software system change histories can be successfully compiled. This is a serious limitation on the detection of refactoring history, and it also poses a threat to the effectiveness of related experiments. (In the Section 3 experiment, related experiments were also performed to download the commit code and its parent commit in Github as input to the system version. However, Ref-Finder and RefactoringCrawler cannot detect its reconstruction due to the destruction of external dependencies.) Therefore, the API (RM, RD) method is more advantageous when selecting the access method, but the specific selection needs to be based on the actual situation of the detected object.

Tools that support multiple types of refactoring should be prioritized. The detection tool needs to cover all refactoring types that are detected, but this condition is not always met, so it is necessary to choose a tool that can cover a relatively large number of refactoring types. In order to pursue the practicality of this feature, the detected refactoring should be a type that developers or researchers often use, and often appear in various example code. Among them, GitHub as an open source code base and version control system, many developers put the program on the cloud, making it a hotbed of refactoring function applications. January 4th, 2017 study on "What are the most common refactoring operations performed by GitHub developers?" ("[Data](#)") shows that *Rename* is the most popular refactoring type(77%) in the approximately 1.1 million JAVA project commits, followed by *Move Class/Method* (13%), then *Extract* (9%), and the remaining 1% is the sum of many other refactoring types. The specific 12 types of refactoring are sorted as follows: *Rename variable*, *Rename method*, *Rename class*, *Move method*, *Move class*, *Extract method*, *Inline method*, *Extract class*, *Extract interface*, *Extract superclass*, *Pull up method*, *Push down method*. It can be seen that the four detection tools correspond to: RM has 10 types of intersections, RD has 9 types of intersections, RC has 5 types of intersections, and RF has 9 types of intersections (K. Prete, 2010b). In summary, the refactoring types of the experiment are determined as the intersection type of each tool and statistical results. Because the detection of refactoring type selection needs to correspond to the actual application situation, if these common refactoring types cannot be effectively detected, the utility of such a tool will be greatly reduced. In contrast, RM, RD and RF have more advantages in this regard.

Detection accuracy and execution time are the primary criteria for selecting detection tools. In question 2, it has been explained that RM and RD have high precision and recall in the four detection tools, and are also more competitive in common refactoring type detection coverage. Moreover, these two tools are the link mode of the API outreach repositories database, especially

in the RM, the interface can also be used to link the versions of the two software systems. In summary, RM is the best of the current detection refactoring tools, and RD is also very competitive.

2.6 Conclusion

This dissertation is an overview of the refactoring detection method. According to the time axis, the development process of the detection refactoring technology from theory to application is introduced. From the emergence of the concept of refactoring, to the introduction of the theory of reverse engineering detection refactoring, to the emergence of various detection refactoring theories, to the emergence of various detection refactoring tools, the article briefly introduces the development history of detection refactoring theory. From concept to theory to tool development, detection refactoring became mature in 20 years and proposed some problems that readers may be concerned about. For these problems, the experiments are designed to meticulously test and compare the four detection refactoring tools. The experimental results illustrate the stability and difference of the detection performance of the tools. Through detailed analysis, we found many problems in the original author's papers, and also confirmed the special phenomena mentioned in some recent research results. The whole experiment evaluates the detection and testing tools from the aspects of refactoring type, precision, recall, execution time. Through the experimental results between the first group and the second group, the preset questions are answered, and the threshold question, credibility and selection tool method are discussed, our views and opinions are attached and the corresponding suggestions are given for the reader to select the detection tool.

Analysis and solution of experimental problems Since the experimental sample is a complex code commit, the detection of individual samples can cause the result to be abnormal due to some factors, thereby affecting the entire type of precision and recall. This situation is caused by:

1. There is a deficiency in this type of algorithm, and no more comprehensive and complicated cases are considered;
2. The sample code may be abnormal and cannot correspond to TruePosition;
3. There is no weighted average so that the number of refactoring has a large impact on the final reconstruction results.

In order to avoid this situation, the second test can be performed. For example, in the case of *MoveClass*, RM detection is used first, and then the RD test is used to ensure the accuracy of the result. However, for the case where RM detects the occurrence of *RenamePackage*, it indicates that the matching algorithm of the tool has a problem in the screening, which is not a case. A large

number of such refactorings are not detected, so it is necessary to continue to improve the related algorithms.

For the two refactorings *Move Class* and *Rename Package*, we measured values for precision and recall which significantly diverge from all other supported refactorings, as well as from the evaluations presented in the literature so far.

Therefore, we further investigated the approach of RefactoringMiner for these two cases and found that the implementation approach of using only the diff of two code versions on Git hinders the proper detection of *Move Class* and *Rename Package*. The reason is that the difference of these two refactorings lies within the content which did *not* change, and this is not shown in the diff. For this reason, RefactoringMiner frequently confuses these two refactorings. We, thus, conclude that analyzing the diff provided by a version control system such as Git is powerful for detecting refactorings that do not depend on the context in which they appear.

If the refactorings *Move Class* and *Rename Package* would be disregarded, RefactoringMiner would even have a precision and recall of 94% and 98%, respectively, in the evaluation shown in this chapter. Since the information required to distinguish between *Move Class* and *Rename Package* (namely which directories have been created or deleted during a commit) could be easily obtained, we conclude that RefactoringMiner—with a simple extension—could in principle reach this high level of accuracy.

Future expectations The development of detection refactoring technology is a new starting. Now the detection theory is more perfect, the detection technology is more diverse, and the detection method is simpler. However, in some aspects, further improvement is needed. The detection content cannot be only for Java. The detection refactoring type needs to continue to increase (From 2013.12.10 Fowler has released 91 refactoring types (“[Catalog of Refactorings](#)”)), and the matching algorithm needs to continue to improve. The detection algorithm needs to be more comprehensive and the operation method needs to be more convenient. Through long-term research, it is found that the use of detection tools needs to be more simplified. It is our initial idea to greatly reduce the time for setting up the detection environment. It is a trend to realize online detection in the future. That is to say, no matter who is, whether there is a programming basis, the online function can be used freely, and the online function can realize detection of various access methods such as system version, code fragment, and submitting database in the code management platform. It avoids the need for Eclipse environment construction, code sorting, interface setting, commit replacement and other work, greatly reducing the impact of human operation on the test results, but also increases the scope of application, reducing the requirements on machinery and equipment, reaching an ideal detection environment.

Chapter 3

Probability Model for Nested Refactoring

3.1 Detecting Problems with Nested Refactoring

In this chapter, we will introduce the nesting mechanism of nested refactoring and how it differs from a single refactoring type, and then outline my research on nested refactoring.

3.1.1 Nested Refactoring

As shown in the Figure 1.2, Refactoring1 can be *Extract Method*, and Refactoring2 can be *Move Method*. First extract a method (Method'), and then the extracted method is moved to another class (Class''), and then the changes are submitted to the version control system. It was supposed to submit two single refactorings, but here is a submission that nests the two refactoring types. The class diagram change process of this process is shown in Figure 3.1 and Figure 3.2.

In Figure 3.1, it is possible to clearly determine which refactoring type is used, but in Figure 3.2 we cannot directly determine the type of refactoring used, so we use traditional tools can occur omissions or misjudgments when detecting nested refactorings. As shown in the Figure 3.3, the diff only shows the added parts and removed parts of the code, different from the normal refactoring diff, it can be found that the nested diff does not fully reflect the code change process, which is also the key reason why it is difficult to be detected. Comparing the two diffs, we can find that although some of the conditions for judging the refactoring types are hidden during the nesting process, some characteristics of the two refactoring types are still retained in the diff. These retained characteristics are an important basis for us to detect nested refactoring.

3.1.2 Problem Solution

Nested refactoring is discovered with the application of refactoring. Research on detecting nested refactoring is necessary, because in the process of research, not only how to detect nested refactorings, but also the compatibility of refactoring types with each other needs to be studied. On the one hand,

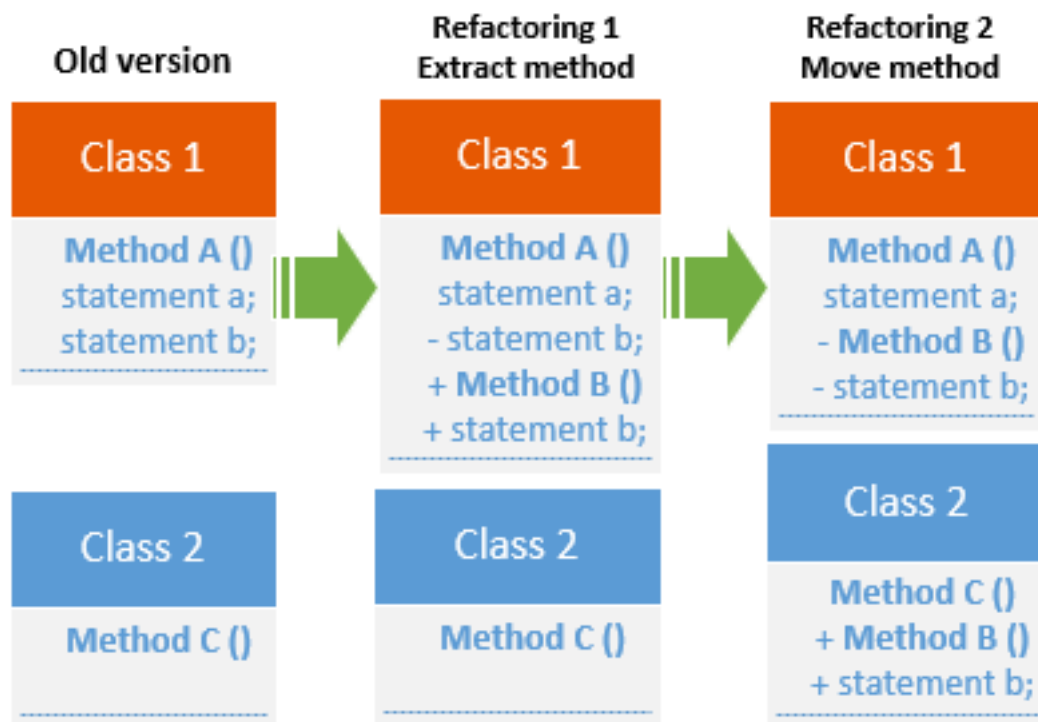


FIGURE 3.1: Normal Refactoring

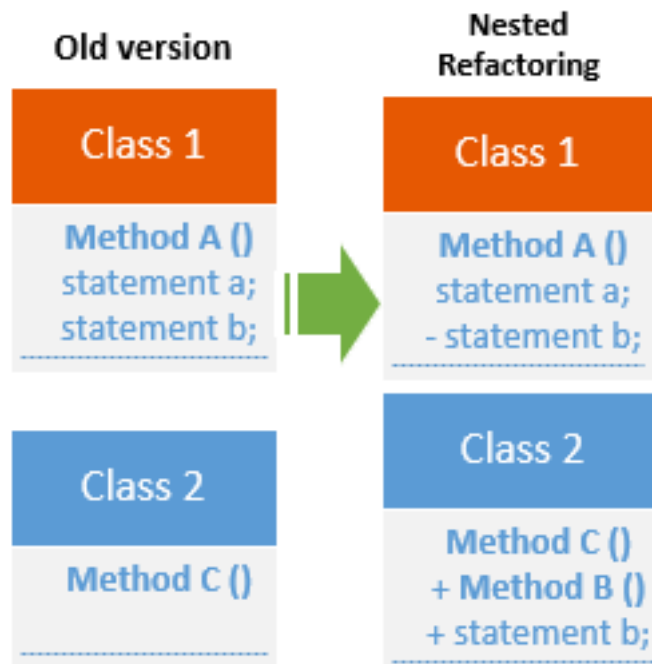


FIGURE 3.2: Nested Refactoring

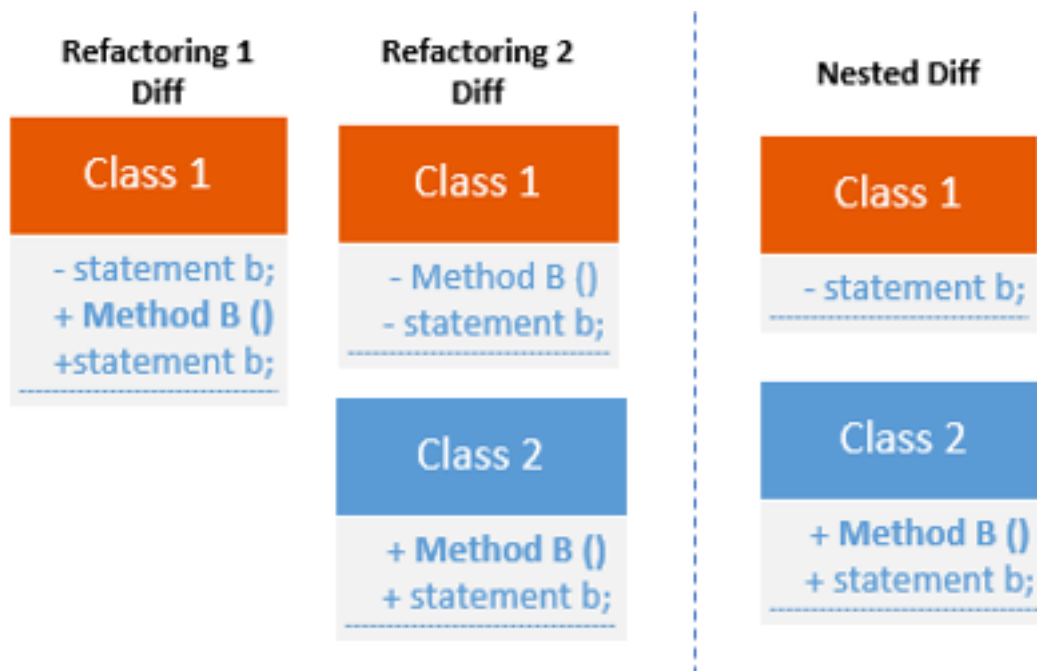


FIGURE 3.3: Diff

this research fills the technical gap that refactoring detection approaches cannot efficiently detect nested refactorings, on the other hand, the research is an important part of refactoring data mining.

In this chapter, we focus on 10 different common¹ refactoring types (the data sample only contains 10 refactoring types), with 35 nested refactoring types (it is a combination of the ten refactoring types included in the sample set). The goal of our work presented in this chapter is to provide a refactoring detection mechanism that is also able to handle nested refactorings without the need to provide a specification for each combination separately. Our algorithm uses the idea of a statistical probability theory to solve the refactoring detection problem, that is, the statistical quantity of various features contained in each refactoring type sample, and calculate the corresponding probability relationship between them, so as to achieve the purpose of detecting refactoring. Thereby, the algorithm is applied to a code change and the classification corresponds to the refactoring detected in the change. We use an algorithm that assigns probabilities to possible classifications and that allows us to inspect them. If multiple refactorings have a high probability, we assume that each of them has been applied in a nested refactoring. This approach works well for nested refactorings with sequences of two refactorings. However, the approach of existing refactoring detection tools does not scale and therefore cannot successfully find all refactorings when nested refactorings happened (T. Kamiya, 2002; Emerson Murphy-Hill Max S Danny Dig, 2014).

¹See a survey about most refactoring types: <https://medium.com/@aserg.ufmg/what-are-the-most-common-refactorings-performed-by-github-developers-896b0db96d9d>

3.2 Probability modeling based on random forest

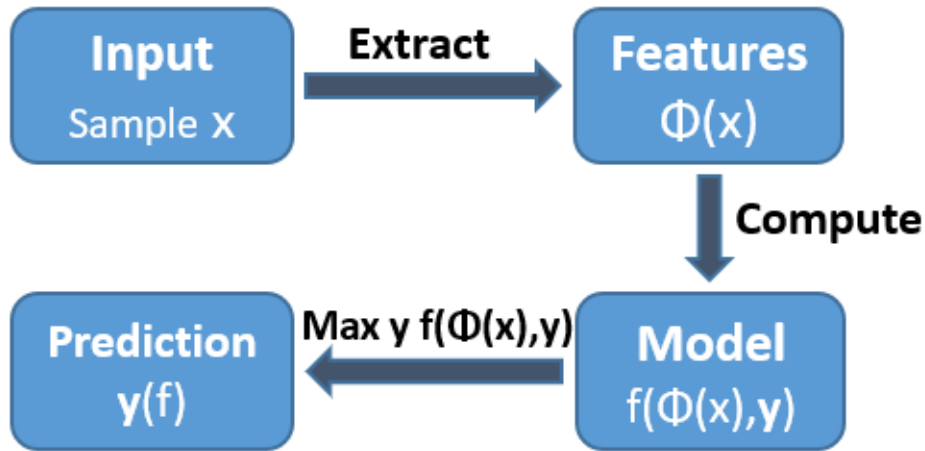


FIGURE 3.4: Algorithms Idea

3.2.1 Theory of Probability Model

Our approach is to use features to calculate condition probability and probability distributions, build probability model (Marco Tulio Valente, 2017; Feller, 1968; Emerson Murphy-Hill Max S Danny Dig, 2014; Friedman J. Olshen R. Breiman, 1984; Breiman, 1996; Breiman, 2001; J. Brant W. Opdyke M. Fowler, 1999; Quinlan, 1989; Quinlan, 1993), and then use probability model to detect decomposition. The functional relationship between the variables is obtained through mathematical statistics.

A probability model is a mathematical representation of a random phenomenon. It is defined by its sample space, events within the sample space, and probabilities associated with each event. The sample space for a probability model is the set of all possible outcomes.

As shown in Figure 3.4, x is the trained sample, y is the predicted refactoring type, $\phi(x)$ is the feature extraction function, and the return of $\phi(x)$ is the defined features $[\phi_1(x), \phi_2(x), \dots, \phi_n(x)]$, $f(\phi(x), y)$ is the function to compute the conditional probability of the feature $\phi(x)$ and the predicted refactoring type y . Our approach is to extract the features $\phi(x)$ from the sample x and calculate the conditional probability of each y when each feature $\phi(x)$ appears, the probability distribution of each refactoring type corresponding to each feature is the probability model, then we use the model to predict the expected refactoring type.

We can consider the whole process of refactoring detection as a probability problem to be determined by probability model. Before describing our approach in more detail, we need to describe a few terms:

Sample : A commit that comes from a version-control system, in our case Github, and the commit contains the commit number and the diff between the code version before and after the refactoring.

Feature : A trivially observable characteristic in the code change, such as a change in the package declaration or the removal of a method. According to the training sample set, we have defined 18 features.

```
+ private void showStatus(String txt) {
    Log.i(TAG, txt);
    TextView tv = new TextView(this);
    tv.setText(txt);
}
```

FIGURE 3.5: Manual Extracting

```
- JcaPEMKeyConverter converter = new JcaPEMKeyConverter();
- keys.add(new KeyPair(converter.getPublicKey(keyPair.getPublicKeyInfo()), null));
+ private KeyPair convertPemKeyPair(PemKeyPair pemKeyPair) throws PEMException {
+     JcaPEMKeyConverter converter = new JcaPEMKeyConverter();
+     return new KeyPair(converter.getPublicKey(pemKeyPair.getPublicKeyInfo()), null);
+ }
```

FIGURE 3.6: Extracting by refactoring function

The main reference for our feature selection is package, class, file and path changes. Features should be easy to distinguish, however in a diff statement changes are the most variable and indistinguishable part, therefore, we do not distinguish changes at the statement level. As an example, consider Figure 3.5 and Figure 3.6, which both show an *Extract Method*, but they look very differently. Extract Method refactoring lets you take a code fragment that can be grouped, move it into a separated method. Figure 3.5 shows the extracted code by refactoring function of IDE, so we can see that the statement lines have moved to a new location, which is described by removing and adding in git; Figure 3.6 is manual refactoring, directly write a new method top of the statement lines that needs to be extracted, so that there will be no changes to the statement lines, so the structure of the diff only differs at the statement-level. Therefore, when not considering lines only containing statements, the similarity of the two diffs is increased. A downside of this choice is that some refactorings cannot be detected, which only impose changes at the statement level, such as **Consolidate Conditional Expression**.

We extract features from three aspects of diff: *object feature*, *scope feature* and *complement features*, each feature is an objective description of the content in the diff, the following is the description and naming of these features. We have also provided short names for these features, which will be used in the tables in this article.

1. Package-Level

- *Package Name Changed* (Pkg_{chg}): The package name is changed in the diff. The new package name is in added part, the old name is in removed part.

2. Class-Level

- *New Class (Cls_{new})*: A class is inserted and the class is newly created. We find that in the diff the class only exists in the "after version" within added part.
- *Add Class (Cls_{add})*: A class is added. But this class is not a newly created class. In contrast to New Class, we find that the class is moved in the diff, which appears within added part and removed part.
- *Remove Class (Cls_{rem})*: A class is removed. we find that the class is moved in the diff, which appears within added part and removed part.
- *New Interface (Ifc_{new})*: An interface is inserted and the interface is newly created, same as New Class.

3. Method-Level

- *New Method (Mth_{new})*: A method is inserted and the method is newly created, same as New Class.
- *Delete Method (Mth_{del})*: A method is deleted. Unlike remove, we find that the method only displays within removed part in the diff, means that the method is completely deleted instead of being moved, so we call this feature Delete Method.
- *Add Method (Mth_{add})*: A method is added. Same as Add Class.
- *Remove Method (Mth_{rem})*: A method is removed. Same as Remove Class.
- *Add Attribute (Att_{add})*: An attribute(field) is added. Same as Add Class.
- *Remove Attribute (Att_{rem})*: An attribute(field) is removed. Same as Remove Class.

4. Package-scope

- *Same Package (Pkg_{same})*: Refactoring affects in the same package. This feature is determined according to the scope of refactored influence. When the refactored influence only in the one package, Same Package is confirmed.
- *Different Package (Pkg_{diff})*: Refactoring affects in different packages. The refactored influence is in two or more packages, Different Package is confirmed.

5. Class-scope

- *Same Class (Cls_{same})*: Refactoring affects in the same class. Same as Same Package.

- *Different Class (Cls_{diff})*: Refactoring affects in different classes. Same as Different Package.

6. File-feature

- *Add File ($File_{add}$)*: Within the diff, a file is added.
- *Remove File ($File_{rem}$)*: Within the diff, a file is removed.

7. Path-feature

- *Path Changed ($Path_{chg}$)*: The path of the file where the refactoring is located has changed.

3.3 Algorithm

Our approach consists of two algorithms, the first is to train the feature probability model, and the second is to detect refactoring type.

1. Collecting all features $\phi(x)$ from sample x according to defined features(section 3.2). Sample x contains element information of packages, classes, methods, scope, file and path in diff.
2. Calculate the conditional probability of each feature and complete the probability model. $f(\phi(x), y)$ represents the probability that the refactoring type is y when the feature $\phi(x)$ appears. As shown in Table 3.1, in the training samples, the count N features $\phi(x)$ are extracted, from each refactoring type sample set, the count of extracted features is $C_1, C_2, C_3, \dots, C_n$. According to Algorithm 1, the probability distribution $f(\phi(x), y)$ of feature $\phi(x)$ and each refactoring type can be calculated.

Algorithm 1 Training Probability Model

Input: Diff of 200 refactoring entities

Output: Model of refactoring detection

> @filename (package+class), line, type

> @filename (package+class), line, type

< @filename (package+class), line, type

< @filename (package+class), line, type

foreach $x \in TrainSet$

$\phi(x) \leftarrow x.(Package, Class, Method, Scope, File, Path)$

Figure out $f(\phi(x), y)$

For int $i=1, i \leq n, i++$

do $f(\phi(x), y) \leftarrow Count\phi(x) / Count_{sum}\phi(x)$

model = $Set[f(\phi(x), y)]$

Return model

TABLE 3.1: Count statistics and Probability calculations

	y_a	y_b	y_c	y_z	Total
$\phi(x)$	C_1	C_2	C_3	C_n	N
$f(\phi(x), y)$	C_1/N	C_2/N	C_3/N	C_n/N	1

3. Use the features contained in the detected data to calculate the average probability of each refactoring type, the maximum probability in the refactoring type is the expected type. As shown in Algorithm 2.(Where n is the count of features extracted there)

Expected Refactoring Type =

$$\mathbf{Max}\{Probability_y | y \in RefactoringTypes\}$$

$$Probability_y = \frac{1}{n} \sum_{n=1}^n f(\phi(x), y)$$

Algorithm 2 Detection Refactoring

Input: FeatureSet

Output: Expected Refactoring Type

foreach $\phi(x) \in FeatureSet$ //FeatureSet represents a feature set was extracted from a refactoring where need to be detected.

void DetectingRefactoring ($\phi(x)$)

model = ProbabilityDistributions Set[$f(\phi(x), y)$] //Table 3.2

For $inti = 1, i < n(numberFeature), i++$

$Probability_y = \frac{1}{n} \sum_1^n f(\phi(x), y)$ //Table 3.3

Result = **Max** $Probability_y$

End

NumberFeature represents the count of extracted features from a diff. In algorithm 2, as shown in Table 3.3, the maximum probability value is the expected refactoring type corresponding to these features.

For example, in a diff that contains only a single refactoring type, three features are extracted, and the three features are $\phi_1(x)$, $\phi_2(x)$, and $\phi_3(x)$. Then the refactoring types here are shown in Table 3.3. Then the refactoring types corresponding to these three features are:

- $Probability_{y_a}: \frac{1}{3}(f_{1a} + f_{2a} + f_{3a})$
- $Probability_{y_b}: \frac{1}{3}(f_{1b} + f_{2b} + f_{3b})$
-
- $Probability_{y_z}: \frac{1}{3}(f_{1z} + f_{2z} + f_{3z})$.

The total probability of each refactoring type is 100%. Comparing the probability values in the result, we can get the most likely refactoring type corresponding to the three extracted features.

TABLE 3.2: Probability distributions

	y_a	y_b	y_z
$\phi_1(x)$	$f_{1a}(\phi_1(x), y_a)$	$f_{1b}(\phi_1(x), y_b)$	$f_{1z}(\phi_1(x), y_z)$
$\phi_2(x)$	$f_{2a}(\phi_2(x), y_a)$	$f_{2b}(\phi_2(x), y_b)$	$f_{2z}(\phi_2(x), y_z)$
.....			
$\phi_n(x)$	$f_{na}(\phi_n(x), y_a)$	$f_{nb}(\phi_n(x), y_b)$	$f_{nz}(\phi_n(x), y_z)$

TABLE 3.3: Example

	y_a	y_b	y_z
$\phi_1(x)$	$f_{1a}(\phi_1(x), y_a)$	$f_{1b}(\phi_1(x), y_b)$	$f_{1z}(\phi_1(x), y_z)$
$\phi_2(x)$	$f_{2a}(\phi_2(x), y_a)$	$f_{2b}(\phi_2(x), y_b)$	$f_{2z}(\phi_2(x), y_z)$
$\phi_3(x)$	$f_{3a}(\phi_3(x), y_a)$	$f_{3b}(\phi_3(x), y_b)$	$f_{3z}(\phi_3(x), y_z)$
Result	$\frac{1}{3} \sum_1^3 f_n(\phi_n(x), y_a)$	$\frac{1}{3} \sum_1^3 f_n(\phi_n(x), y_b)$	$\frac{1}{3} \sum_1^3 f_n(\phi_n(x), y_z)$

3.4 Proof of Concept

This section records the whole process of how we train the model and how we test the model, and we summarize the algorithms we use in the process of training and testing. Firstly, we introduce the whole training process in detail, including feature extraction and training model; secondly, we use the model to verify; finally, we evaluate and summarize the whole experiment.

3.4.1 Training

For training and validation in our approach, we use the data set used by Silva et al. (Danilo Silva, 2017) and published on their homepage². This contains 539 samples, and only 10 refactoring types are covered in the data set. For the training, we randomly select 200 commits samples making sure that the training samples cover all 10 refactoring types with 20 samples each. In the following we sometimes abbreviate the refactoring names by building their acronyms to make our tables fit. The refactorings we support in this study are the following:

- Extract Method (EM)
- Inline Method (IM)
- Pull-up Method (PuM)
- Push-down Method (PdM)
- Move Method (MM)
- Move Attribute (Move Field, MA)

²See: <https://aserg-ufmg.github.io/why-we-refactor>

- Move Class (MC)
- Extract Super Class (ESC)
- Extract Interface (EI)
- Rename Package (RP)

We analyze these training samples one by one to determine the refactoring features according to Section 3.2. For illustration, consider Figure 3.7 showing the diff of a sample classified as Rename Package as an example. This contains the following features: 1. The path of the file is changed; 2. The package name is changed; 3. The class does not change.

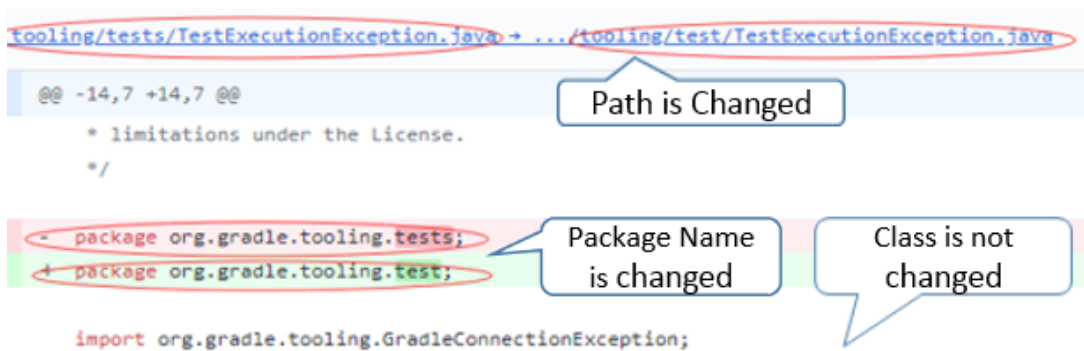


FIGURE 3.7: Example of Rename Package

The sample in Figure 3.8, classified as Extract Method, contains the features: 1. Refactoring takes place in the same package; 2. In the same class; 3. Adds a new method; 4. Removes some statement; 5. The added content is consistent with the removed statement. (Since this training does not consider statement-level features, only the first three features are retained here.)

The Table 3.4 shows the distribution of extracted features in 200 samples, that is, the number of features contained in each refactoring type sample set is counted. Using the Algorithm 1, we calculate the corresponding conditional probability between each refactoring type and each feature one by one, and get the probability model (Table 3.5). For example, when the feature **Package Name Changed** appears, 74.1% is Rename Package, 25.9% is Move Class, and other refactoring types do not include this feature. Subsequent verification and experiments are based on this model.

3.4.2 Validation for Single Refactorings

After training, the model can be used to determine the most likely refactoring type for a commit. Some of the existing refactoring detection approaches use *Similarity* to judge the refactoring type (Figure 3.9). The new algorithm idea is to use the occurrence *Probability* of the feature to judge the refactoring type (Figure 3.10).

The steps to use the approach are as follows, as shown in Figure 3.11:



FIGURE 3.8: Example of Extract Method

TABLE 3.4: Feature summary.

	EM	IM	PuM	PdM	MM	MA	MC	ESC	EI	RP	Total
Pkg _{same}	20	20	20	20	9	13	7	20	20	0	149
Pkg _{diff}	0	0	0	0	11	7	6	0	0	0	24
Ifc _{new}	0	0	0	0	0	0	0	0	20	0	20
Cls _{same}	20	20	1	0	1	1	7	0	0	20	70
Cls _{diff}	0	0	19	20	19	19	13	0	0	0	90
Pkg _{chg}	0	0	0	0	0	0	7	0	0	20	27
Cls _{new}	0	0	0	0	0	0	0	20	0	0	20
Cls _{add}	0	0	0	0	0	0	20	0	0	0	20
Cls _{rem}	0	0	0	0	0	0	20	0	0	0	20
Mth _{new}	20	0	0	0	0	0	0	0	0	0	20
Mth _{del}	0	20	0	0	0	0	0	0	0	0	20
Mth _{add}	0	0	20	0	20	0	0	0	0	0	40
Mth _{rem}	0	0	0	20	20	0	0	0	0	0	40
Att _{add}	0	0	0	0	0	20	0	0	0	0	20
Att _{rem}	0	0	0	0	0	20	0	0	0	0	20
File _{add}	0	0	0	0	0	0	3	20	20	0	43
File _{rem}	0	0	0	0	0	0	20	0	0	0	20
Path _{chg}	0	0	0	0	0	0	7	0	0	20	27

TABLE 3.5: The trained probability model.

	EM	IM	PuM	PdM	MM	MA	MC	ESC	EI	RP
Pkg _{same}	13.4%	13.4%	13.4%	13.4%	6.0%	8.7%	4.7%	13.4%	13.4%	0.0%
Pkg _{diff}	0.0%	0.0%	0.0%	0.0%	45.8%	29.2%	25%	0.0%	0.0%	0.0%
Ifc _{new}	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	100.0%	0.0%
Cls _{same}	28.6%	28.6%	1.4%	0.0%	1.4%	1.4%	10.0%	0.0%	0.0%	28.6%
Cls _{diff}	0.0%	0.0%	21.1%	22.2%	21.1%	21.1%	14.4%	0.0%	0.0%	0.0%
Pkg _{chg}	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	25.9%	0.0%	0.0%	74.1%
Cls _{new}	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	100.0%	0.0%	0.0%
Cls _{add}	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	100.0%	0.0%	0.0%	0.0%
Cls _{rem}	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	100.0%	0.0%	0.0%	0.0%
Mth _{new}	100.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
Mth _{del}	0.0%	100.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
Mth _{add}	0.0%	0.0%	50.0%	0.0%	50.0%	0.0%	0.0%	0.0%	0.0%	0.0%
Mth _{rem}	0.0%	0.0%	0.0%	50.0%	50.0%	0.0%	0.0%	0.0%	0.0%	0.0%
Att _{add}	0.0%	0.0%	0.0%	0.0%	0.0%	100.0%	0.0%	0.0%	0.0%	0.0%
Att _{rem}	0.0%	0.0%	0.0%	0.0%	0.0%	100.0%	0.0%	0.0%	0.0%	0.0%
File _{add}	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	7.0%	46.5%	46.5%	0.0%
File _{rem}	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	100.0%	0.0%	0.0%	0.0%
Path _{chg}	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	25.9%	0.0%	0.0%	74.1%

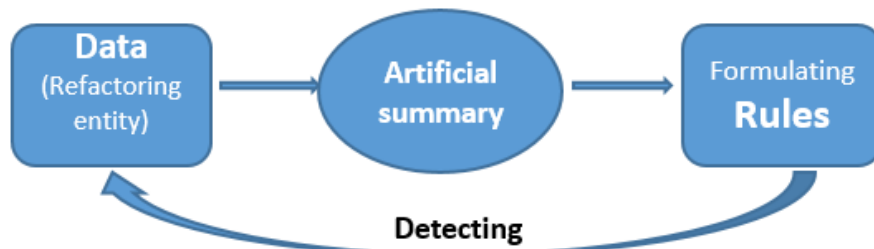


FIGURE 3.9: Traditional Idea of Refactoring Detection



FIGURE 3.10: New Idea of Refactoring Detection

1. Extract all the defined features in the diff.
2. In the trained probability model, find and extract the same features as the first step.
3. Calculate the average probability of each refactoring type and determine the maximum value.

For example, according to the above steps, we have extracted the following features from a sample with an unknown refactoring: 1. Same package, 2. Same class, 3. New a method. Find the corresponding features in the trained model(cf. 3.5), Table 3.6 shows the probabilities according to the three extracted features as well as the average probability of each refactoring type. The highest average probability is 47.3% for the Extract Method refactoring, which is thus reported by our approach when looking for a single refactoring.

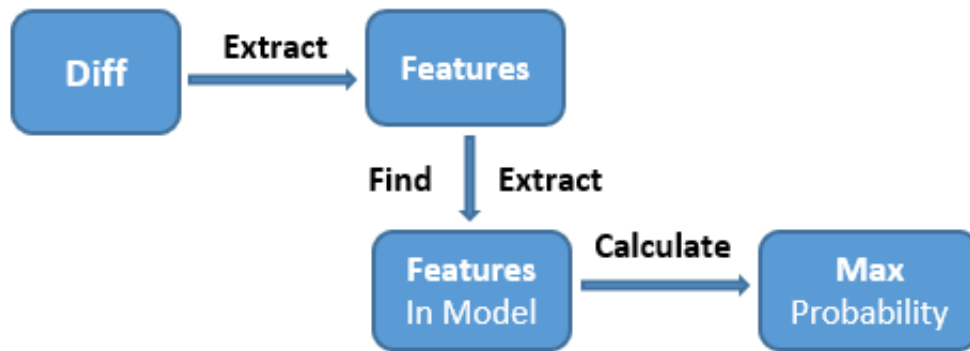


FIGURE 3.11: Process

For validation, we use this approach on 100 samples (the test samples are also derived from 539 sample sets, which are randomly selected and avoid all samples used for training) containing a single refactoring and compared the result of our approach with the previous classification of the samples. In this validation, our model achieved a precision of 98%. This proves that the model we selected for training is effective.

TABLE 3.6: Test result of a single refactoring

	EM	IM	PuM	PdM	MM	MA	MC	ESC	EI	RP
Pkg _{same}	13.4%	13.4%	13.4%	13.4%	6.0%	8.7%	4.7%	13.4%	13.4%	0.0%
Cls _{same}	28.6%	28.6%	1.4%	0.0%	1.4%	1.4%	10.0%	0.0%	0.0%	28.6%
Mth _{new}	100.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
Result	47.3%	14.0%	4.9%	4.5%	2.5%	3.4%	4.9%	4.5%	4.5%	9.5%

3.5 Detecting Nested Refactoring

In the previous section, we have shown that our approach performs at the same level as existing approaches for samples with single refactorings. Now

we will discuss the performance of our approach for detecting nested refactorings. Since samples in existing benchmarks do not contain nested refactorings, we need to create some new samples ourselves. For this purpose, we create samples with nested refactorings of two single refactorings from the 10 refactoring types we trained on.

Not all of refactoring types can be combined in a nesting relationship, because they are not performed on the same kind of program element: For example, Extract Method inserts a method on which the refactoring Push-down Method cannot be applied. Thus we first did an experimental study and found that a subset of 22 most common refactorings for their ability to be nested and already identified 217 possible combinations with two refactorings. The results of this research also show that it is a difficult task to make judgment rules for each nesting possibility. To create these samples, first, we explored the possibility of intermixing these ten refactoring types, and combined them. In order to fully verify the effect of our detection approach on nested refactorings, we set two principles for building a nested refactoring sample: 1. It is technically possible; 2. It complies with the refactoring logic. Besides this, our main purpose is to explore the role of new algorithm ideas in nested refactoring. Following these principles, 35 nested refactoring types can be found with our 10 supported single refactorings. The 10 supported refactoring types are selected because our training set only contains 10 refactoring types, and the trained model also only contains the feature probabilities of these 10 refactoring types. If more refactoring types are trained in the training set, our approach can also support more refactoring types.

3.5.1 Extract Features and Calculation

We apply our approach to all 35 samples as discussed above. For each, we first need to extract the features. For example, in an *ExtractMethod* + *MoveMethod* sample, we can extract the features: Different package, Different class, New method. When applying our model (cf. 3.5) to detect these features, we get the result shown in Table 3.7. From the table, we can find that there are several refactorings with comparatively high probability. Therefore, we can conclude that not one single refactoring was applied, but multiple refactorings are applied at once. The two refactorings with the highest probability are *ExtractMethod*(33.3%) and *Move Method* (22.3%), thus our approach reports finding a nested refactoring comprised of these two refactorings, which are in fact the refactorings we nested in the sample.

TABLE 3.7: Test result of a nested refactoring

	EM	IM	PuM	PdM	MM	MA	MC	ESC	EI	RP
Pkg _{diff}	0.0%	0.0%	0.0%	0.0%	45.8%	29.2%	25.0%	0.0%	0.0%	0.0%
Cls _{diff}	0.0%	0.0%	21.1%	22.2%	21.1%	21.1%	14.4%	0.0%	0.0%	0.0%
Mth _{new}	100.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
Result	33.3%	0.0%	7.0%	7.4%	22.3%	16.8%	13.1%	0.0%	0.0%	0.0%

3.5.2 Results and Analysis

In 26 cases of 35, our approach could find the two correct nested refactoring types. From the remaining 9 cases, we could correctly identify one of the two nested refactorings and only in two cases we could identify none.

We found that in the cases, where we only detect one correct refactoring type, the reason for most errors is the similarity of Move Class and Rename Package. Especially in Move Class when the class that needs to be moved is a single class (When there is only one class in a java file), Move Class and Rename Package contain almost identical features. At the same time the specific weight of the Rename Package corresponding to the Rename Package in the training model is higher than the specific weight of the Move Class.

In the two cases where our approach could not identify any refactoring correctly, we found that there is not only a confusion between Move Class and Rename Package, but also the misjudgment of Pull-up Method and Move Method, Push-down Method and Move Method. We found that Pull-up Method, Push-down Method and Move Method are very similar in with regard to our extracted features. What is more, Pull-up Method and Push-down Method have only three features while Move Method has many features, (e.g., in the scope category, there is a probability for Move Method for four features: Different Package, Same Package, Different Class and Same Class.) Therefore, distinguishing these refactorings is difficult and Pull-up Method and Push-down Method are easily shadowed by Move Method.

3.5.3 Specific Strategy

For the cases, where our approach did not detect all nested refactorings correctly, we analyzed our model and prediction process, looking for a way to further improve our results. Comparing the features of Rename Package and Move Class throughout all experimental data, we found that the extracted features correspond to Move Class for a main class as well as Rename Package. A difference is that the features are only extracted for one class file when Move Class occurs and for multiple classes when Rename Package occurs, as a package commonly contains multiple classes.

Therefore, we add a step of disambiguation between Move Class and Rename Package. When the obtained result shows Move Class and Rename Package, we check how often the features "package name changed" and "path changed" are detected for a sample. If there is only one, we have a high probability that the refactoring actually is Move Class; if these features appear multiple times, we have a high probability that it is a Rename Package.

After statistics, among the 200 training samples, multiple identical package names and path changes occurred in 20 Rename Package samples, but no one feature was found in the Move Class samples. We repeated our evaluation for nested refactorings with this new calculation. With the improved calculation, we could detect 32 of the 35 nested refactoring correctly and for the remaining three samples, we correctly detected one of the two refactorings. Thus, for detected nested refactorings, our precision is at 91.4%.

3.6 Evaluation

Goal. Conceptually, this approach is based on statistical probability theory, using the probability relationship between features and refactoring types to build a probability model, and then using the model to judge and predict single refactoring types and nested refactoring types. Using the refactored diff information, extract the features, and then calculate the possible refactoring type probability according to the model. Our algorithm idea has been verified in the previous section.

Question. In terms of the experimental process, we have defined 18 method-level and above features based on the characteristics of the refactoring types in the training set. According to the detection results, we can find that these features can be used to judge and predict the 10 refactoring types being trained. If statement-level refactoring types need to be trained and tested, then more detailed features can be defined to support the model, provided that a sufficient amount of statement-level refactoring sample is required.

Metric. According to the detection results, when the approach detects these 10 refactoring types, the precision of single refactoring is 98%, and the precision of nested refactoring is 91.4%. The good precision is due to high-quality data. We have manually confirmed every refactoring involved in training, only after confirming that the refactoring type noted in each sample is consistent with the refactoring type used in the code can be used to train the probability model. If there is a problem with the training data, it will directly affect the result. Therefore, unlike existing refactoring detection tools, our approach requires high data quality. Better data means better models and better prediction results.

3.6.1 Threats to Validity

Internal Validity

The dependent variable is the prediction (refactoring type), as shown in Figure 3.12. We predict the refactoring types based on refactoring features that are directly determinable from diff of a commit. We have described objectively how to extract the features. Since the sample data used to verify the detection mechanism we provide is based on 10 refactoring types, the features defined in this article only work for the 10 types that are trained. In order for the mechanism to be applicable to more refactoring types, so the work of calibrating features is open, so feature extraction cannot be performed automatically, therefore, tester bias is a potential threat. But by practicing the training steps and being familiar with our definition of the features, the impact of this bias becomes very limited. In addition, for randomization and random selection, we have chosen a "gold standard" data set used already in other similar studies as a sample pool for training and validation. In this way we have avoided a researcher bias in the selection of the samples.

Sample pretreatment. In the process of training and testing, all the samples we use are committed to include refactoring. These samples belong to the "gold standard", so there is no mention of distinguishing between refactoring and non-refactoring. Because our approach specializes in the part of refactoring detection. But when we need to detect a sample with unknown information, we can use the first part of the RefactoringMiner function (finding refactoring candidates) to complete the screening of the sample. This function can traverse the detected submissions and output the information that refactoring occurred. From the recall of RefactoringMiner, you can find that this function performs well. Before using our approach for detecting, this function can be used to distinguish between refactoring and non-refactoring, and output the refactored sample to be tested, and then we will perform feature extraction on it.

For multiple refactorings. In our research process, we use a single refactoring sample, that is, only one refactoring occurs in a sample. Therefore, the problem of how to distinguish between multiple refactoring and nested refactoring may arise. Their concepts and essence are different. Multiple refactoring is actually a sample that contains two or more refactorings. Their refactoring types can be the same or different, but their refactoring objects are absolutely different. When a sample that contains several refactorings needs to be detected, we will treat it as a series of single refactorings, and the feature extraction one by one can well judge the refactoring type. Specific method: According to the function of finding refactoring candidates of RefactoringMiner, output a candidates list, which contains all refactored objects. When we extract the features of the first refactored object, shield the relevant features of other refactored objects, and so on. Multiple refactoring is different from nested refactoring, each refactoring is not directly related to each other, so it is easy to distinguish them.

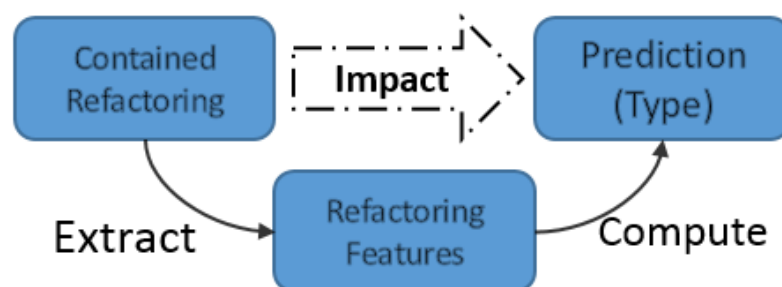


FIGURE 3.12: Variable relationship

External Validity

Repeatability: If the experiment is carried out again by a different person on the same data, then the result is the same. To ensure this, we have an

TABLE 3.8: Framework of Programming Language

Java	Package	Class	Method
C	Head file	Structure	Function
Python	Module	Class	Function

algorithmic description for identifying the features. This description is unambiguous for three reasons: First, all descriptions are based on facts in diff images and do not require additional thinking and evaluation; second, the refactoring features are filtered according to the code structure and characteristics of Java, which is easy to identify; third, the diff on Github contains all the feature information, and the change has a color mark, which is very convenient for locating the feature position.

In order to avoid overfitting, we define refactoring features only down to the method-level and omit changes at the statement-level. By omitting the details of the statement-level, we applied abstraction, which introduces some uncertainty and which in turn reduces overfitting. Furthermore, there is randomness in the sources of the samples in the data set:

1. Project. The sample pool of the 539 commits we used was selected from 748 repositories (Danilo Silva, 2016), many well-known projects, such as JetBrains/intellij-community, apache/cassandra, Elastic/elasticsearch, etc.
2. Contained refactoring. Our training samples cover all refactoring types included in the sample pool.
3. 35 nested refactoring samples. When creating these samples, the process of nested refactoring is random, such as moving a class or method, without designing, but randomly selecting the object being moved and the location of the move.

These attributes are full of randomness, and the selection of samples is also randomly chosen. Such as selection bias and situational factors cannot pose a threat to validity. Instead, we performed multiple calibrations on feature extraction and statistical methods throughout the process to ensure the balance of experimental results.

The external generalizability has been considered at the beginning of the design of the algorithm. While the concrete realization of our approach cannot be universally applicable for all programming languages, the approach itself is language-independent. Since we trained our model using samples using Java code it is only applicable to Java projects. We believe, nevertheless, that our approach can be transferred to other programming languages, which will required the definitions of the features specific to the syntax and structural characteristics of the other programming languages and training a new model.

3.7 Conclusion

This article introduces an approach that can effectively detect nested refactorings, increasing the flexibility and scalability of detection. We have also verified that our approach is feasible and reliable, in the research process of refactoring detection technology, based on the different algorithm ideas, our approach can be used as an aid and supplement to other existing tools, and has made contributions to the development of refactoring detection technology.

Our purpose is to contribute a mechanism that can detect nested refactoring types. The feature definition part of this approach is open, because training and testing only cover 10 refactoring types in this article. This provides users with more options to train models that only detect specific refactoring types. Statement-level features also can be defined to support statement-level refactoring types. We can also collect the change features of the sub- or superclasses, and capture some keywords (including "abstract", "implementation" and "inheritance") as features, this would solving the two false negative results in our evaluation.

In the future work, we will encounter more complex refactoring data mining work, in order to detect more layers (three or more refactoring types are nested) nested refactorings, we need to extract and define more features to improve the ability to recognize different refactoring types. The training and test refactoring types we choose are usually used in some IDEs, but the number of established refactoring types is ever increasing. Therefore, in design, we also consider supporting more refactoring types, using a variety of different refactoring types to train, defining new features, and combining the defined features, we can get a new probability model that supports more refactoring types, thereby supporting the detection of multiple refactoring types.

Chapter 4

Diff Extractor and Diff Encoder

4.1 The Role of Diff in Refactoring Detection

In computing, the utility `diff` is a tool for comparing code data, used to calculate and display differences between the text contents of files. Unlike the concept of edit distance, which is used for other purposes, `diff` is line-oriented rather than character-oriented, as it attempts to determine the smallest set of deletions and insertions to create a file from another file. The utility displays changes in one of several standard formats so that they can be more easily parsed, changed and patched by either a human or a computer. Typically, `diff` is used to display changes between two versions of the same file.

In the context of refactoring, which only changes the internal structure of code and not its external presentation, `diff` is certainly the most intuitive way to identify the impact of refactoring on code. When refactoring detection, code containing diffs is considered the main object of study, as these diffs contain essentially all refactoring information, making them the most critical data in refactoring detection studies.

4.2 Analysis of Refactoring Diff

4.2.1 Classification of Refactoring Diff

We have analysed the diffs of each Java refactoring type and we have found that at the practical level of refactoring, the diffs of each refactoring type can be divided into six categories based on objective changes in the code, which are **Move**, **Rename**, **Remove and substitute**, **Add and substitute**, **Fixed transformation**, **Combine and reorganize**.

- **Move.** The act of refactoring is achieved by moving code. The code that is moved can be a statement, field, method or class. **Move** is usually represented in the diff such that the removed part is the same as the added part. Refactorings in this category: Collapse Hierarchy, Extract Class, Extract Method, Extract Superclass, Move Field, Move Method, Move Statements into Function, Move Statements to Callers, Pull Up Constructor Body, Pull Up Field, Pull Up Method, Push Down Field, Push Down Method, Slide Statements, Extract Variable. Take *Move Method* as an example, as shown in the Figure 4.1 with the diff shown at the right-hand side.



FIGURE 4.1: Move statement Type

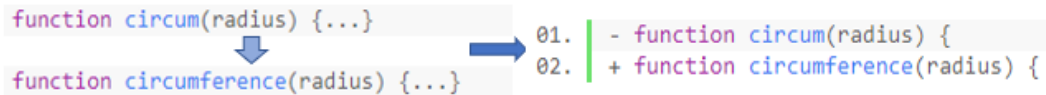


FIGURE 4.2: Rename Type

- **Rename.** Such refactorings involve changing the name of a code element to achieve the purpose of the refactoring. Names that can be changed include the class name, method name, package name and field name. **Rename** is usually represented in a diff such that one line with the removed part is immediately followed by a line with the added part, whereby there is one difference between these two lines. Refactorings in this category: Rename Method, Rename Function, Change Signature, Rename Field, Rename Variable, Split Variable. Take *Rename Method* as an example, as shown in the Figure 4.2.
- **Remove and substitute.** Some refactorings are achieved by removing and replacing statements or elements of statements in the code. Refactorings in this category: Remove Parameter, Remove Dead Code, Remove Setting Method, Remove Subclass, Inline Method, Inline Variable, Remove Middle Man, Preserve Whole Object, Change Reference to Value. Take *Remove Setting Method* as an example, as shown in the Figure 4.3.
- **Add and substitute.** Adding and replacing some new statements or elements of statements is done by some refactoring. Refactorings in this category: Add Parameter, Introduce Assertion, Introduce Parameter Object, Change Value to Reference, Encapsulate Variable. Take *Introduce Assertion* as an example, as shown in the Figure 4.4.



FIGURE 4.3: Remove Type

```

if (this.discountRate)
  base = base - (this.discountRate * base);
  ↓
assert(this.discountRate >= 0);
if (this.discountRate)
  base = base - (this.discountRate * base);
  → 01. + assert(this.discountRate >= 0);

```

FIGURE 4.4: Add Type

```

for (const p of people) {
  if (! found) {
    if ( p === "Don") {
      showAlert();
      found = true;
    }
  }
  ↓
for (const p of people) {
  if ( p === "Don") {
    showAlert();
    break;
  }
}
  → 01. - if (! found) {
     02. - found = true;
     03. + break;

```

FIGURE 4.5: Fixed transformation Type

- Fixed transformation.** In order to improve the structure of the code, specific redundant code structures are streamlined and reshaped by means of fixed optimisations. Refactorings in this category: Decompose Conditional, Encapsulate Collection, Encapsulate Record, Hide Delegate, Remove Flag Argument, Replace Conditional with Polymorphism, Replace Constructor with Factory Function, Replace Derived Variable with Query, Replace Error Code with Exception, Replace Exception with Precheck, Replace Command with Function, Replace Function with Command, Replace Magic Literal, Replace Nested Conditional with Guard Clauses, Replace Parameter with Query, Replace Query with Parameter, Replace Subclass with Delegate, Replace Temp with Query, Replace Type Code with Subclasses, Replace Inline Code with Function Call, Replace Loop with Pipeline, Replace Primitive with Object, Replace Type Code with State/Strategy, Introduce Special Case, Replace Control Flag with Break. Take *Replace Control Flag with Break* as an example, as shown in the Figure 4.5.
- Combine and reorganize.** This improves the readability, systematicity and other performance of code by combining and reorganising code. Refactorings in this category: Split Phase, Split Loop, Return Modified Value, Combine Functions into Transform, Combine Functions into Class, Extract Subclass, Consolidate Conditional Expression, Parameterize Function, Separate Query from Modifier, Substitute Algorithm. Take *Combine Functions into Class* as an example, as shown in the Figure 4.6.

```

function base(aReading) {...}
function taxableCharge(aReading) {...}
function calculateBaseCharge(aReading) {...}

class Reading {
  base() {...}
  taxableCharge() {...}
  calculateBaseCharge() {...}
}

```

↓

```

01. - function base(aReading) {...}
02. - function taxableCharge(aReading) {...}
03. - function calculateBaseCharge(aReading) {...}
04. + class Reading {
05. + base() {...}
06. + taxableCharge() {...}
07. + calculateBaseCharge() {...}
08. + }

```

FIGURE 4.6: Combine Type

```

01. package hello;
02.
03. public class Structure {
04.     static int num = 1;
05.     public static void main(String[] args) {
06.         String str = "refactoring.com";
07.         System.out.println(num);
08.         System.out.println(str);
09.     }
10. }
11. }

```

FIGURE 4.7: Refactoring objects and Code elements

4.2.2 Analysis of Refactoring Diff

An analytical study of each type of refactoring shows that refactoring is aimed at optimising the structure of a program while following the syntax of the programming language. These operations do not change the actual function of the program, do not change values parameters and constants, and do not change branching conditions, and are acts of purely structural improvement. The Java sample code in Fig. 4.7 shows simple Java code that contains 11 different program elements, of which 10 elements that can be refactored; we call these “refactored objects”. Only ⑦ cannot be refactored, and ①②③④⑤⑥⑧⑨⑩⑪ can be refactored. (① Package name, ② Class name, ③ Field name, ④ Constant, ⑤ Method name, ⑥ Parameter, ⑦ String, ⑧ Field, ⑨ Statement, ⑩ Method, ⑪ Class) The relationship between refactored objects and refactoring operations is shown in Table 4.1.

As can be seen, refactoring can be understood as the process of fine-tuning components in the code, which is more concisely and intuitively represented in diff.

TABLE 4.1: Element Operations Table

Move	⑧⑨⑩⑪
Rename	①②③⑤
Remove and substitute	⑥⑧⑨⑩⑪
Add and substitute	⑥⑧⑨⑩⑪
Fixed transformation	④⑥⑧⑨⑩⑪
Combine and reorganize	⑥⑧⑨⑩⑪

4.3 Diff Extractor

4.3.1 Basic Algorithm of Diff Extractor

To analyse the diff, we chose to use RefDiff's parsing and matching algorithm as the base algorithm, which chooses to represent changes in a structured way to reflect the syntax of the code being analysed. But since RefDiff strives for analyzing code written in a variety of languages, the syntax tree can only contain nodes representing code elements that exist in all supported languages. This largest common denominator are classes and methods. The full code of the program element for which a tree node stands is stored as a text property, called the *node code*. The data structure used by RefDiff is called *Code Structure Tree (CST)*.

The flowchart shown in Fig. 4.8 illustrates the workflow followed by RefDiff. Firstly, the source code before and after the change is parsed. The **Before Node Set** and the **After Node Set** contain the nodes of the CST created for these code versions respectively. Secondly, the CST nodes only contained in the Before Node Set are collected in the set **Remove** as they represent the code that has been removed during the code change. Likewise, the set **Add** contains the nodes that only appear in the After Node Set. Finally, RefDiff determines for each pair of nodes in Remove and Add whether they match. Two nodes match if they have the same identifier, the same child nodes, a similarity above the threshold, the same usage or inheritance relationships.

Nodes that match are expected to have participated in a refactoring, where the refactoring was applied to the node from Remove (or its corresponding code element) resulting in the node from Add. To determine the type of refactoring applied, RefDiff further inspects node pairs according to some rules based on information present in the nodes, including the node type, namespace, local name, location, parent node, parameters, etc.

4.3.2 Implementation Algorithm of Diff Extractor

This section describes the implementation the diff extractor and how the extraction of diffs works, summarized by Algorithm 3. The input to the extractor consists of BeforeNode and AfterNode, which are derived from the refactoring candidate node pairs corresponding to each of the RefDiff detection results. The output of the extractor is the diff text contained in the

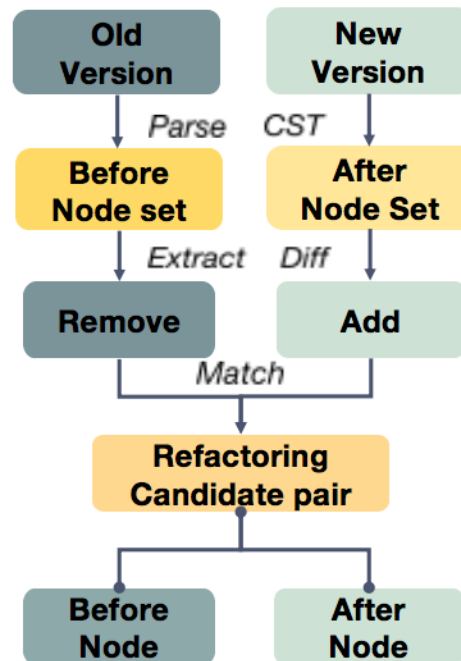


FIGURE 4.8: Prase and Match

BeforeNode and AfterNode code. First, we extracted all the codes in both the BeforeNode and AfterNode nodes and output them as **NodeCode** in a combination of the BeforeNodeCode + AfterNodeCode order. We then used an *ExtractIntersection* function to extract the intersection code of NodeCode and FullDiff (FullDiff is the set of all differences between the old and new versions detected by RefDiff, obtained by Python¹), and the return value of this intersection code is the diff code contained in the two nodes BeforeNode and AfterNode, as shown in the example in Fig. 4.9.

4.4 Diff Encoder

How the diff is input to the deep learning model as data is the most important step. Encoding the diff is the fastest way to implement it, and the encoding approach needs to highlight the behaviour of the refactoring. This section begins by presenting some relevant research on encoding codes and our approach to encoding tailored to refactoring diff.

4.4.1 Related Encoding Approach

Related research on code or text encoding includes Code2Vec, CodeBERT, bag-of-words model (BOW), word vector model (WordEmbedding), etc.

Code2Vec (Uri Alon, 2019) is the transformation of code fragments into fixed-length, continuously distributed vectors that can be used to predict the semantic information of code fragments. By studying this code embedding,

¹subprocess.check_output(['git', 'diff', '-U0', commit_a, commit_b], cwd=repository_path)

Algorithm 3 Diff Extractor**Input** BeforeNode, AfterNode**Output** resultDiff // Diff of detection result**Begin Body**

1. resultDiff $\leftarrow \emptyset$;
2. NodeCode $\leftarrow \emptyset$;
3. NodeCode.add (BeforeNode.code);
4. NodeCode.add (AfterNode.code);
5. resultDiff = ExtractIntersection (NodeCode, FullDiff)
6. return resultDiff;

End Body**Function** ExtractIntersection (NodeCode, FullDiff)nodeDiff $\leftarrow \emptyset$;foreach Line_{nc} \in NodeCode {Line_{diff} \in FullDiff {**if** Line_{nc} = Line_{diff} **then**| nodeDiff.add(Line_{nc})**end**

}}

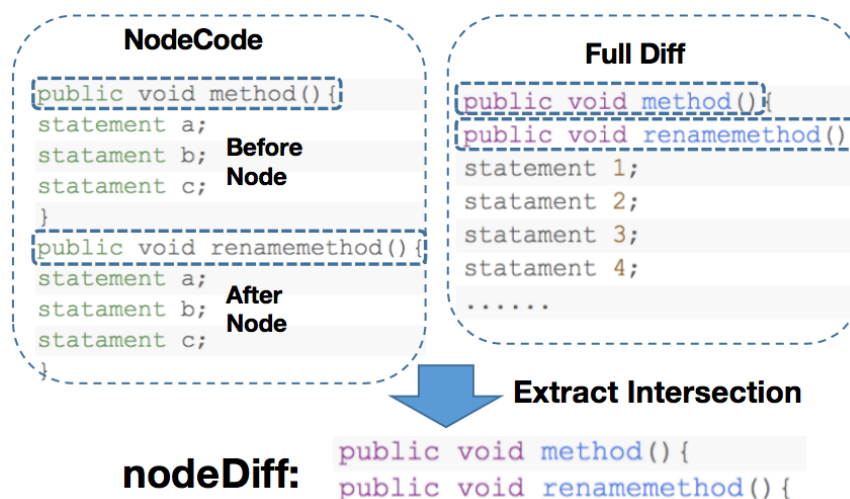
return nodeDiff**End Function**

FIGURE 4.9: Example of the ExtractIntersection algorithm

researchers hope to apply it to various tasks in programming languages, such as generating corresponding semantic labels for code fragments. The difficulty with this approach is that it requires the integration of many expressions and statements in a method body to produce a single label that represents its semantic information. But Code2Vec is not suitable for coding refactoring diffs. Firstly, the range of refactored code elements is very large, from the top-level package to the bottom-level statements in java, and it is difficult to give correct predictions for semantic information with too many or too few diff code fragments. Secondly, even if the prediction of a diff code fragment is correct, there is no way to prove what type of refactoring occurred in the diff, since the external representation of refactoring is constant. Thirdly, refactoring features are our most important requirement, and the main expression of refactoring features is the internal structure change of the code being refactored. Converting diff code fragments into vectors undoubtedly makes the representation of refactoring features complex and abstract.

CodeBERT (Zhangyin Feng, 2020) handles both natural languages (NL) and programming languages (PL), capturing the semantic connections between natural and programming languages and outputting generic representations that can broadly support both NL-PL comprehension tasks (e.g. natural language code search) and generative tasks (e.g. code document generation). The CodeBERT model is built on a multi-layer Transformer and the output of CodeBERT includes: 1. a contextual vector representation of each token (for both natural language and code) 2. an aggregated sequence representation. CodeBERT also does is not suitable to encode refactoring diffs, as its emphasis is on establishing the link between NL and PL through semantic and contextual correspondences. As with Code2Vec, the vectorisation of diffs is not a suitable way to highlight the features of refactoring diffs, because our result checker needs to predict the corresponding refactoring type based on the structural feature of the refactoring diff, rather than having to understand the code semantics.

Models and methods for dealing with natural language such as the bag-of-words model (BOW) (Zisserman, 2003; G. Csurka, 2004), word vector model (WordEmbedding) (Tomas, 2013; Jurafsky Daniel, 2000) etc. are not suitable for direct use in coding diff codes. Keywords in programming languages are used in a very different way to words in natural languages. For example, using the bag-of-words model for encoding, a computer will use the number of occurrences of a word (noun or verb) as the dimension of the word. However, keywords in programming languages (int, for, etc.) occur more frequently and are syntactically necessary but not practically meaningful for our purpose.

In summary, the diff encoder needs to be designed to rely on the features of the refactoring diff, and how to highlight the changing features of the diff structure resulting from the act of refactoring is the most important consideration for the encoder.

4.4.2 Jigsaw Hypothesis

To encode a diff, we first need to consider the many characteristics of the programming language:

- **The programming language contains limited keywords.** For example, there are 32 keywords in C language (for, int, etc.) and above 50 keywords are in Java. Although the programmers can freely choose variable names, function names, etc., these names are only used to distinguish symbols from the computer's point of view, and they do not contain language information, so these myriad variable names and function names can be expressed in a simplified way, this characteristics provides a very ideal premise for encoding, and these limited keywords can also be important features in expressing structural change.
- **The programming language is hierarchically structured.** Examples of the hierarchical structure of Java are: "*package* → *class* → *method* → *statement*", or the data declaration structure: "*DataType Identifier*". The overall structure is clear and fixed, so that the features of the structural changes after refactorings are fixed, and there is not a single refactoring type accompanied by multiple structural changes with different manifestations.
- **The programming language is stable.** Programming languages have strict syntactic rules. The syntax of a programming language is unambiguous and cannot be changed by programmers. So the structure of the code for different refactoring diffs is also stable, which lays the foundation for training the diff feature model.

Combining the characteristics of programming languages and refactoring diffs, based on the definition that a refactoring only changes the internal structure and not the external presentation, we propose a hypothesis to describe the changes to the structure of code by the act of refactoring. "**Jigsaw Hypothesis**": code can be thought of as a complete jigsaw puzzle, and refactoring a piece of code is the process of moving, replacing, adding, deleting, transforming and reorganising pieces of a jigsaw or the whole jigsaw, but the content expressed by the jigsaw remains the same regardless of the changes.

4.4.3 Encoding Approach for Diff

Code To Array

After a long period of research into refactoring diffs, and inspired by the hypothesis of jigsaw puzzles, the process of refactoring diff removing and adding is like extracting pieces from a complete jigsaw puzzle, restructuring them and putting them back together again. The content of the removed and added pieces of the jigsaw in this set hardly changes, only the positional structure of the pieces changes, and even if some pieces unrelated to the refactoring are added, it still does not affect the feature expression of the refactoring-related pieces in the set.

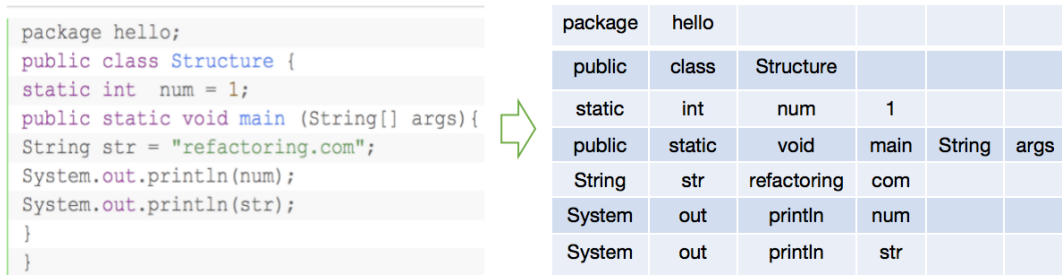


FIGURE 4.10: Putting the code into an array

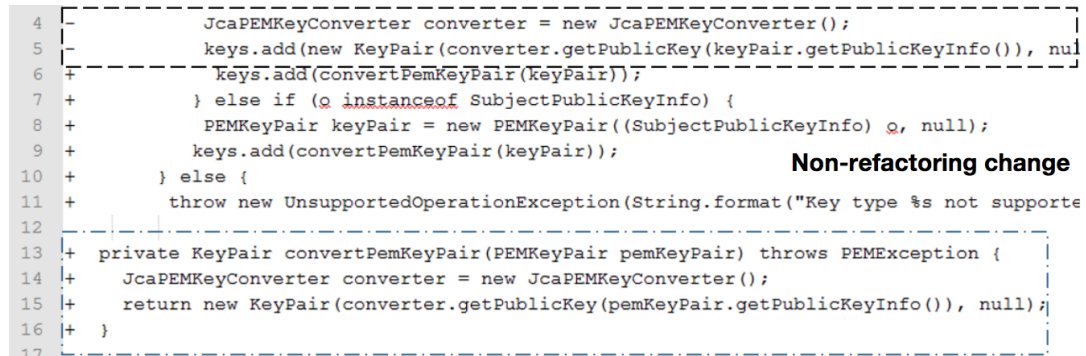


FIGURE 4.11: Example of ExtractMethod

To highlight the impact of refactoring on the structure of the code, we minimise the pieces of the jigsaw by treating each word in the code as a jigsaw piece and putting the jigsaw into an array to obtain a two-dimensional array of words, as shown in Fig. 4.10, which holds the complete structure of the code. The words in the array cannot directly be used for training our model, because user-defined identifier names are always project specific and their number is unlimited, This would lead to an encoding explosion (e.g. one-hot).

Encoding Tokens

In our research experience, when we have used the bare eye to look for refactorings, what matters most is not the exact content and semantics of the refactored diff code, but whether there is a correspondence between the removed part of the diff and the added part, or the changes that each refactoring type causes to the location and structure of the code, therefore, it is no restriction to abstract away exact identifier names. As shown in the *ExtractMethod* example in Fig. 4.11, the refactored part of the code is in the box, the internal sequence of the parts of the code being refactored remains almost unchanged, the relative position of the words contained in each line of code remains the same, and the words in each line of code remain almost the same. In fact, these are the features of a refactoring, and it is the focus of the diff encoder to encode these features in a way that highlights the changes in the structure of the code.

Identifier names can vary in length, but machine learning algorithms expect inputs of a fixed size. We have already argued above that the exact names of identifiers are not relevant for our approach, which is why we can replace identifiers with values of a fixed length. What is relevant, however, is that identical identifiers are represented by the same values. We have observed that programmers rarely change the format of lines (e.g., by adding or removing line breaks) when they are moved during a refactoring. Therefore, single lines are essentially unchanged or just variable names change within the lines during refactoring. So a line of a diff can be seen as a feature, and whatever changes are made to the structure by refactoring, the line feature remains, which is very beneficial for matching networks based on feature overlap.

So we chose a very simple encoding approach that meets these requirements: an array containing the length of each word, as shown in Fig. 4.12. Each column in the array corresponds to a line in the diff, and as can be seen the first and fourth columns, representing an unchanged moved line in of code, are identical. The second and fifth columns, representing a line which is moved and altered slightly, are almost identical. The third column of the array is the method declaration line, containing some keywords that highlight structural changes in the code, indicating the structural changes contained in the diff of the refactoring. Since the array cannot have empty values, we fill the spaces with "0". It is clear from the added and removed parts that there is a correspondence, i.e. the same combination of numbers, or a sorting of numbers, which can be used as features for training. This encoding approach comes from long experience of using the bare eye to determine the type of refactoring. We pay more attention to whether there is a correspondence between the part removed by the diff and the part added, and whether there are syntactic keywords to modify the structural changes in the code, rather than the semantics and functionality of the code. The refactoring diff is encoded in such that it preserves the features of structural changes in the code, and that some of the non-refactoring code changes included in the diff cannot change the presence of these features. The vast majority of refactoring types possess special structural features, which is the main reason why one can discover and define refactorings.

Since the number of lines of code and the number of elements in each line vary, we use the code line containing the most words (n) and the number of lines contained in the diff (k) as the array edge lengths, i.e. each diff is encoded into a $k*n$ array, as shown in Figure 4.13, move type and rename type separately, as shown below in figure 4.14 and figure 4.15.

To demonstrate the feasibility of this encoding approach, we applied it to the RefDiff-supported programming languages C and JavaScript. For example, Fig. 4.16 shows the code and encoding array of *Move Function* in the C context, and the Fig. 4.17 shows the *Inline Function* array in JavaScript. Very distinctive features can also be found in these arrays, which proves that the encoding approach is also applicable to other programming languages.

When refactoring packages, classes or methods in Java, we can notice some unique features because the code that modifies the structure starts with

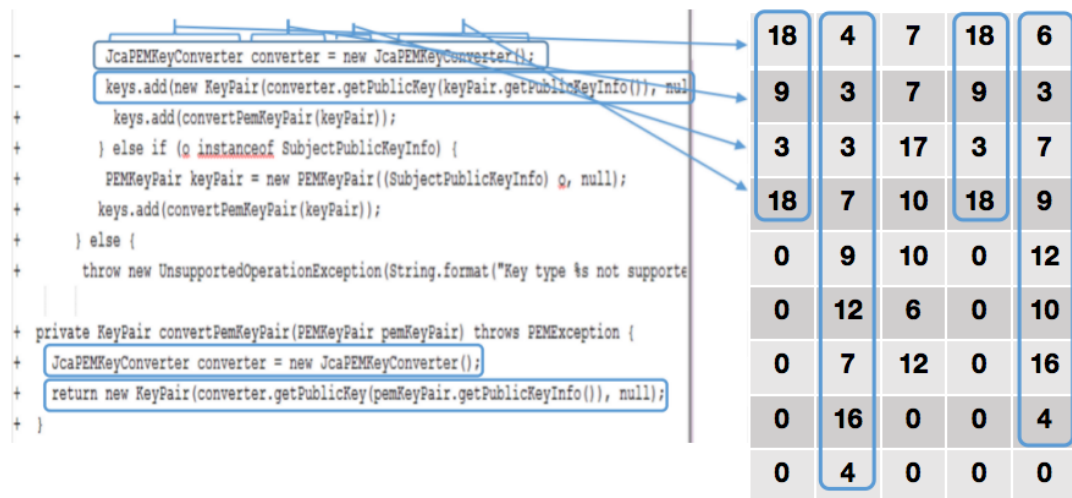


FIGURE 4.12: Example of encode

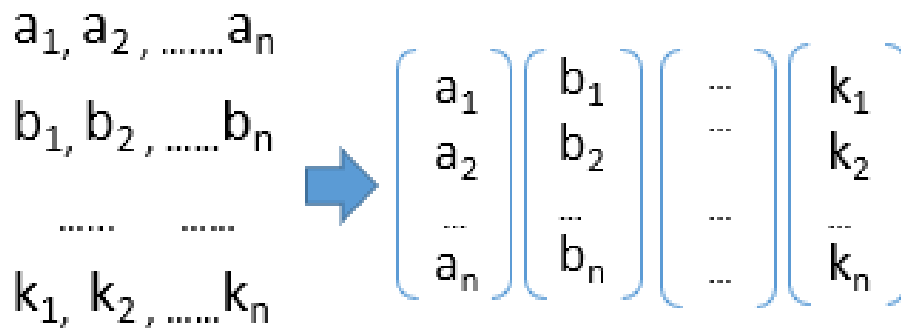


FIGURE 4.13: Encoding to arraying

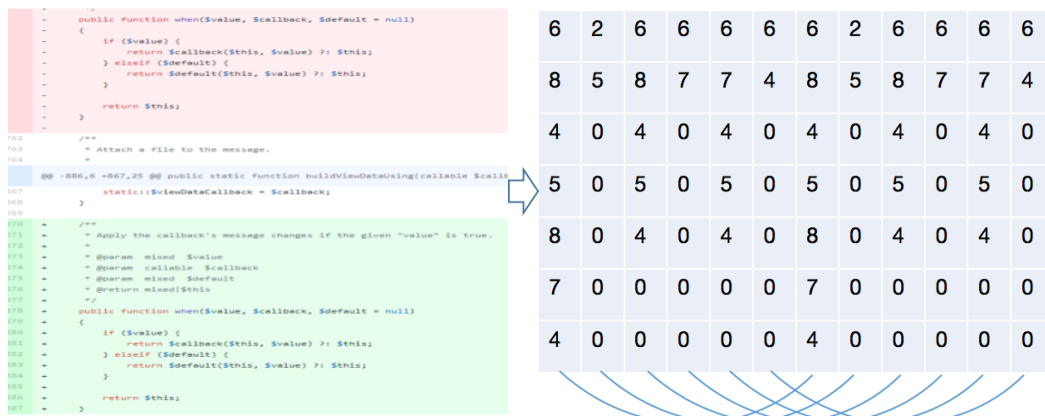


FIGURE 4.14: Example of Move Type

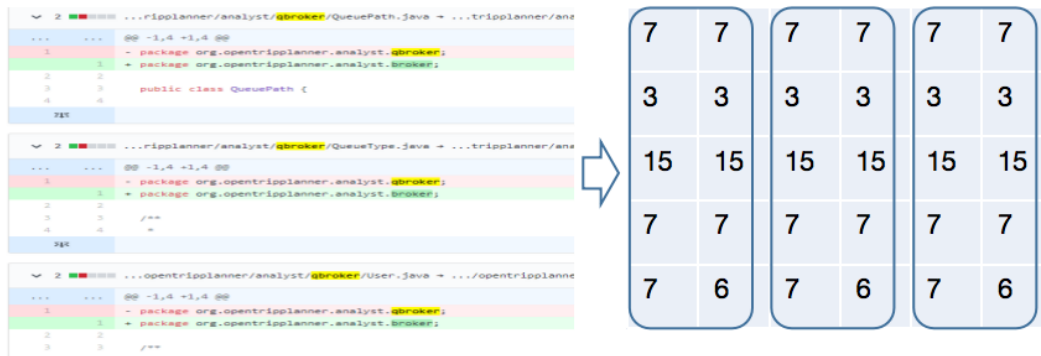


FIGURE 4.15: Example of Rename Type

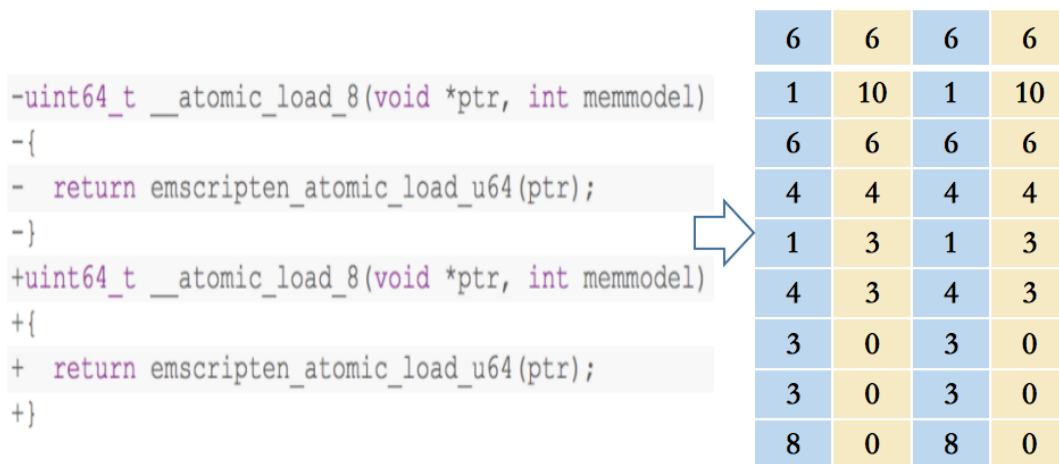


FIGURE 4.16: Example of C



FIGURE 4.17: Example of JavaScript

keywords (package, class, private, public etc.) that keywords are the most intuitive expression of the diff structure. During the encoding process, the keywords of the syntactic structure are given special values to distinguish them from user-defined identifiers or literals. For example, "package" is used as a keyword for declaring package files would be encoded as a "7" according to the word length, which would be the same as "private" or "finally". Also, e.g., variable, method names or values could have the same length and would therefore appear identical in our encoding. To avoid at confusion of keywords with each other or with user-defined names and values, we map all 58 keywords of the Java grammar to unique values. As keywords vary between programming languages, the user can change the assignment to suit the needs of different refactoring detection.

Array To Image

Finally, we convert these arrays into grey-scale images as data for training machine learning algorithms. As diff arrays are not uniform in size, some are small (e.g. RenameMethod) and some are large (e.g. MoveClass), directly resizing the entire array to the size of the largest array in the training array would take up a lot of memory, slow training, and even cause errors. So we first choose to convert the arrays into grey-scale images and then unify the size of these grey-scale images, which can ensure a stable and efficient training process.

Chapter 5

Training Model Base on Diff

This chapter introduces two deep learning models based on diff training, Diff Structure Feature Model and Diff Feature Matching Network. The design intent, deployment, training process and comprehensive evaluation of the models are presented here.

5.1 Solution Problem

5.1.1 Code similarity algorithm

CST (code structure tree) of RefDiff is a tree structure similar to an abstract syntax tree (AST), where the data structure focuses only on the coarse-grained code elements in the code area. Coarse-grained means that only large node types in programming languages are parsed, e.g. classes and methods in java; files, classes and functions in javascript. Building a CST for multiple programming languages requires finding the "greatest common denominator" of each programming language's syntax structure, and one matching algorithm based on this "greatest common denominator" is a prerequisite to support application of multiple programming languages. The source code is parsed to obtain all the structural nodes contained in the code, each containing information such as node name, parent node, relationship, location, node code, etc.

The values used in the matching algorithm to express the similarity of node codes are based on the Term Frequency-Inverse Document Frequency (TF-IDF) and the Jaccard coefficient. TF-IDF (G. Salton, 1986) is a common weighting technique used in information retrieval and data mining that reflects the importance of a word to the documents in a document collection. Jaccard index (F. Chierichetti, 2010), also known as Jaccard similarity coefficient, is used to compare similarities and differences between finite sample sets and is often used to compare text similarity. In fact, RefDiff uses a variation of TF-IDF, where each word and symbol in a node code is first treated as a token, then the weight of each token is derived by IDF, and finally the Jaccard coefficient is used to describe the similarity between two node codes. In the context of a code element, all tokens are treated as terms and the code element is treated as a document. Let E be the set of all node codes and n_t be the number of tokens contained in E . The inverse document frequency (*idf*) is defined as:

$$idf(t) = \log\left(1 + \frac{|E|}{n_t}\right)$$

Once the weight of each token is obtained, the similarity between the two node codes nc_1 and nc_2 is calculated. U is the set of all tokens. m_i be the multiplicity function that represents the multiplicity of the set of tokens for code element nc_i . The following equation defines the similarity between nc_1 and nc_2 :

$$sim(nc_1, nc_2) = \frac{\sum_{t \in U} \min(m_1(t), m_2(t)) \times idf(t)}{\sum_{t \in U} \max(m_1(t), m_2(t)) \times idf(t)}$$

The basic principle of this formula is that similarity is maximal (1.0) when the multiset of tokens representing nc_1 and nc_2 contains the same tokens with the same cardinality. Conversely, if the multiset contains no common tokens, the similarity is 0. A token with a higher idf will have a higher weight. In addition RefDiff has designed *Extract similarity* and *Inline similarity* based on the characteristics of the refactoring type, i.e. only the similarity of the extracted or inlined part of the code is calculated, mainly for the benefit of the *ExtractMethod* and *InlineMethod* refactoring types. For other refactoring types such as Move Type and Rename Type etc., the full node code is used as the matching text.

5.1.2 Analysis

In some cases similarity based on word-frequency theory is not suitable as a key condition for matching refactoring candidates, and RefDiff's 80.4% recall already shows that word-frequency similarity misses about a fifth of refactoring candidates. After our long-term study of refactoring, we found that, in theory, RefDiff would undoubtedly perform very well if the old and new versions of the code were refactored "by the book", but in practice, the old and new versions of diff are very complex, and there are many factors that affect similarity. Below we focus on the two most significant challenges.

Challenge 1: Non-refactored code changes Refactoring is often accompanied by non-refactoring code changes, where the node being refactored has code changes other than those caused by the refactoring.

To compensate this, RefDiff defines a threshold value of 0.5 for similarity. Node pairs with a lower similarity are not further considered. Lowering this threshold further would, however, lead to also including node pairs that do not actually correspond to a refactoring, which would impact the precision. For example, in *ExtractMethod*, the extracted code is put into a new method, but the new method also has a large number of non-refactored code changes added to it, so that the similarity can easily be diluted below the threshold, resulting in some false negatives. So word frequency similarity as an important condition for matching is not sufficient in some cases.

Challenge 2: Noise nodes associated with diff Non-refactored changes exist not only in the node being refactored, but also in other nodes that are not related to the refactoring, and these nodes become noise for refactoring candidates to match. For example, if any changes are made to a node’s code, that node will be put into *Remove* or *Add* to participate in the match, even when the changes are not related to a refactoring. Since similarity matching is computed for all possible pairs of nodes in *Remove* and *Add*, this noise node’s code may coincidentally be similar to another node’s code, increasing the probability of a false positive match, which is the main reason why 3.6 percentage points of RefDiff’s detection results are false positives.

Based on the challenges encountered in RefDiff’s matching algorithm, two different models were designed to address the challenges using the features contained in diff.

5.2 Diff Structure Feature Model

5.2.1 Approach Overview

The Diff Structure Feature Model (Tan Liang, 2022a) is based on the phenomenon that the refactoring diff structure is different for different refactoring types and that this diff structure does not change with the presence of non-refactored code. Based on this phenomenon, we collected ten refactoring types of refactoring diffs as training data and trained a diff structure feature model with checking function.

An overview of our solution approach is presented in Fig. 5.1. The first part is just the regular use of RefDiff: the code change to be analyzed is used as input by RefDiff, which produces a detection result. Our approach is to further process each reported result, for which we extract the **Before Node** and **After Node**, and feed them to our result checker, which performs three steps. The *Diff Extractor* determines the textual differences between the Before Node and the After Node, which is then encoded into a two-dimensional array which correspond to a grey-scale image by the *Diff Encoder*. The image is fed into the refactoring *Diff Structure Feature Model*, which finally produces a **Prediction Result**, i.e., a prediction of the applied refactoring type. Finally, we compare the refactoring type provided by RefDiff’s Detection Result with the one predicted by our Diff Feature Model, and only if they are the same they will be output as the final result.

The purpose of this approach is to identify false positives reported by RefDiff and, thus, increase the precision. But it can also help to increase the recall, when our result checker is combined with lowering the similarity threshold used for candidate selection by RefDiff. Lowering this threshold leads to an increase in the number of considered candidates. The additional candidates are either true positives, which are the ones we actually want to add, as well as false positives as an undesired collateral effect. Thus, the recall is increased and at the same time the precision is reduced. Nevertheless, if we apply our result checker to the now larger set of results reported by

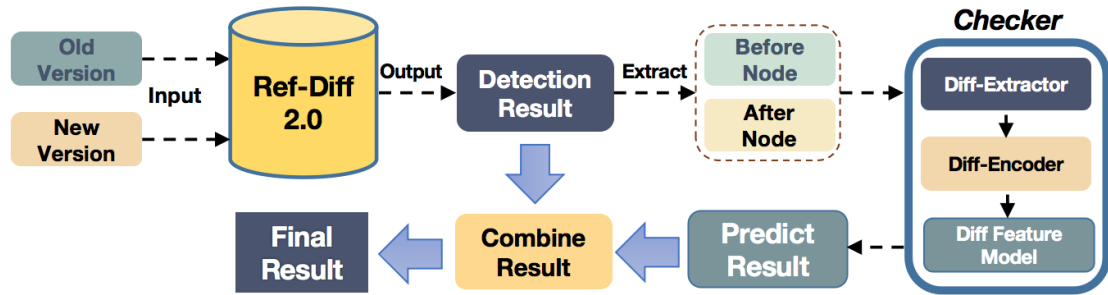


FIGURE 5.1: Approach Flow chart

RefDiff, we can reduce the number of false positives and thereby improve the precision again.

The result checker uses a deep learning model to learn which features in code changes are characteristic for different refactorings. Thereby, we make two assumptions: the refactoring features are not changed by the presence of additional non-refactoring code changes. And the refactoring features are different for each refactoring type. This assumption holds for code diffs, with the exception of PullUpMethod, PushDownMethod and MoveMethod, which have the same diff features, as the textual diff does not allow to distinguish between the parent, child and unrelated other classes as the target for the move.

5.2.2 Training Process

We use Java refactoring examples as training and test data for our diff feature model, mainly because the richest data on refactoring examples is available for Java and because other refactoring detection tools are also optimised for Java. Thus, it will allow us to perform a fair comparison in the end.

Data preparation. We have three sources from which we collect samples for our data set, needed for the training and validation of our machine-learned diff feature model. First, we selected a large number of code changes (in terms of Git commits) from open source projects. Next, we added the data set used by the RefDiff 2.0 authors¹. And finally we added code refactorings (provided as documented commits in a Git repository) from the Tsantalis team².

To ensure the correctness of this data set, we analysed all samples with the original RefDiff 2.0, manually validated the results and only kept the confirmed true positives. In addition, we manually inspected all samples that were rejected by RefDiff 2.0 because of a similarity score below the threshold, again manually determined which of them are true refactorings and added them to our data set.

We split this full data pool into a set of *Training Data* and *Test Data*, i.e. for testing the result checker. The number of samples for each refactoring

¹<https://github.com/aserg-ufmg/RefDiff/blob/master/RefDiff-evaluation/data/java-evaluation/evaluation-data-public.xlsx>

²<https://aserg-ufmg.github.io/why-we-refactor/#/allCommits>

TABLE 5.1: Data Distribution

Refactoring Type	Data Pool	Training Data	Test Data
Extract Interface	330	306	24
Extract Method	850	355	495
Extract Superclass	369	319	50
Inline Method	422	310	112
Move Class	1100	381	719
Move Method	519	314	205
Pull Up Method	361	330	31
Push Down Method	380	342	38
Rename Class	395	308	87
Rename Method	550	360	190
Total	5276	3325	1951

TABLE 5.2: Training Set and Validation Set

Refactoring Type	Training	Validation	Total
Extract Interface	276	30	306
Extract Method	320	35	355
Extract Superclass	288	31	319
Inline Method	279	31	310
Move Class	343	38	381
Move Method	283	31	314
Rename Class	278	30	308
Rename Method	324	36	360
Total	2391	262	2653

type is not balanced, especially the number of ExtractMethod and MoveClass samples far exceed that of other refactoring types. Thus, to avoid overfitting on those refactoring types, we made sure that we used a comparable number of samples (between 300 and 400) for each refactoring type in our training data set. The result is summarized in Table 5.1. The data pool contains a total of 5276 refactoring including 10 different types of refactorings: *Extract Interface*, *Extract Method*, *Extract Superclass*, *Inline Method*, *Move Class*, *Move Method*, *Pull Up Method*, *Push Down Method*, *Rename Class* and *Rename Method*.

Training process. For the training, we input the node codes of the samples in the Training Data into the diff extractor. Next, we input these diff data into the diff encoder to convert them into grey-scale images of the uniform size 50*50*1 and applied supervised deep learning to them. All our samples in the data pool contain a label of the correct refactoring type they contain, which is used as the ground truth during training and later during the evaluation.

The training data is randomly divided into training and validation sets in a ratio of 9:1 to apply a 10-fold cross validation approach during training, as shown in Table 5.2.

During the training phase, we recognized that for the diffs for Pull Up

TABLE 5.3: Parameters of Diff Feature Matching Network

Layer Name	Layer Type	Input Size	Filter Shape	Stride
Conv	C	224*224*1	3*3*3*32	2
Conv	DW	112*112*32	3*3*32	1
Conv	C	112*112*32	1*1*32*64	1
Conv	DW	112*112*64	3*3*64	2
Conv	C	56*56*64	1*1*64*128	1
Conv	DW	56*56*128	3*3*128	1
Conv	C	56*56*128	1*1*128*128	1
Conv	DW	56*56*128	3*3*128	2
Conv	C	28*28*128	1*1*128*256	1
Conv	DW	28*28*256	3*3*256	1
Conv	C	28*28*256	1*1*256*256	1
Conv	DW	28*28*256	3*3*256	2
Conv	C	14*14*256	1*1*256*512	1
5*(Conv +Conv)	DW	14*14*512	3*3*512	1
	C	14*14*512	1*1*512*512	1
Conv	DW	14*14*512	3*3*512	2
Conv	C	7*7*512	1*1*512*1024	1
Conv	DW	7*7*1024	3*3*1024	2
Conv	C	7*7*1024	1*1*1024*1024	1
Pool	Avg	7*7*1024	7*7	1
FC	-	1*1*1024	1024*1000	1
Softmax	-	1*1*1000	Classifier	1

Method/Push Down Method and Move Method are exactly the same in our encoding, the only difference being the position the method is moved to. Therefore, we excluded Pull Up Method and Push Down Method from our training and instead rely on the original RefDiff to distinguish between the three. That means, our result checker will accept results reported as Pull Up Method or Push Down Method if they are recognized as Move Method by our model.

As the diffs are represented as grey-scale images, the image data only is single-channel, other than typical image data which is three-channel. Therefore, we apply a deep separable convolution that does not require fusion of inter-channel information, mobileNetV1 (Andrew G. Howard, 2017) as the training algorithm, the body architecture shows as on Table 5.3.

5.2.3 Model Evaluation

The training results are shown in Fig. 5.2 and Fig. 5.3. The classification accuracy of the mobileNetV1 algorithm converges to 1 for both the training and validation sets in 100 training batches. The loss function (cross-entropy, which measures the similarity between predicted and actual values) curves

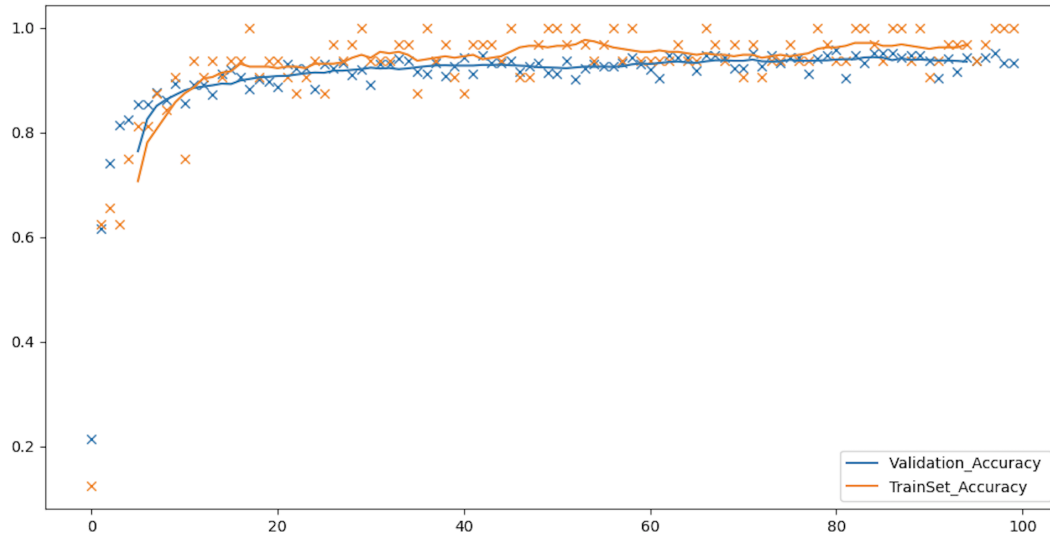


FIGURE 5.2: Accuracy Chart

of the model in the validation and training sets are decreasing and approaching 0, indicating that the model is converging. This is shown in Fig. 5.3. The loss function allows us to assess the fitness of the model in terms of the overall training process. At the start of training, both the training set error and the validation set error are within the underfitting range, and the error decreases with increasing training time and model complexity. After the 20th training batch, the error values begin to stabilise and then make small adjustments to reach a critical point of best fit at approximately 70 batches, before levelling out. The training set error and validation set error continue to remain stable after the best-fit point, indicating that the model is not over-fitted.

To demonstrate that the training results were non-coincidental, we trained the training data 10 times, extracting different data as the training and validation sets each time. For these 10 models, as shown on in Fig. 5.4, we collected the accuracy and error values for their final training and validation sets, with a training set median accuracy of 0.969, validation set median accuracy of 0.948, training set median loss value of 0.035, and validation set median loss value of 0.186. This indicates the reliability of our approach and the resulting models.

To demonstrate the stability of the model, we designed a controlled experiment using a real *ExtractMethod* example. The example has a similarity of 0.339, which is below the threshold determined by RefDiff because it contains some non-refactored code. We then made experiments where we either excluded some of the non-refactoring code changes or added further changes as additional noise. We made sure that the code lines involved in the refactoring remained unchanged. We applied our model to all these different variants of the sample, which recognized the correct refactoring in all cases. For the original sample, our model predicted Extract Method with a score of 0.959 (i.e., high confidence), which increased to 1.0 (absolute certainty) as the non-refactoring code changes were reduced. When adding noise, our model

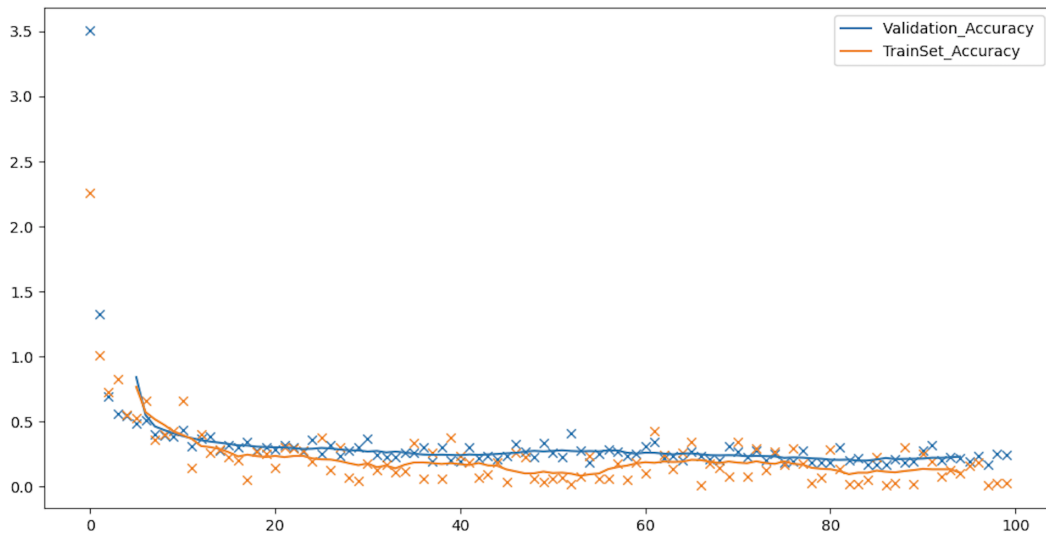


FIGURE 5.3: Loss Value Result

still could make the correct prediction, until the number of lines with non-refactoring code changes increased well beyond the number of changes due to the refactoring. These experiments show that the model is linearly stable and has better robustness than RefDiff’s algorithm.

The performance of the model has demonstrated that the diff features of each refactoring type are captured by the model and that these features are highly identifiable. This also proves that our idea of using diff to distinguish between refactoring types is feasible, and confirms that the checker’s encoding approach of the diff encoder effectively expresses the refactoring diff features.

5.3 Evaluation

The core problem addressed in this section is the use of the result checker to fully optimise false positive and false negative results caused by the threshold problem of RefDiff 2.0. The content of this section is a comprehensive evaluation of the result checker, including its performance, threats to validity, and limitations,. It is possible to again verify the utility of refactoring diff in practical applications, as well as the effectiveness of our diff encoder and diff feature models.

5.3.1 Approach Evaluation

The core problem addressed in this section is the use of the result checker to fully optimise false positive and false negative results caused by the threshold problem of RefDiff 2.0. The content of this section is a comprehensive evaluation of the result checker, including its performance, threats to validity, and limitations,. It is possible to again verify the utility of refactoring diff

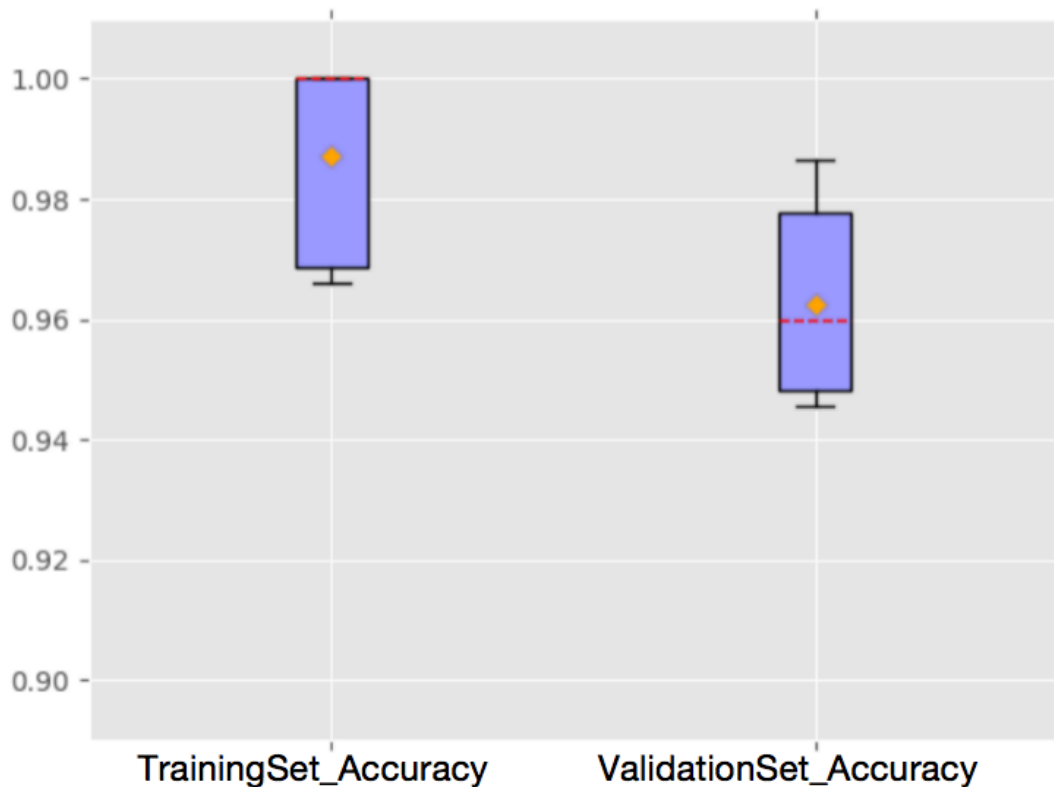


FIGURE 5.4: Accuracy distribution

in practical applications, as well as the effectiveness of our diff encoder and diff feature models.

Performance of Result Checker

For our evaluation we used four different configurations as test subjects: (1) RefDiff 2.0, original threshold, no result checker, (2) RefDiff 2.0, threshold 0.2, no result checker, (3) RefDiff 2.0, original threshold, with result checker, and (4) RefDiff 2.0, threshold 0.2, with result checker. We did not reduce the similarity threshold further than 0.2, because while collecting the data pool for the ground truth discussed in the previous section, we found that the majority of true refactoring candidates had a similarity greater than 0.2. Reducing the threshold further would put additional stress on the result checker, eventually leading to a decreased precision. As we were already satisfied with the recall achieved in the evaluation presented here, we did not attempt to reduce the threshold further.

As test data, we used the 1951 refactoring instances from our data pool that were not used in the training (cf. Section 5.2.2), which also comprise ten different refactoring types. The test results are shown in Table 5.4. We also used these test data to test RefactoringMiner 2.0 with results of 99.6% for precision and 96.9% for recall as our reference.

TABLE 5.4: Test Result

Refactoring Type	#	RefDiff 2.0		RefDiff 2.0 (0.2)		RefDiff 2.0 with Checker		RefDiff 2.0 with Checker (0.2)	
		Precision	Recall	Precision	Recall	Precision	Recall	Precision	Recall
Extract Interface	24	0.875	0.875	0.875	0.875	1.000	0.875	1.000	0.875
Extract Method	495	0.965	0.707	0.906	0.960	1.000	0.703	0.996	0.951
Extract SuperClass	50	1.000	0.743	1.000	0.743	1.000	0.743	1.000	0.743
Inline Method	112	0.954	0.741	0.850	0.964	1.000	0.741	1.000	0.946
Move Class	719	1.000	0.974	0.999	1.000	1.000	0.974	1.000	0.996
Move Method	205	0.874	0.810	0.715	0.941	0.988	0.785	0.989	0.902
Pull Up Method	31	0.963	0.839	0.833	0.968	1.000	0.839	1.000	0.968
Push Down Method	38	0.973	0.947	0.860	0.974	1.000	0.947	1.000	0.974
Rename Class	87	0.951	0.897	0.933	0.966	0.974	0.862	0.976	0.920
Rename Method	190	0.964	0.700	0.360	0.953	0.992	0.679	0.983	0.921
Total	1951	0.967	0.835	0.794	0.966	0.997	0.828	0.995	0.952

Result Analysis

From the tests it can be seen that the result checker improves the precision of RefDiff 2.0 by 3.0 percentage points, reaching almost 100%, while recall dropped by 0.7 percentage points when the similarity threshold is not changed. The main reason for this is that when RefDiff outputs a false positive refactoring result, the final result will only output that false positive if the checker outputs the same false positive, but the probability of this happening is extremely low, so the number of cases where the final output is a false-positive result is significantly reduced. The recall in this benchmark drops slightly, because the checker incorrectly predicts the few true-positive results of refactoring types *ExtractMethod*, *MoveMethod*, *RenameClass* and *RenameMethod*. Thus this prediction does not match the RefDiff results, causing them not to be output in the final results.

When the similarity threshold for RefDiff 2.0 was reduced to 0.2, the precision decreased significantly and the recall increased significantly, indicating that the lowered threshold matched more true refactorings, but also led to more false-positive results. The only two recalls that did not change after lowering the threshold were *ExtractInterface* and *ExtractSupClass*, mainly because RefDiff defines the decision conditions for them differently, not using the similarity threshold, so a change in the threshold would not affect their detection results. The table shows that many true refactorings in the four refactoring types *ExtractMethod*, *InlineMethod*, *MoveMethod* and *RenameMethod*, were missed due to the similarity threshold, and lowering the threshold allowed some refactorings with a code similarity of 0.2–0.5 to be matched successfully, which is the main reason for the increased recall. However, lowering the recall means that a large number of false positives are matched, especially for the *RenameMethod*, as the similarity threshold is the only condition for the *RenameMethod* to be matched, and a lowered threshold leads to a sharp increase in the number of false positives matched. Since the other refactoring types are subject to other conditions on matching, such as the signature and child conditions for *MoveClass*, and the use relationship between nodes for *ExtractMethod*, the lowering of the threshold allows for a small increase in the number of false positives of other refactoring types. In the case of a large number of false positives, the effect of the checker on the precision is highlighted, as the checker intervenes to filter out a large number of false positives, and although a few true positives are incorrectly predicted in the

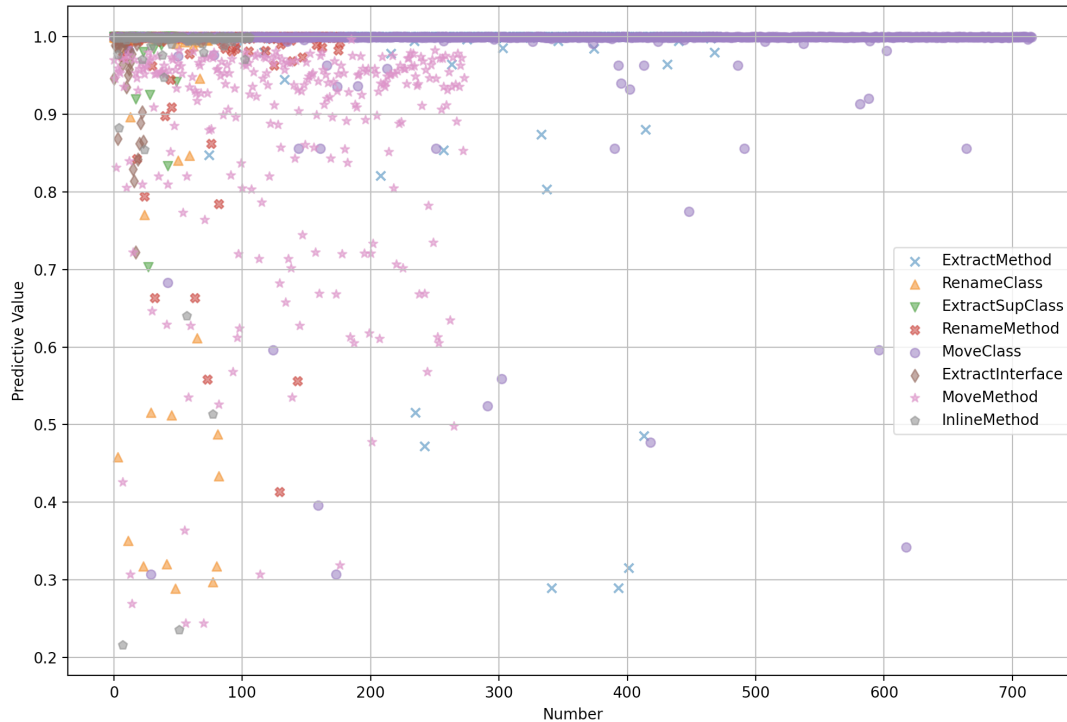


FIGURE 5.5: Distribution of predicted values for test data

process, the checker improves the precision of RefDiff 2.0 (threshold = 0.2) by 20.1 percentage points, while the recall was only reduced by 1.4 percentage points. Overall, the result checker (threshold = 0.2) improved recall by 11.7 percentage points and precision by 2.8 percentage points compared to the original RefDiff 2.0. This is very close to the detection performance of RefactoringMiner 2.0, but still retains the applicability of to multiple programming languages.

The benchmarks have demonstrated that the result checker improves the detection performance of RefDiff, and the diff feature model is applicable to data outside of the training and validation sets, indicating that the model fits well and has excellent generalisation capabilities. In addition to this, the checker also provides a score for the predicted refactoring type for each refactoring diff, which can be interpreted as the probability that the prediction is correct. Fig. 5.5 shows the distribution of the predicted probability of the refactoring diff reported by the checker (threshold = 0.2) for the different refactoring types. The figure shows that in most cases the predicted probability is very close to 100%. It can be assumed that the lower the predicted probability is, the more likely the checker is to detect the refactoring type incorrectly. To confirm this assumption, we collected 32 predictions of our result checker with a probability below 0.5, and we manually confirmed that 18 of these predictions were incorrect. This also confirms the results of the controlled experiments for model evaluation (Section 5.2.2). Although the false predictions of the checker cannot directly affect the final result, the user can decide whether to intervene manually to consult the diff code directly based on the probability.

Running time. We trained the model using a computer with a 2.3 GHz

Intel Core i5 processor and 8 GB of 2133 MHz LPDDR3 memory. It took a total of 136 minutes to train the diff feature model. The average prediction time for each diff was 26 ms.

5.3.2 Threats to Validity

There are at least three threats to validity for the result checker in this chapter.

Firstly, the result checker's diff extractor may extract non-diff code as a diff. In the diff extractor, we use the `ExtractIntersection` function to extract the parts of the node code that are common to `FullDiff`, but if there are non-diff codes in the node code that are the same as those in `FullDiff`, then these non-diff codes will be extracted as diffs. We tested the checker's prediction results when this happens, and the results show that the non-diff code has very limited impact, because the diff code containing the refactoring features will be extracted anyway, and these diff codes are the main basis for predicting the refactoring type.

Secondly, the result checker's diff encoder may confuse `RenameMethod` and `MoveMethod` when encoding them. Our diff encoder is designed based on the assumption that the elements associated with the refactoring features in the diff are invariant using the length of each word to encode them. But when the method is renamed to another name of the same length, the encoded `RenameMethod` corresponds to the same two-dimensional array as some `MoveMethod` arrays. This has not happened in either training or testing in this chapter, and even if it had, the checker error result would not be output because of inconsistency with the `RefDiff` detection result.

Thirdly, the performance of the checker when confronted with a few refactoring types whose refactoring diffs do not conform to the characteristic that the contents of the removed and added parts remain unchanged. Examples include **Introduce Assertion**, **Remove Setting Method**: here the diffs of multiple refactoring candidates can vary a lot but they contain very specific keywords such as (*assert* and *set*), which can be used as basis for the diff feature model to identify them.

5.3.3 Challenges and limitations

The result checker has two limitations in training and testing. Firstly, the result checker does not support checks for nested refactorings at this time, i.e., multiple refactorings applied to the same program element, such *Extract and Move Method* or *Move and Rename Class*. The challenge here is that some nested refactoring diffs may not have representative features, i.e. the manifestation of the diff may be confused with some single refactoring types. Such effects need to be studied in more depth.

Secondly, manually applied refactorings and refactorings applied by IDE tools sometimes differ. For example, in *ExtractMethod*, manual refactoring inserts the method code directly on top of the extracted code, which will not show the removed and added parts of the extracted code in the diff and will easily cause false predictions. However, from the training and test data,

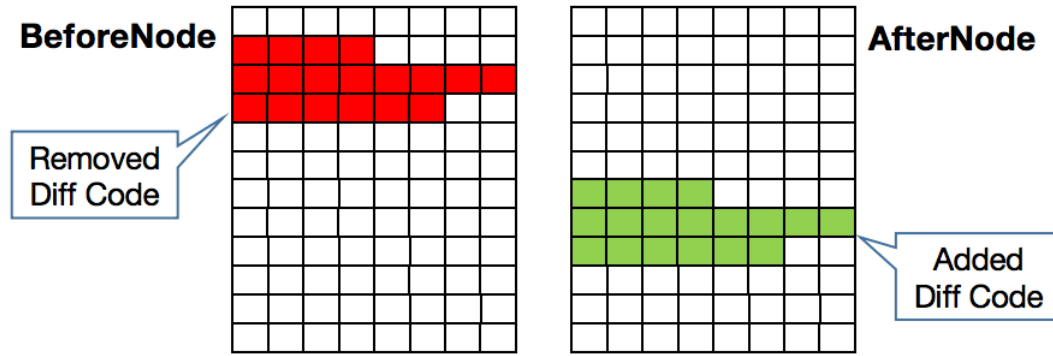


FIGURE 5.6: Abstract Illustration

this is rarely the case, and with the widespread use of refactoring tools, the refactoring diff will become more standardised.

5.4 Diff Feature Matching Network

5.4.1 Approach Overview

Diff Feature Matching Network (Tan Liang, 2022b) is designed based on the phenomenon that the removed and added parts of a refactoring diff are almost identical.

Similarity theory based on word frequency cannot cover all the cases that may be encountered in refactoring detection. In the example shown in Fig. 5.6, the two grids represent the entire code of BeforeNode and AfterNode. The red boxes represent the remove part of the diff and the green boxes represent the add part. Refactoring in this context refers specifically to the type of refactoring achieved by changing the structure of the code, which can be called structural refactoring. The refactorings move code around without or with only minimal changes or apply small changes to code in place. In such cases, the red and green sections would be very similar and word frequency analysis is good at detecting matches. But when faced with noisy nodes, with addition or removal of code that does not belong to the refactoring, the red and green parts would contain also the code of the non-related changes, impacting the word frequencies such that the two sections would not be recognized as matching.

To improve this, we replace this text-similarity-based matching algorithm with a diff feature matching network to determine whether the two nodes match, i.e., participate in the same refactoring, which is less vulnerable to noise mixed with refactoring code changes. The flowchart in Fig. 5.7 shows an overview of our whole approach.

Firstly, we extract the nodes in **Remove** and **Add** (**BeforeNode** and **AfterNode**) and match them in turn. Secondly, using our **Diff Extractor**, the

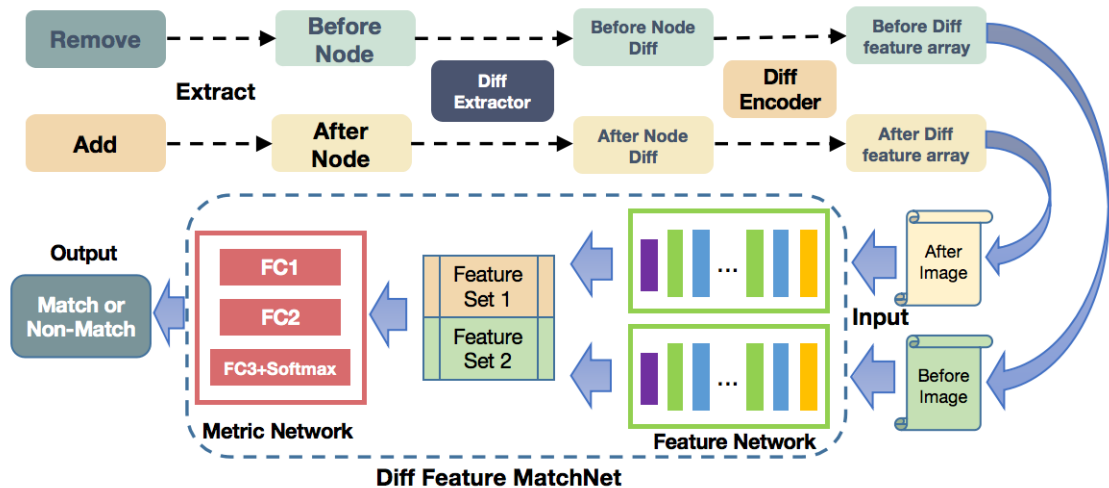


FIGURE 5.7: Approach Flow chart

diffs present in the nodes are extracted, and **Diff Encoder** extracts the required features from both the before and after nodes, including word frequencies, code structure, code order, syntactic constraints and other characteristics. Next, the two arrays are converted into images that are used as input for our **Feature Network**—inspired by the image matching algorithm MatchNet (Xufeng Han, 2015)—to extract their feature sets, seeing that the two feature networks form a twin-tower structure with shared parameters. Finally, their feature sets are paired and input into **Metric Network**, which computes the feature matching rate of the two feature sets and outputs the matching result. This design almost completely bypasses the problem of defects when judging node matching by word frequency similarity, and the focus on diff also avoids false negatives to a certain extent.

5.4.2 Training Process

This section documents the preparation of training data for the diff feature matching network, the process of building the matching network and network structure.

We used java refactoring data as the training data for the diff feature matching network, as refactoring is more widely used and developed in java compared to other programming languages, and open source datasets exist to be used. Our training data was primarily sourced from an open source dataset containing 538 commits³, as well as some data collected by our team, for a total of 2272 refactoring instances. The training data consists of eight refactoring types *Extract Method*, *Inline Method*, *Move Class*, *Move Method*, *Pull Up Method*, *Push Down Method*, *Rename Class* and *Rename Method*. These refactoring types were chosen because RefDiff detects them using similarity as a matching condition. We keep the number of each refactoring type to the same order of magnitude when collecting data to prevent model distortion

³<https://aserg-ufmg.github.io/why-we-refactor/#/allCommits>

TABLE 5.5: Data Distribution

Refactoring Type	Data Pool	Training Data	Validation Data
Extract Method	379	303	76
Inline Method	374	299	75
Move Class	391	313	78
Move Method	383	306	77
Rename Class	375	300	75
Rename Method	370	296	74
Total	2272	1817	455

caused by data imbalance and to ensure that the trained model is suitable for matching multiple refactoring types.

Data preparation

For each refactoring instance in the training data, we used RefDiff 2.0 to run it, and then after a second manual check, we obtained the BeforeNode and AfterNode corresponding to each true refactoring match, and then extracted their node codes. Among these true refactorings we identified samples that RefDiff 2.0 missed to detect because the similarity did not exceed the threshold. For these samples, we extracted the node codes manually. Then, after processing by diff extractor and diff encoder, two greyscale maps of size $64*64*1$ representing the refactoring diffs were obtained, which are placed in two separate folders but with the same filename. Next, we create the matching dataset by pairing the images with the same filename as one training data item with the label **Match**. For creating the mismatched dataset, we pair images in the two folders with different filename at random as training data item with the label **Non-Match**. The Non-Match dataset contains the same amount of data as the **Match** dataset. As shown in Table 5.5, the data in the Match dataset contains a total of 2272 data, with 1817 items for training and 455 for validation in a ratio of 8:2 between the training and validation sets. Non-Match has the same amount of training and validation data as Match.

Training Process

The training of diff feature matching network follows the MatchNet network framework, which consists of a sampling layer, a feature extraction network, a metric network and a cross-entropy loss function. The structure of the diff feature matching network is shown in Fig. 5.8, and the parameters of the diff matching network are listed in Table 5.6. In the data preparation phase, we have completed the sampling layer.

First, the feature extraction network was constructed, which is based on the classical convolutional neural network AlexNet (Krizhevsky Alex, 2015), but with some changes to it by adding a preprocess layer and bottleneck layer. The former to normalise the data and the latter is used to control the

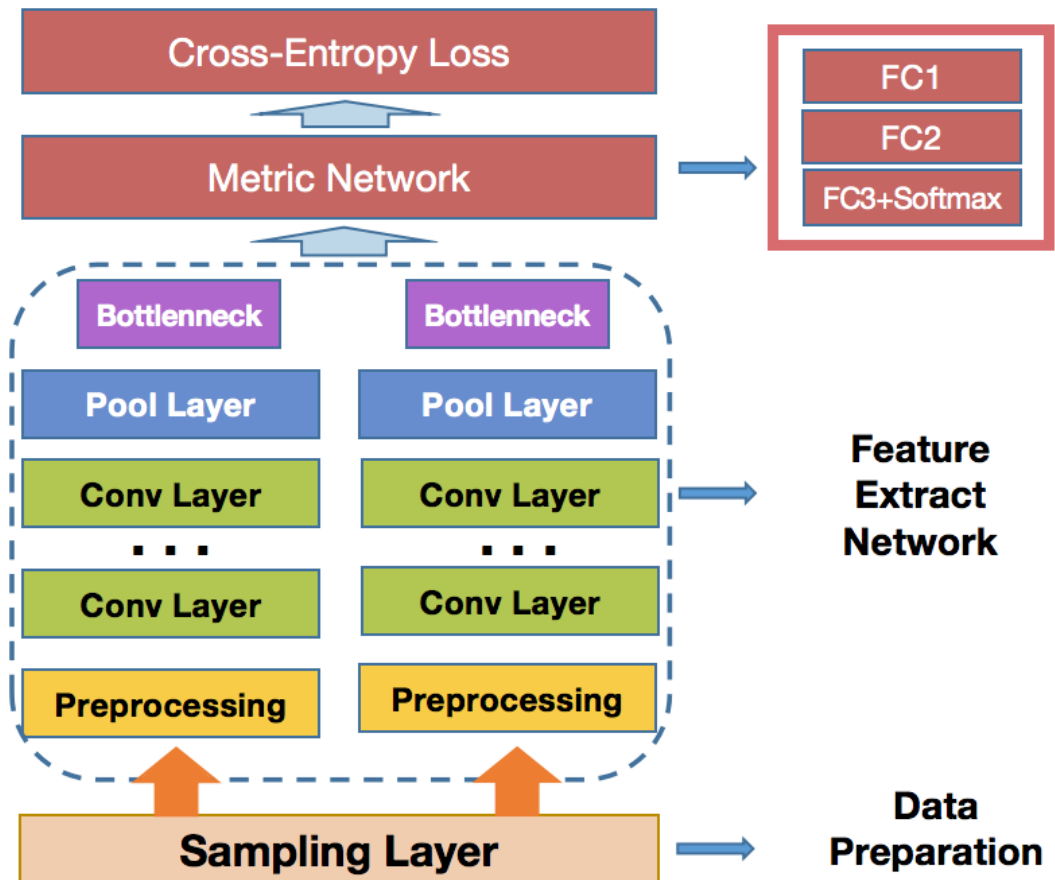


FIGURE 5.8: Structure of Diff feature Matching Network

TABLE 5.6: Parameters of Diff Feature Matching Network

Layer Name	Layer Type	Output Dimension	Size	Stride
Conv0	C	64*64*24	3*3	1
Pool0	Max	32*32*24	3*3	2
Conv1	C	32*32*64	3*3	1
Pool1	Max	16*16*64	3*3	2
Conv2	C	16*16*96	3*3	1
Conv3	C	16*16*96	3*3	1
Conv4	C	16*16*64	3*3	1
Pool2	Max	8*8*64	3*3	2
Bottleneck	FC	B	-	-
FC1	FC	F	-	-
FC2	FC	F	-	-
FC3	FC	2	-	-

dimensionality of the feature vector input from the feature extraction network to the full connect layer to avoid overfitting. In the training phase, the feature network acts as a two-stage pipeline, sharing parameters, and the outputs of the two towers are concatenated together as the input to the metric network. It is worth mentioning that, unlike images, in order to accommodate matching the full range of single to multi-line codes, we chose to use a convolutional kernel of size 3*3, which allows for better extraction of array features. Secondly, in the Metric network, three full connect layers with ReLU nonlinearity and a Softmax function are set up for feature comparison, and the output is obtained as two values of the probability [0,1] that the two images are similar, both positive and summing to 1, which can be interpreted as the probability of a match and the probability of a non-match. Finally, a cross-entropy loss function with a learning rate of 0.0001 is set to evaluate the performance of the diff feature matching network. In the training process, for each batch size, 50% of the data with **Match** labels and 50% of the data with **Non-Match** labels were used.

5.4.3 Model Evaluation

The training results are shown in Fig. 5.9. A total of 20 epochs of training were performed, and the final training and validation sets of the model accuracy were 0.990 and 0.917, and their loss values were 0.056 and 0.320. The accuracy and loss curves of the training set in the figure indicate that the model converges well, and the loss value of the validation set stays flat after the 11th epoch with a small oscillation and does not keep getting larger as the training set converges, indicating that the model is not overfitted.

To better evaluate the performance of the diff feature matching model, we designed a controlled experiment on a true *ExtractMethod* refactoring instance, as shown in Fig. 5.10. This example was judged by RefDiff to be non-refactored. Since the similarity of the two nodes in this example was only 0.339, which is less than the threshold of 0.5, the two nodes were not matched as refactoring candidates, thus resulting in a false negative detection result.

The red part is the removed part of diff and the green part is the added part of diff. The code content in the boxes is the refactored part, and the code outside the boxes is the non-refactored code change. In this example, we compare the matching performance of the diff feature matching network and word frequency similarity of RefDiff. Firstly, the refactoring part is kept constant during the comparison. We tested the performance of the two matching approaches by adding some noisy code to the added part and reducing some non-refactored code from the added part. Adding noisy code means adding 1-20 lines of code that are unrelated to the refactoring to the original code. Reducing the non-refactored code means reducing some non-refactored code change lines from the original added part code.

As shown in Table 5.7, there are ten matching cases, and the matching values are the performance of the two approaches in each case. The results

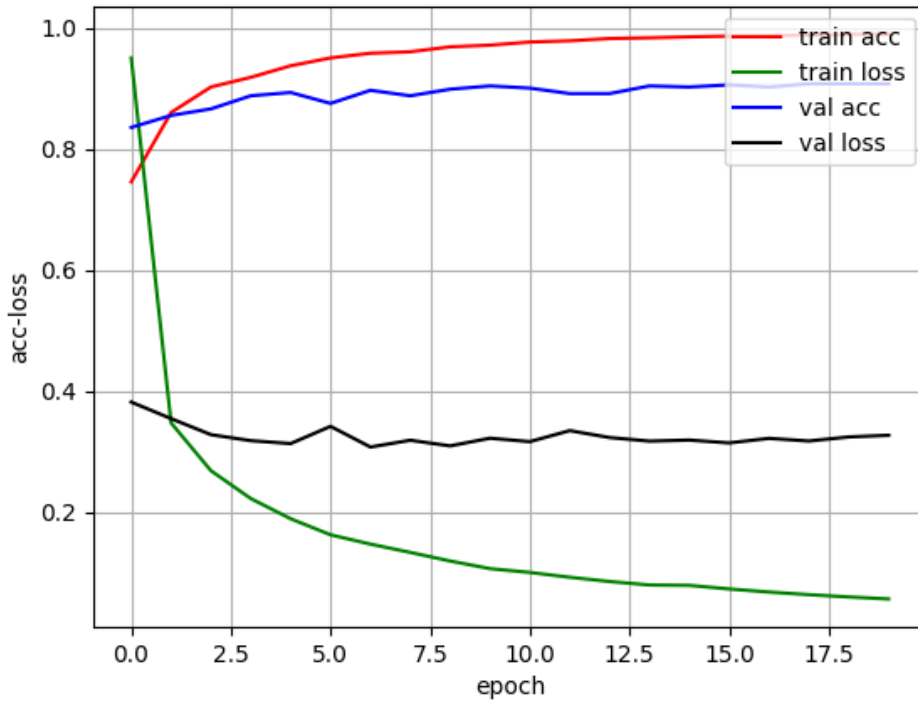


FIGURE 5.9: Train Result

```
-if (method.isVarArgs())
-throw new IllegalArgumentException(String.format("Invalid varargs method ...;
-if (ps.getVariables().size() != method.getParameterTypes().length)
-throw new IllegalArgumentException(String.format("The number of arguments for ...
-method.getName(), method.getParameterTypes().length, ps.getVariables().size()));
```

Refactoring Part

```
+private void validateParameters() {
+if (method.isVarArgs())
+throw new IllegalArgumentException(String.format("Invalid varargs method ...
+ColumnDefinitions variables = statement.getVariables();
+Set<String> names = Sets.newHashSet();
+for (ColumnDefinitions.Definition variable : variables) {
+names.add(variable.getName());
+}
+if (method.getParameterTypes().length < names.size())
+throw new IllegalArgumentException(String.format("Not enough arguments for ...
++ "found %d but it should be at least the number of unique bind parameter names ...
+method.getName(), method.getParameterTypes().length, names.size());
+if (method.getParameterTypes().length > variables.size())
+throw new IllegalArgumentException(String.format("Too many arguments ...
++ "found %d but it should be at most the number of bind parameters...
+method.getName(), method.getParameterTypes().length, variables.size());
```

Refactoring Part

FIGURE 5.10: Testing Example of Comparison

TABLE 5.7: Testing Result

Test Situation	Word-frequency Similarity	Diff Matching Network
Original -10	0.631804	0.999999
Original -5	0.460249	0.999997
Original -3	0.431828	0.999789
Original -1	0.370762	0.999824
Original	0.338710	0.999668
Noise +1	0.312438	0.999869
Noise +3	0.274755	0.998295
Noise +5	0.218686	0.973053
Noise +10	0.188181	0.826769
Noise +20	0.148184	0.998854

in the table show that the matching performance of the diff feature matching network is excellent and stable. Each value represents the matching probability of removed part and added part, and neither the reduction of non-refactored code changes nor the addition of code noise affects its matching performance, indicating that the diff feature matching network has good robustness. For word frequency similarity, it can be found that the non-refactored code change has a huge impact on the similarity, and the similarity can only exceed the threshold of 0.5 when reducing 5 to 10 lines of non-refactored code, while the similarity in the face of noisy code will keep decreasing with the increase of noise, indicating that the word frequency similarity is only suitable for matching with pure refactoring behaviour or accompanied only by a small amount of non-refactored code changes, which also validates our analysis of the RefDiff similarity problem.

In terms of runtime, word frequency similarity is more advantageous, due to the low computational load of 0.38 ms on average, compared to 220ms-240ms per match for the diff feature matching network in the python environment.

5.4.4 Approach Evaluation

The core research problem of this section is to solve the problem of false positives and false negatives due to word frequency similarity by using diff feature matching networks.

Deployment

In the Java environment, we tried to deploy a matching network in RefDiff instead of the word frequency similarity matching algorithm, but pre-processing the matching data required multiple data transformations, and loading the model and computing the results by using the *deeplearning4j* library took a significant amount of time in total, averaging around 5s-7s per match, which made that approach infeasible for matching a large number of nodes.

In order to better allocate computational power and maintain the original design architecture of RefDiff as much as possible, we decided to lower the RefDiff word frequency similarity threshold to 0.1, the aim is to obtain as many refactoring candidates as possible and then triage these candidates. This is shown in Fig. 5.11.

Step 1: We classify the candidates into those with a similarity of $\lambda \in (0.1, 0.5]$ and a similarity of $\lambda \in (0.5, 1.0]$.

Step 2: Those candidates with $\lambda \in (0.5, 1.0]$ are the ones that the unmodified RefDiff would recognize. As we know that these results are already reliable, we do not have to apply our approach for candidate detection. The remaining candidates would have been eliminated by RefDiff, but we know that a lot of false negatives are among them, therefore we apply our approach to identify additional candidates among them. The main purpose of the triage process is to allow the diff feature matching network to focus its computational power on finding as many true refactorings as possible among the candidates with similarity below the threshold.

Since our approach currently is rather slow, we designed the additional Diff Token Filter to filter to reduce the number of candidates to which our network is applied. This filter computes the word frequency similarity, using the same TF-IDF algorithm as the original RefDiff matching algorithm, but only on the diffs. This means, it only focuses on the diff part of the node code, from which we remove all symbols and only keep the more representative words as tokens. Applying this filter to the candidates with a similarity of $\lambda \in (0.1, 0.5]$ allows us to detect additional refactorings that RefDiff would have missed. Take *ExtractMethod* from the model evaluation in the previous section as an example (Fig. 5.10). The node similarity is 0.339, while the similarity calculated for the extracted diff word tokens is 0.632. We also apply the Diff Token Filter to the candidates with a similarity of $\lambda \in (0.5, 1.0]$. By doing so, we can filter out some false positive candidates before letting the regular RefDiff algorithm detect contained refactorings.

This suggests that for true refactorings the Diff Token Filter is more instructive than the original similarity. Therefore, for $\lambda \in (0.1, 0.5]$, the diff threshold is set to 0.2, which serves as a filter and also tries not to miss true refactoring candidates.

Step 3: Considering the time consumed in calling the matching network in the java environment, we decided to complete the matching of candidate data by the matching network in the python environment. The candidate results that pass through the matching network will be merged with the results reported from RefDiff into the final detection result.

Performance of Diff Feature Matching Network

In our evaluation, firstly, in the context of java, we used three different configurations as test subjects. (1) RefactoringMiner 2.0, (2) RefDiff 2.0, (3) RefDiff 2.0, plug-ins the diff feature matching network with Diff Token Filter (abbreviation: RefDiff 2.0 with Diff-MatchNet). This test is concerned with the performance of the diff feature matching network in solving the threshold

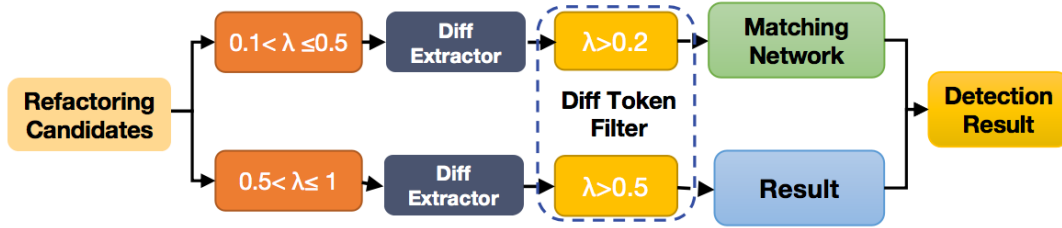


FIGURE 5.11: Filter Layer

problem, so the target refactoring focuses only on the eight types of structured refactoring that use word frequency similarity as a matching condition. The test data were obtained from RefDiff 2.0⁴ and they contained a total of 3154 refactorings. The results are shown in Table 5.8, where we focus on the improvement of the diff feature matching network over RefDiff for the same benchmark.

In addition, we also tested the performance of the diff feature matching network against JavaScript and C, test data also from RefDiff 2.0, and the results are shown in Table 5.9 and Table 5.10.

TABLE 5.8: Test Result

Refactoring Type	#	RefDiff 2.0		RefactoringMiner 2.0		RefDiff 2.0 with Diff-MatchNet	
		Precision	Recall	Precision	Recall	Precision	Recall
Extract/Extract Method and Move Method	1037	0.962	0.663	1.000	0.911	0.982	0.895
Inline Method	122	0.957	0.721	0.991	0.926	0.973	0.885
Move Class	1100	0.999	0.970	1.000	0.991	1.000	0.987
Move Method	319	0.871	0.803	0.993	0.909	0.961	0.928
Pull Up Method	91	0.974	0.824	1.000	0.967	1.000	0.945
Push Down Method	40	0.950	0.950	1.000	0.950	1.000	0.950
Move and Rename/Rename Class	95	0.922	0.874	1.000	0.884	0.967	0.937
Rename Method	350	0.946	0.694	0.977	0.860	0.975	0.880
Total	3154	0.964	0.805	0.996	0.935	0.986	0.932

TABLE 5.9: Comparison in JavaScript

Refactoring Type	#	Precision	#	Recall
Extract Function	10	1.000	10	0.900
Inline Function	10	0.900	5	0.800
Move and Rename File	10	1.000	3	1.000
Move and Rename Function	10	1.000	7	1.000
Move Class	2	1.000	0	N/A
Move File	10	1.000	10	1.000
Move Function	10	0.900	10	1.000
Rename File	10	1.000	10	1.000
Rename Class	5	1.000	0	N/A
Rename Function	10	0.900	10	1.000
Total	87	0.967	65	0.969

Result Analysis

For the Java test, it can be seen from the test that the diff feature matching network improved the recall of RefDiff by 12.7 percentage points, which

⁴<https://github.com/aserg-ufmg/RefDiff>

TABLE 5.10: Comparison in C

Refactoring Type	#	Precision	#	Recall
Change Signature	10	1.000	10	1.000
Extract Function	10	1.000	10	0.800
Inline Function	10	0.900	10	1.000
Move and Rename File	0	N/A	10	1.000
Move and Rename Function	10	0.900	10	1.000
Move File	10	1.000	10	1.000
Move Function	10	0.900	10	0.900
Rename File	10	1.000	10	1.000
Rename Function	10	0.900	10	1.000
Total	80	0.952	90	0.967

has caught up with the performance of RefactoringMiner. In terms of precision, the matching network with Diff Token Filter helped RefDiff improve by 2.2 percentage points, demonstrating that focusing on diff is able to filter out some false positives. In detail, the match network's improvement in recall for ExtractMethod, InlineMethod, MoveMethod and RenameMethod is huge, even approaching or surpassing the performance of RefactoringMiner in some types.

For the JavaScript and C tests, RefDiff's results were 91% precision and 88% recall for JavaScript and 88% precision and 91% recall for C. For the same test data, the diff feature matching network boosted both precision and recall for JavaScript and C by 5 to 7 percentage points. We reached a precision of 96.7% and recall of 96.9% in JavaScript test, a precision of 95.2% and recall of 96.7% in for C. The test results show that the positive impact of the matching network is significant, reflecting the fact that the diff matching network is equally valid for other programming languages as the text matching network.

Theoretically, all refactorings are matched except for false negatives caused by setting thresholds and mis-matches by the diff matching network, but our test data shows that there are a number of controversial 'refactorings' where the content of the refactorings has been changed, for reasons such as changing judgement conditions, changing parameters, changing the way of function calling, etc. For example, in the case of *RenameMethod*, the method name was renamed and the contents of the method were changed, completely losing the correspondence between the remove and add parts of the refactoring diff. This has led to some of the "false negative" results.

Running time. We trained the model using a computer with a 2.3 GHz Intel Core i5 processor and 8 GB of 2133 MHz LPDDR3 memory. It took a total of 33 minutes to train the diff feature matching network. The average total time consumed by candidates to pass through the filters and feature nets is 240ms to 260ms.

5.4.5 Threats to Validity

There are two validity threats to the diff feature matching network in this chapter.

Threat 1, the diff extractor may extract non-diff codes as diffs. In the diff extractor, we use the *ExtractIntersection* function to extract the parts of the node code that are common to FullDiff, but if there are non-diff codes in the node code that are identical to some lines in FullDiff, then these non-diff codes will be extracted and treated as diffs, although this is a small probability. This situation does not affect the work of the matching network, as the remove and add parts of the refactoring diff are unaffected and will be matched as long as they contain enough of the same features, with the non-diff code having as little impact as the non-refactored code changes.

Threat 2, there is a threat to the data used for evaluation. There is subjectivity in the true/false positives of java refactoring data that directly affects detection performance. Different verifiers may have different interpretations of these refactoring types. We therefore chose RefactoringMiner 2.0 as a control tool, where comparisons under the same benchmark can reduce the impact on tool performance due to small amounts of data. The data size for testing JavaScript and C is so small that the test results are not informative. Testing other languages than java, our main aim was to demonstrate that the diff matching network is applicable to other programming languages and that matching networks in other languages can also solve the problems posed by thresholds.

5.4.6 Challenges and Limitations

Limitation 1: Some refactoring types cannot be detected as they either only remove or only add code. Examples include *Introduce Assertion*, *Remove Setting Method*. In these cases the approach does not work, because naturally there is no correspondence between the remove and add parts of the diff, as one of the parts is empty. However, these particular refactoring types are only rarely used, and it would be possible to detect them in the future using keyword features.

Limitation 2: When matching a diff with only a few short lines of code, the mismatch rate of the matching network increases. Because feature extraction networks have a minimum convolution kernel of 3×3 , the features contained in very short lines of code, where with the features are convoluted and pooled many times, may be diluted, resulting in a failure to match at the full connect layer. We are studying the use of smaller convolution kernels to deal with this problem.

Chapter 6

RefDiff-Model

In the course of my research, we have identified a number of potential pitfalls that could affect detection performance when processing data using the diff extractor and diff encoder, and have attempted to fuse the two models from the previous chapter, as well as attempting to merge all of my work findings, in the hope of arriving at a better integrated solution that can both guarantee detection performance and break through existing limitations.

6.1 Problem Analysis

Both approaches in previous chapter that rely too heavily on the performance of a single model when deployed and used, which makes it so that incorrect model predictions will directly affect the final detection results. We have determined that the erroneous results are caused by flaws in the extraction and encoding approaches, and in some cases may be the result of extraction to the wrong diff or confusion in the encoding.

6.1.1 Problems with existing models

Problem with Diff Structure Feature Model

Diff Structure Feature Model is designed based on the fact that each type of refactoring diff has a special structure. The special structure of refactoring types is also an important dependency for the manual identification and classification of refactorings. The encoding of diff and the special encoding of programming language keywords are trained to achieve model recognition of different refactoring types. The model is trained on the diff of the refactoring candidate (diffs of BeforeNode + diffs of AfterNode), and the corresponding refactoring type is the training label. The model is equivalent to adding a filtering layer to RefDiff for filtering out detection results that differ from the structure of the corresponding refactoring type, directly increasing the precision of the tool. According to this mechanism, developers use a reduced threshold to obtain more detection results, and then use the structure feature model to filter out false positives among them, achieving the goal of improving RefDiff's detection performance. In addition, depending on the assignment of different programming language keywords, structure feature models can be trained to predict different programming languages, thus maintaining the core advantage of RefDiff's wide generality.

However, crudely lowering the threshold produces a large number of false positive results, and using the model to calibrate them one by one consumes a significant amount of time. Also, over-reliance on a single model will inevitably lead to incorrect predictions, which is why this result checker also leads to a small decrease in recall when boosting the precision of RefDiff. Thus, as with *Diff Feature Matching Network*, the deployment of structure feature models, and the move away from single dependencies, needs further optimisation.

Problem with Diff Feature Matching Network

Diff Feature Matching Network was designed based on the fact that the removed and added parts of each refactoring diff are almost identical. This design applies to most types of refactoring, i.e. refactorings that are implemented as moving and replacing code, where the code that is moved in the diff is the removed and added part. The refactoring does not change the external representation of the code, only the internal structure, so the refactored part of the code will show a very obvious correspondence between the removed and added parts of the diff, which makes the matching network almost ignore the threat of non-refactored code changes, almost solving the problem of low recall due to threshold. The diff feature matching network was created with reference to the image matching algorithm MatchNet (Xufeng Han, 2015). The sampling layer processes diffs of BeforeNode (removed part) and diffs of AfterNode (added part), the feature network extracts the features contained in the removed and added parts, and then the metric network compares these features to produce a match.

Moreover, the matching network is trained on pure diff code text, which does not rely on any syntax, and each match is an independent feature extraction, and the expression of these features does not differ due to different syntax, so a matching network trained on one programming language can be widely generalised to other programming languages. However, the main function of this matching network is to determine whether there is a corresponding feature in the diff, but this condition is not a sufficient condition for determining the refactoring, and it is still possible to match a false positive result, so other conditions are needed to collaborate to further determine the refactoring.

6.1.2 Problems with Diff Tool

Problem with Diff Extractor

Threat to Diff Extractor: The original extractor was implemented using the intersection of NodeCode and FullDiff (the set of all diffs for both versions of the code), but this algorithm of extracting the intersection is likely to extract non-diff codes as diffs. As the extraction algorithm compares NodeCode and FullDiff line by line, if there are non-diff codes in NodeCode that are the same as some lines of FullDiff, then these non-diff codes will also be extracted as a diff, affecting the training accuracy.

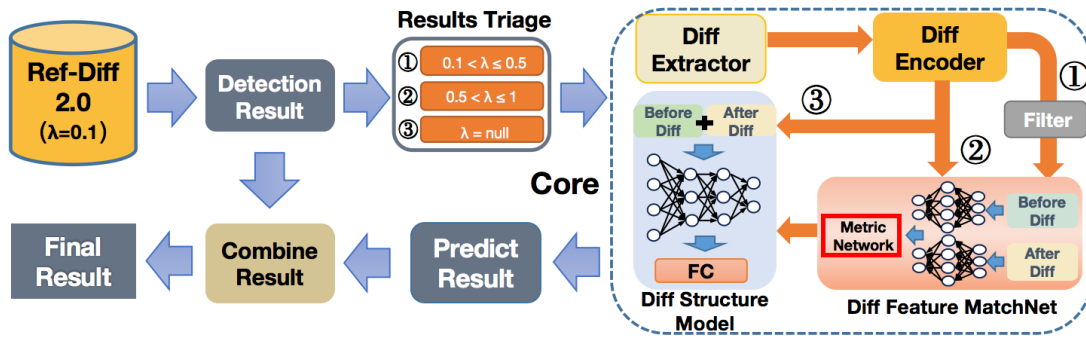


FIGURE 6.1: Approach Flow chart

Problem with Diff Encoder

Threat to Diff Encoder: The original encoder uses the length of each word as its abstract expression, converting the diff to an array based on a "Jigsaw Hypothesis", but the character lengths can cause confusion in some cases. The main impact of this obfuscation manifests itself in the prediction of non-refactored code changes as matches, and in some occasional cases, unrelated code can be matched because the characters are the same length.

6.2 Approach Overview

To overcome these shortcomings, we propose **RefDiff-Model** (Tan Liang, 2023) that combines information from RefDiff's code representation and the two deep learning approaches, forming a mechanism for mutual corroboration of the prediction results. In this way, we combine the advantages of the two approaches mentioned above, design an approach that can escape the limitations of generality imposed by the different syntax of programming languages, and use the mined information from the diff code to give RefDiff a higher level precision and recall.

6.2.1 Approach Workflow

This section describes the process of implementing the RefDiff-Model, how to make more efficient use of the information contained in the CST nodes, how to mine the NodeCode for refactoring-related information and how to merge them to obtain better detection results. The flow chart of the optimisation is shown in the Figure 6.1.

First, we input the code to be detected into RefDiff 2.0. In order to obtain as many true positives as possible, we set the similarity threshold to 0.1, based on the available data, setting a lower threshold would hardly obtain a true refactoring, but would instead detect more false positives and increase the detection burden.

In the next step, we triage the detection results by threshold range as follows: ① $threshold \in [0.1, 0.5]$, ② $threshold \in (0.5, 1.0]$, ③ $threshold = null$. The main purpose of this triage is to perform different processing in the core

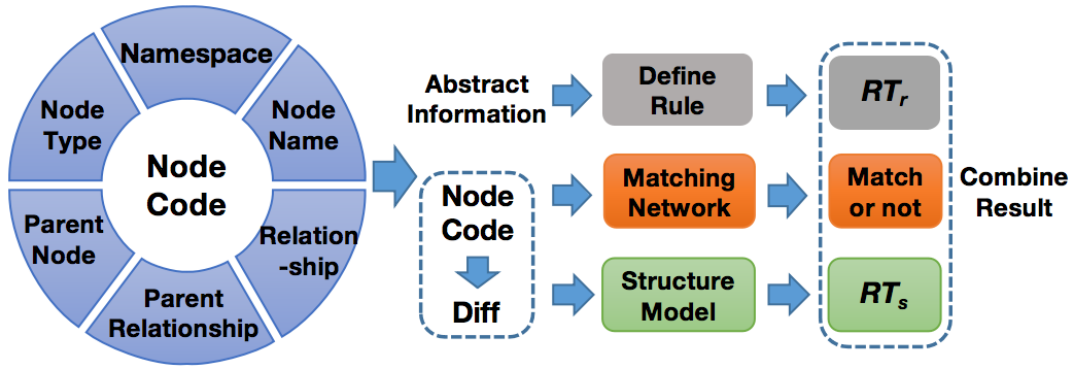


FIGURE 6.2: Node information processing concept

module according to the threshold range. The main reason for the triage process is that the time cost of multiple calls to the Diff feature matching network is higher than computing word frequency similarity. Therefore, it is more feasible to use word frequency similarity for primary screening of candidates.

This Core step is the most important part of this optimisation solution, where the NodeCode of the detection results is pre-processed and then passed through the models to obtain more information to help with the detection refactoring. The detection results of all streams need to be extracted from their NodeCode using Diff Extractor, then encoded by Diff Encoder and finally input into the models according to the data processing requirements of each stream. For ①, our main aim is to find all the true refactorings with a similarity within the first range and that are missed by RefDiff. Since this range will match a large number of false positive candidates, so we set up a diff-only filter (described in detail in Section 5.3) for filtering out these candidates before inputting them into the model for prediction. For ②, the precision of 96.4% for RefDiff has proven that the vast majority of candidates in this range are correct and so are directly input into the model for prediction. For ③, the threshold is not used as a matching condition for judging refactoring, so the encoded diff is input directly into the diff structure feature model to predict its corresponding refactoring type.

Finally, we combine all the predictions obtained in the core module with the defined judgements made by RefDiff based on the node abstraction information, and then unify them as the final result. Our idea is shown in Figure 6.2, and our approach is more sufficient than RefDiff when using the information contained in NodeCode (Figure 1.3).

6.2.2 Core Mechanism

The combined results are a mechanism for cross-corroboration. The final result is processed under different conditions. For ① and ②, the merge information consists of node abstraction information (RT_a), judgement information of the matching network (Match (M) or Non-Match (nM)) and judgement information of the diff structure (RT_s). When $\{RT_a = RT_s \wedge M\}$, this means that the candidate's structure features and abstract information point

to the same refactoring type, and that there is a matching correspondence between the removed and added parts of the candidate's diff. So we consider this to be a correct refactoring detection with a very high confidence level. Whereas when $\{RT_a = RT_s \wedge nM\}$ or $\{RT_a \neq RT_s \wedge M\}$, this means that the matching correspondence of the candidate's diff and the candidate's diff structure features do not simultaneously support the refactoring type conclusions derived from the abstract information. There may be model prediction errors, so the candidate is referred to manual intervention for identification due to insufficient confidence level. When $\{RT_a \neq RT_s \wedge nM\}$, the result is considered to be a false positive with very low confidence level. For ③, the merge information consists of node abstraction information (RT_a) and judgement information (RT_s) for the diff structure. When $\{RT_a = RT_s\}$ is true and $\{RT_a \neq RT_s\}$ is false.

This design of the optimisation solution not only finds as many true refactorings that are missed matches as possible, but also avoids the output of false positive results by way of mutual corroboration between information, largely avoiding the reliance on a single model for refactoring detection and fully achieving the goal of an overall improvement in the performance of RefDiff detection.

6.3 Optimised Solutions

6.3.1 Optimised Diff Extractor

The original Diff Extractor took the FullDiff of a changed file that contains the node in question and reduced this to the code lines that are contained in the NodeCode by means of a line-by-line comparison. In practice, the state of each line of NodeCode (added, removed, unchanged) is unknown, so the developer removes the flags from FullDiff that indicate the state of the code (the "+" and "-" at the beginning of each line). Such processing, while not missing any line of diff code in the NodeCode being extracted, may result in the non-diff code in the NodeCode being extracted as a diff because the non-diff code in the NodeCode is the same as the diff code of other nodes in FullDiff, which may affect the accuracy of the model.

The aim of our optimisation is to reduce the occurrence of false extractions and optimise extraction accuracy as much as possible. The optimised diff extraction algorithm is shown in Algorithm 4. In Figure 4.8, the removed code only exists in the beforeNode, and the added code only exists in the afterNode. So, we add a "-" flag to each line of code contained in the beforeNode and add "+" flag to the code in the afterNode, then we use the **ExtractInteraction** algorithm to obtain a more accurate diff, and the presence of the "+" and "-" further prevents incorrect extraction. (ExtractInteraction is a function for outputting lines that intersect in two texts, based on regular expressions.) This also allows the removed and added parts of the diff to be distinguished explicitly, making it easier for the user to view the diff when manual intervention is required, as shown in the *ExtractMethod* example Figure 6.3.

Algorithm 4 Refactoring Candidate Diff Extractor**Input** BeforeNode, AfterNode, FullDiff(V_1, V_2)**Output** nc-diff // Diff of NodeCode**Begin Body**

1. nc-diff $\leftarrow \emptyset$;
2. beforeNodeCode \leftarrow addFlag(BeforeNode.NodeCode);
3. afterNodeCode \leftarrow addFlag(AfterNode.NodeCode);
4. nodeCode \leftarrow beforeNodeCode + afterNodeCode;
5. nc-diff.add (**ExtractIntersection** (nodeCode, FullDiff(V_1, V_2)));
6. **return** nc-diff;

End Body**Function** addFlag(code)codeWithFlag $\leftarrow \emptyset$;**if** code \in BeforeNode **then**

| codeWithFlag ("-", code);

end**if** code \in AfterNode **then**

| codeWithFlag ("+", code);

end

return codeWithFlag;

End Function**Function** ExtractIntersection (nodeCode, FullDiff(V_1, V_2))nc-diff $\leftarrow \emptyset$;foreach Line_{nc} \in nodeCode {Line_{diff} \in FullDiff {**if** Line_{nc} = Line_{diff} **then**| nc - diff.add(Line_{nc})**end**

}}

return nc-diff;**End Function**

```
JcaPEMKeyConverter converter = new JcaPEMKeyConverter();
Keys.add(new KeyPair(converter.getPublicKey(keyPair.getPublicKeyInfo()), null));
private KeyPair convertPemKeyPair(PEMKeyPair pemKeyPair) throws PEMException {
JcaPEMKeyConverter converter = new JcaPEMKeyConverter();
return new KeyPair(converter.getPublicKey(pemkeyPair.getPublicKeyInfo()), null);
```



```
JcaPEMKeyConverter converter = new JcaPEMKeyConverter();
Keys.add(new KeyPair(converter.getPublicKey(keyPair.getPublicKeyInfo()), null));
+private KeyPair convertPemKeyPair(PEMKeyPair pemKeyPair) throws PEMException {
+JcaPEMKeyConverter converter = new JcaPEMKeyConverter();
+return new KeyPair(converter.getPublicKey(pemkeyPair.getPublicKeyInfo()), null);
```

FIGURE 6.3: Diff Change After Optimization

6.3.2 Optimised Diff Encoder

Diff Encoder is the transformation of a diff into array data that can be trained by deep learning algorithms. Although such an encoding approach can be used for model training, there are still some problems, such as encoding confusion. In most cases the character lengths are sufficient to abstractly represent the relative positions of words in the array, but there are still cases where different words have the same character length, leading to encoding confusion, which affects the prediction accuracy of the model. Therefore, we have designed suitable optimisation approaches for the specificities of *Diff Structure Feature Model* and *Diff Feature Matching Model*, respectively.

Encoding Approach For Diff Features Network Network

For the encoding optimization approach applied to the diff feature matching network, we chose to highlight the abstract expression of the uniqueness of each word. As shown in the algorithm 5 First, the symbols and duplicate tokens contained in the diff are removed; then each non-duplicate token is assigned a unique value (the number of each word in the tokenList is used here); finally, the code for the removed and added parts of the diff is transformed into array_Remove and array_Add according to the flags ("- and "+").

Algorithm 5 Optimise Approach of Diff Encoder

Input diff with "+" and "-"

Output array_Remove and array_Add

Begin Body

1. tokenList $\leftarrow \emptyset$;
2. diff \leftarrow diff.removeSymbol
3. tokenList \leftarrow diff.removeDuplicates;
4. diff_a \leftarrow diff.startwith("-");
5. diff_b \leftarrow diff.startwith("+");
6. arrayA \leftarrow diff_a.convert(assign(tokenList));
7. arrayB \leftarrow diff_b.convert(assign(tokenList));
8. **return** array_Remove, array_Add

End Body

This is illustrated in Figure 6.4, which shows the encoding of the diff in Figure 6.3, whereby the lines in the diff are represented as columns in the array. The left-hand side shows the original encoding approach and the right-hand side shows the optimised representation. It can be seen that each word has a unique abstract expression except for "0", and the whole diff contains a total of 17 non-duplicated words, ensuring the uniqueness of the abstract expression for each word is significant in eliminating encoding confusion. As marked in green and yellow in the figure, the optimised encoding approach allows for the uniqueness of each word to be expressed. This approach ensures that the same tokens, such as variable or method names, in the code are also mapped to the same number. In the old approach, in contrast, different

Remove		Add			Remove		Add		
18	4	7	18	6	8	5	4	8	17
9	3	7	9	3	6	1	3	6	2
3	3	17	3	7	2	2	15	2	3
18	7	10	18	9	8	3	14	8	6
0	9	10	0	12	0	6	16	0	9
0	12	6	0	10	0	9	10	0	16
0	7	12	0	16	0	13	12	0	7
0	16	0	0	4	0	7	0	0	11
0	4	0	0	0	0	11	0	0	0

FIGURE 6.4: Example of uniqueness encoding

names were mapped to the same number already if they has the same length, which would lead to wrongly detecting similarity in some cases. Such clashes even become more likely the more different names are used in the analysed program.

Encoding Approach For Diff Structure Features Model

For the diff structured feature model, we chose to continue using character length as an abstract expression of the token, but we chose to add flags to each word after encoding it. Instead of encoding the diff as a grey-scale image, we used a two-dimensional array of uniform size. This allows more features to be retained, such as whether the value is negative or positive, in the array and avoids feature loss when resizing grey-scale image data.

Figure 6.5, the array after the example (Figure 6.3) encoding, the left side is the original encoding result and the right side is the optimised result. The enhancement of the flags to the structural features is more significant if the keywords are specially assigned. The reason for not using the unique encoding is that the structure feature model is trained on diffs of different refactoring types, and the structure features of each type need to be identified and extracted. The unique assignment approach in fact weakens the functionality of keywords as structure features, especially in some short diffs. For example, *RenameMethod*, if encoded according to unique values, would contain some of the same abstract assignments in each training data, and the model could then increase the weight of these same assignments, thus affecting accuracy, we have demonstrated with training results in the next section.

Remove + Add						Remove + Add				
18	4	7	18	6		-18	-4	7	18	6
9	3	7	9	3		-9	-3	7	9	3
3	3	17	3	7		-3	-3	17	3	7
18	7	10	18	9		-18	-7	10	18	9
0	9	10	0	12	→	0	-9	10	0	12
0	12	6	0	10		0	-12	6	0	10
0	7	12	0	16		0	-7	12	0	16
0	16	0	0	4		0	-16	0	0	4
0	4	0	0	0		0	-4	0	0	0

FIGURE 6.5: Optimisation Result

6.4 Training

Following the optimisation approach described in the previous section, we divided the 6048 refactoring samples into two datasets, a training set and a test set. Since each refactoring type has a different amount of sample data, in particular *ExtractMethod* and *MoveClass*, are much more numerous than the other refactoring types. To ensure that the amount of data for each refactoring type was of the same order of magnitude, we limited the amount of training data to between 400 and 500 so that the model would not have a biased amount of data, the distribution of which is shown in Table 6.1. From the 4459 refactoring instances used for training, we extracted their diffs using the optimised Diff Extractor, and then trained the diff structure feature model and the diff feature matching network respectively using different encoding approaches. Among the 10 refactoring types, the diffs of *MoveMethod*, *PullUpMethod* and *PushDownMethod* are identical, but the difference is the relationship between the class of the method being moved. Therefore, we decided to keep only *MoveMethod* as training data during the training of the model to ensure a balance in the number of different refactoring types. In the final evaluation, when testing *PullUpMethod* and *PushDownMethod*, the condition $RT_a = RT_s$ was considered to hold as long as the prediction of the model was *MoveMethod*.

6.4.1 Optimised Diff Features Marching Network

Model Training

Data Preparation. First, we extracted each refactoring diff by using the optimised Diff Extractor, and then placed the removed and added parts of the

TABLE 6.1: Data Distribution

Refactoring Type	Data Pool	Training Data	Test Data
Extract Interface	440	416	24
Extract Method	1037	488	549
Extract Superclass	459	409	50
Inline Method	491	420	71
Move Class	1051	500	551
Move Method	528	463	65
Pull Up Method	432	401	31
Push Down Method	460	422	38
Rename Class	518	454	64
Rename Method	632	486	146
Total	6048	4459	1589

diff, in two different folders, with the same file name. We then paired these files with the same filename and used them as the dataset labelled **Match**. We then randomly paired these files, ensuring that each pair has a different file name, to form a dataset labelled **Non-Match**. Non-Match and Match contain the same amount of data. We then encoded the data using an optimised Diff Encoder and divide the data set in a ratio of 8:2 between the training and validation sets, with Non-Match having the same amount of training and validation data as Match. Finally, the training preprocessing was completed.

The diff feature matching network consists of a sampling layer, a feature extraction network, a metric network and a cross-entropy loss function. The structure of the diff feature matching network is shown in Figure 6.6.

During the data preparation phase, we have completed work on the sampling layers. First, the feature extraction network was constructed, which is based on the classical convolutional neural network *AlexNet* (Krizhevsky Alex, 2015), but with some changes to it by adding a preprocessing layer and a bottleneck layer. The former was used to normalise the data and the latter was used to control the dimensionality of the feature vectors fed from the feature extraction network to the fully connected layer in order to avoid overfitting. During the training phase, the feature network acts as a two-stage pipeline, sharing parameters, with the outputs of the two towers connected together as the input to the metric network. In the Metric network, three fully connected layers with ReLU nonlinearity and a Softmax function were set for feature comparison, and the output was obtained as two values of the probability $\in [0, 1]$ that the two encoded arrays are similar, i.e., the probability of a match. Finally, a cross-entropy loss function with a learning rate of 0.0001 was set to evaluate the performance of the diff feature matching network, which was trained for a total of 20 epochs. With the same training and validation data, we trained a matching network following the original encoding approach as a comparison to show the effect of the optimisation approach on the performance of the matching network.

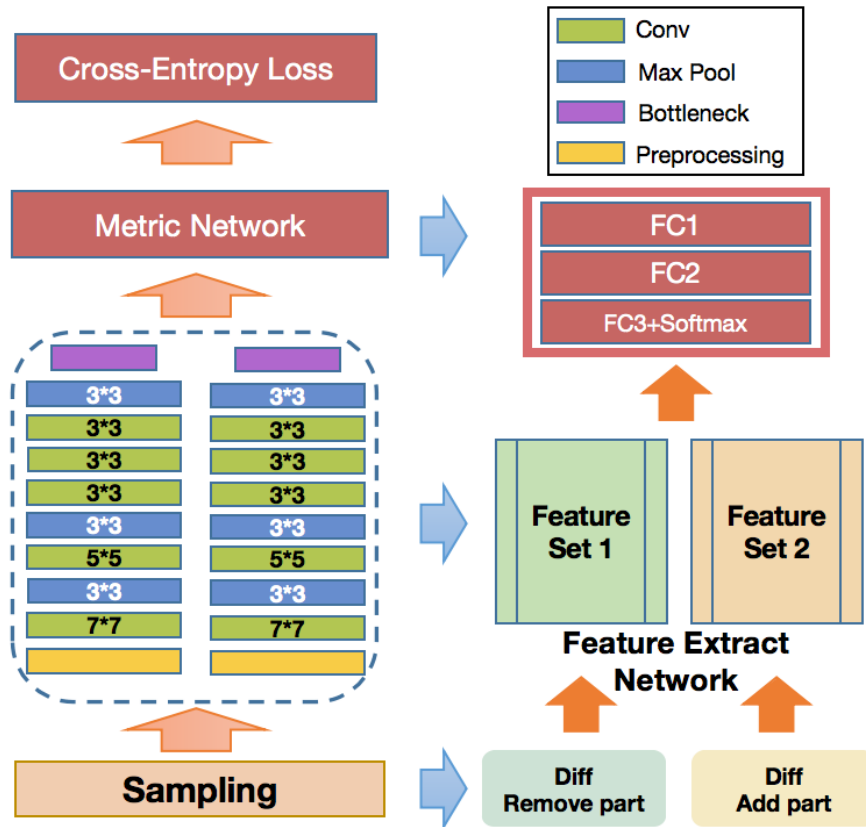


FIGURE 6.6: Matching Network

Model Evaluation

Figure 6.7 and Figure 6.8 show the impact of the encoding approach on the diff feature matching network before and after optimisation.

As can be seen in the Figures, the accuracy of the training set is 0.9926 (new) and 0.9906 (old), and the accuracy of the validation set is 0.9374 (new) and 0.8637 (old), an improvement of 7.37 percentage points over the original approach. The loss values for the training set were 0.0525 (new) and 0.0532 (old) and the loss values for the validation set were 0.2672 (new) and 0.3986 (old), with the loss values for the optimised coding method being lower than the original approach by 0.1314. The model converges faster and is smoother when the optimised encoding approach is applied, especially in the validation set, with higher accuracy and lower loss values. Furthermore, the model is not overfitted. Altogether this demonstrates the positive impact of the optimised encoding approach on the model.

To prove that this was not a coincidence, we built different training and validation sets randomly to train the model a total of ten times. The training results showed that the accuracy and loss values for the training set of the optimised and old approach were very close and always converged. The former continues to converge faster and more smoothly. For the validation set, the average accuracy of the optimised approach is 6.988 percentage points higher and the average loss value is 0.1363 lower than the old approach. The average time spent per training is 27 minutes.

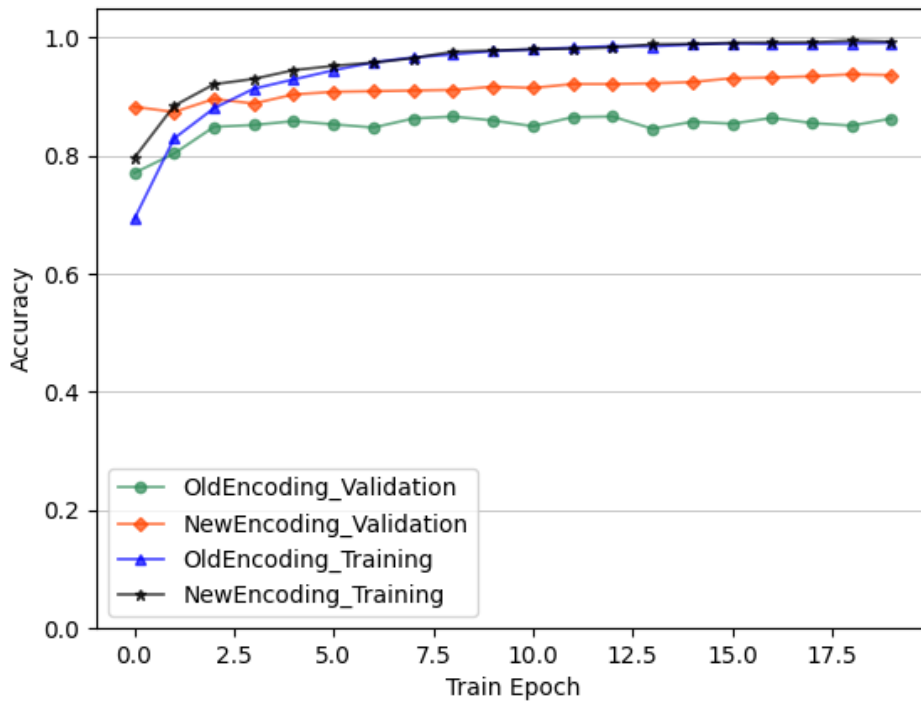


FIGURE 6.7: Matching Network

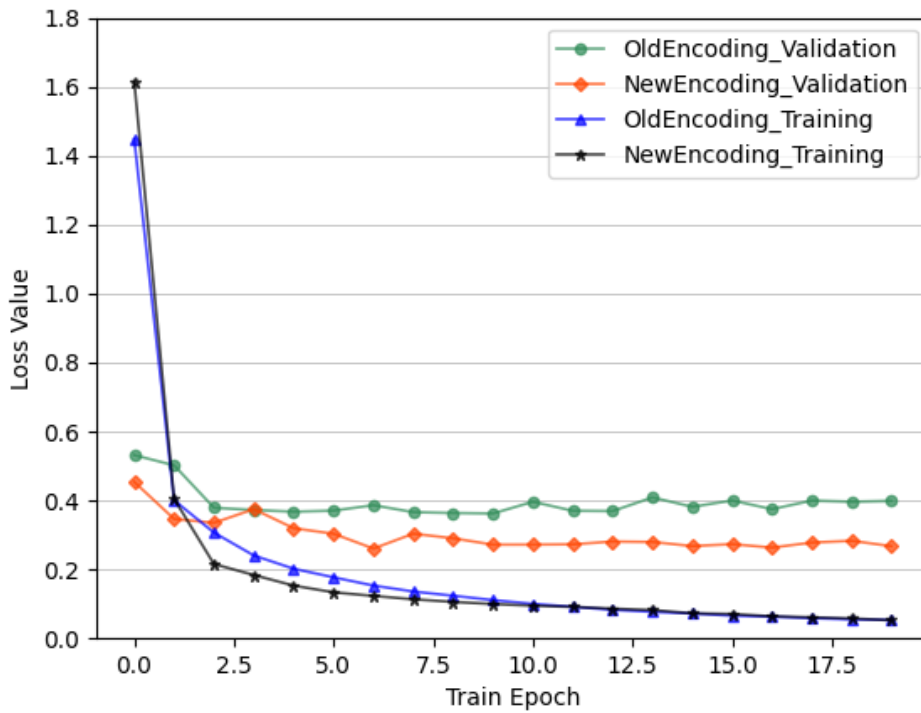


FIGURE 6.8: Matching Network

In order to show the effect of the optimised approach on the stability of the model, a control experiment was designed, where the experimental test consisted of RefDiff's word frequency similarity and the matching network of the encoding method before and after optimisation. The subject of the control experiment was an ExtractMethod instance, which was judged as a false negative by RefDiff because the similarity threshold was 0.338710, which was less than the threshold of 0.5 normally used by RefDiff. The diff of this refactoring instance contained a large number of non-refactored code changes, so it led to a false negative, this was used to verify the stability of the old and new feature matching network. We used increasing noise (additional non-refactored code) to verify this, and as the noise increased, the similarity based on word frequency was decreasing. By adding 20 lines of noise, the similarity had dropped to 0.139861, while both matching networks consistently maintained a matching probability close to 1, demonstrating that the matching networks possess stronger robustness.

6.4.2 Optimised Diff Structure Features Model

Model Training

The structure feature model supports the detection of eight refactoring types (in addition to the PullUpMethod and PushDownMethod). These refactoring diffs are extracted and encoded, and then randomly divided into a training and validation set in a ratio of 8:2. Based on the single-channel data nature of the diff array, we therefore chose to use the deeply separable convolution *mobileNetV1*, which does not require inter-channel information fusion, as the trainer for 100 batches. As with the diff feature matching network, we set up a control group using the original encoding approach.

Model Evaluation

Figure 6.9 and Figure 6.10 show the changes in the accuracy and loss values of the old and new models before and after optimisation.

As can be seen from the accuracy trends, the optimised approach converges faster, the accuracy of the training set remains smooth after convergence, and the convergence of the loss values of the optimised approach is also lower and smoother. The performance of the model in the optimised approach is better than the original approach. Moreover, the curves of the training and validation sets of the optimisation approach are very stable after convergence, and the validation set does not suffer from accuracy degradation, indicating that the model is not overfitted. Also to demonstrate the non-accidental performance of the model, we trained it ten times based on different training and validation data, and the training results converged in all cases. Therefore, the advantages of the optimisation were shown in each case. The training time for the optimised approach is longer (28 hours, about 3 hours for the original approach) because the training input for the optimised approach are 224*224 arrays, whereas the input for the original approach are 50*50 grey-scale images, the former being more computationally

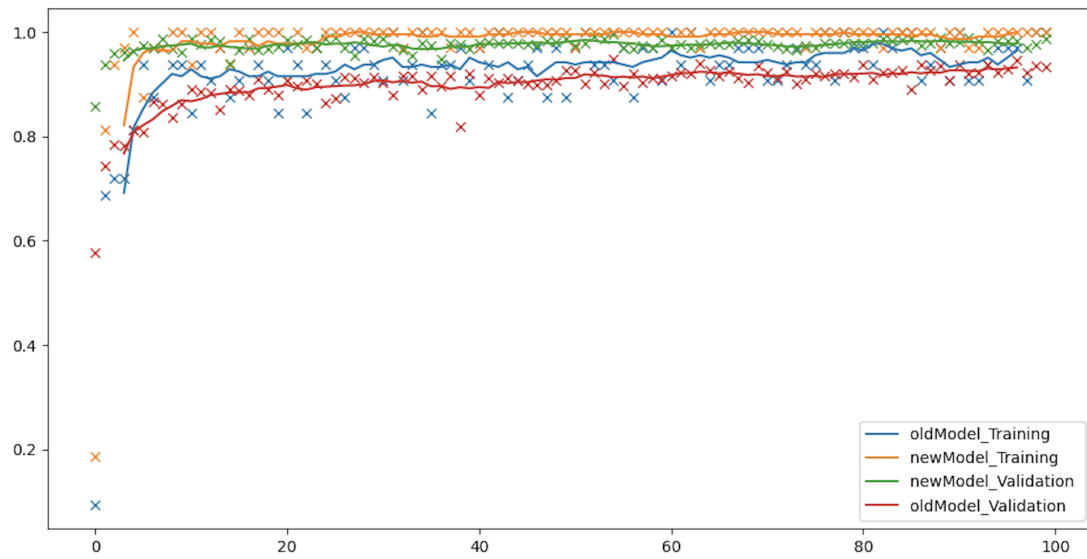


FIGURE 6.9: Accuracy Chart

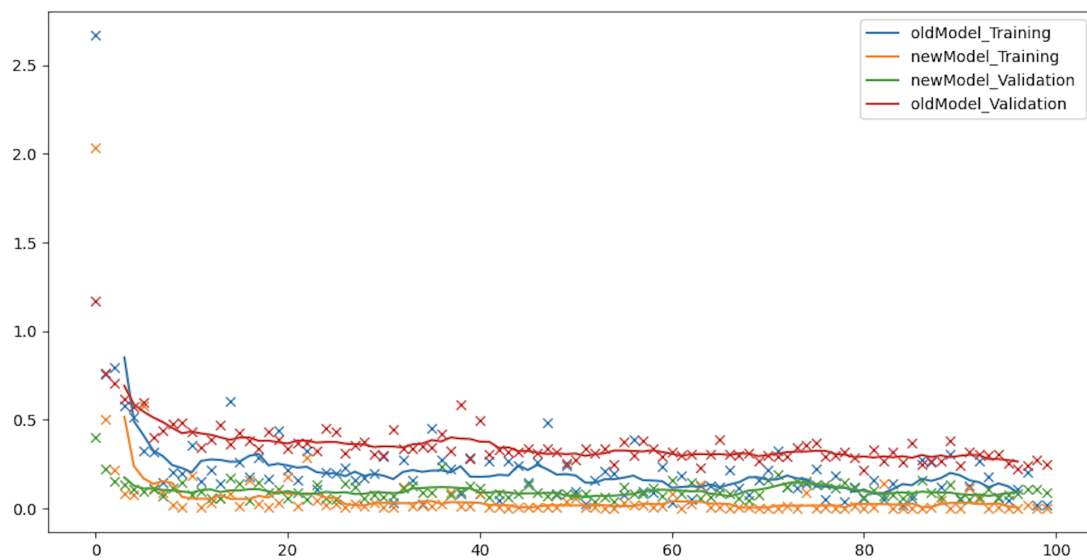


FIGURE 6.10: Loss Chart

TABLE 6.2: Evaluation of experimental results

Test Situation	Original Approach	Optimised Approach
Original	0.996934	0.999958
Noise +3	0.997701	0.999976
Noise +5	0.998075	0.999974
Noise +10	0.918348	0.999977
Noise +15	0.865232	0.999972
Noise +20	0.786634	0.999979
Noise +30	0.758181	0.999967
Noise +50	0.619542	0.999977

intensive. In addition, we have experimented with the approach based on uniqueness encoding. The encoded diff is trained and although it converges successfully, the accuracy and loss values of the validation set continued to oscillate after convergence and the model performance was not stable, so it justifies the choice of our optimisation solution.

In the same way as the evaluation of the diff feature matching network, a control experiment was designed for the diff structural feature model to compare the effect of the optimisation approach on the stability of the model. The experiment verifies the stability of the models based on the original and optimised approaches by using the same *ExtractMethod* instance and adding noise code to the added part of the diff. The experimental results are shown in Table 6.2. The model corresponding to the original approach is the one in which the prediction probability decreases under the influence of more noisy codes, although all the detection results are correct, the decrease is significant. In contrast, the model corresponding to the optimisation approach is more robust in the face of noisy code, as the prediction results are correct and the prediction probabilities are all very stable. This robustness is also due to the enhancement of the structure features by the "-" flags.

Generalized Applications

To demonstrate the general applicability of our research approach to code data mining, we also collected refactoring data for the programming languages C and JavaScript, with training data taken from the Ref-Diff database¹. For JavaScript, we trained models supporting eight refactoring types using 1600 refactoring instances. For C, we trained models supporting six refactoring types using 1200 refactoring instances. We used these data to train the corresponding diff structure feature models for C and JavaScript, and the accuracy trend of the training set is shown in Figure 6.11. Both models converged more slowly to the accuracy curve than the Java model, completing convergence after 30 batches, mainly due to insufficient training data. So the model's parameter weights were calibrated relatively slowly. Since the diff feature matching network does not require special encoding of keywords, a matching network trained on java refactored diff data can support matching

¹<https://github.com/aserg-ufmg/RefDiff>

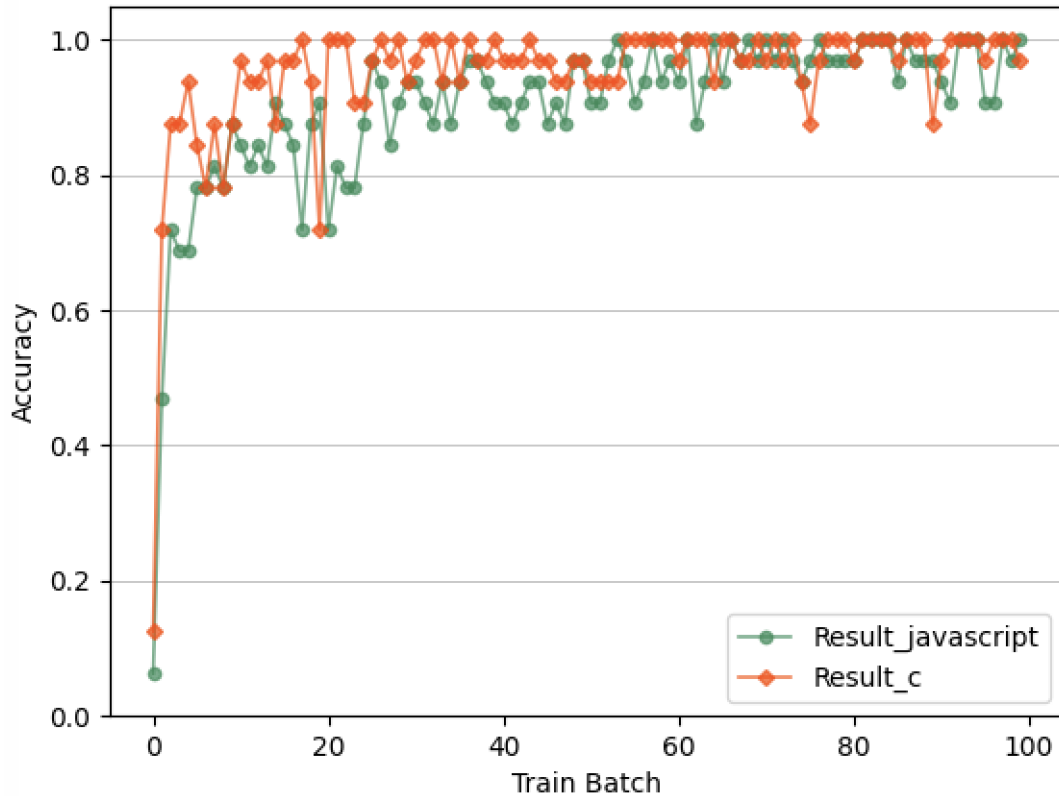


FIGURE 6.11: Diff Convert to Array

work for other programming languages, and we show the matching results in the next section.

6.5 Evaluation

The research problem addressed in this chapter is to optimise Diff Extractor and Diff Encoder to jointly apply *Diff Structure Feature Model* and *Diff Feature Matching Network* to implement an detection approach for mining the diff code about information of refactoring. In this section we will test and evaluate the entire optimisation solution, including performance, efficiency, validity threats, and limitations.

In practice, when using our solution to detect some submissions with $threshold \in [0.1, 0.5]$, hundreds of refactoring candidates will be matched, most of which are false positives, and it is a huge burden to match them one by one using the diff feature matching network. So we set up a diff-only word frequency similarity filtering layer before inputting candidate diffs into the matching network. The major difference between this filtering layer and RefDiff's similarity is that it only calculates word-frequency similarity for the removed and added parts of extracted diff from Diff Extractor, rather than for all code similarities in the node. And, as with the diff encoder, it removes all symbols and retains only the more representative words as token. The main purpose of this filter layer is to filter out most of the false candidates that are not related to refactoring. We set its threshold to 0.2 for two main reasons: (1)

diff similarity is more reflective of refactoring behaviour than node-all code similarity, so the diff similarity of true refactorings is usually higher than the latter. (2) The threat of non-refactored code changes on the diff similarity filtering layer is still present, so 0.2 is a conservative choice to ensure that true refactoring candidates are not filtered out as much as possible.

The test data were 1589 refactoring instances out of the training set in Section 6.4. The approaches tested included RefDiff 2.0, a result checker based on the diff structural feature model, a single diff feature matching network, and our comprehensive optimisation solution (RefDiff-Model). The test results are shown in Table 6.3. In addition, at the same benchmark we also tested RefactoringMiner 2.0 as a reference benchmark, with 99.6% for precision and 96.9% for recall. We also tested the performance of the RefDiff and optimisation solutions compared in C and JavaScript, as shown in Table 6.4 and Table 6.5.

TABLE 6.3: Test Result

Refactoring Type	#	RefDiff 2.0		RefDiff 2.0 with Checker (0.1)		RefDiff 2.0 with MatchNet		RefDiff-Model	
		Precision	Recall	Precision	Recall	Precision	Recall	Precision	Recall
Extract Interface	24	0.875	0.875	1.000	0.917	N/A	N/A	1.000	0.917
Extract Method	549	0.967	0.696	0.996	0.943	0.971	0.929	1.000	0.947
Extract SuperClass	50	1.000	0.820	1.000	0.840	N/A	N/A	1.000	0.920
Inline Method	71	0.983	0.732	0.985	0.915	0.957	0.930	1.000	0.958
Move Class	551	1.000	0.980	1.000	0.978	0.993	0.989	1.000	0.993
Move Method	65	0.845	0.754	0.966	0.877	0.952	0.923	1.000	0.954
Pull Up Method	31	0.964	0.871	1.000	0.968	1.000	0.968	1.000	0.968
Push Down Method	38	0.974	0.974	1.000	0.974	1.000	0.974	1.000	0.974
Rename Class	64	0.935	0.906	0.968	0.938	0.953	0.953	1.000	0.969
Rename Method	146	0.908	0.725	0.992	0.890	0.943	0.904	1.000	0.911
Total	1589	0.965	0.828	0.995	0.945	0.977	0.951	1.000	0.961

TABLE 6.4: Comparison in C

Refactoring Type	#	RefDiff 2.0		RefDiff-Model	
		Precision	Recall	Precision	Recall
Extract Function	20	0.889	0.800	1.000	1.000
Inline Function	20	0.773	0.850	1.000	1.000
Move File	20	1.000	1.000	1.000	1.000
Move Function	20	0.810	0.850	1.000	0.950
Rename File	20	1.000	1.000	1.000	1.000
Rename Function	20	0.864	0.950	1.000	0.950
Total	120	0.886	0.908	1.000	0.983

6.5.1 Result Analysis

The most striking performance of the optimised solution in the Java data test was the achievement of 100% precision, which proved to satisfy the condition $\{RT_a = RT_s \wedge Match\}$ and was almost completely certain to be a refactoring. Compared to RefDiff, the optimised solution increased precision by 3.5 percentage points and recall by 13.3 percentage points. Meanwhile, 36 candidates were judged to be manually intervening checks during the test and they were not counted in the results. Although the judgement of manual

TABLE 6.5: Comparison in JavaScript

Refactoring Type	#	RefDiff 2.0		RefDiff-Model	
		Precision	Recall	Precision	Recall
Extract Function	20	0.889	0.800	1.000	1.000
Inline Function	20	0.824	0.700	1.000	1.000
Move Class	20	1.000	1.000	1.000	1.000
Move File	20	1.000	1.000	1.000	1.000
Move Function	20	0.952	1.000	1.000	1.000
Rename File	20	0.950	0.950	1.000	1.000
Rename Class	20	0.950	0.950	1.000	0.950
Rename Function	20	0.833	0.750	1.000	0.950
Total	160	0.929	0.894	1.000	0.988

intervention is subjective, it offers the possibility of further improving detection performance. RenameMethod was the type with the lowest recall, and some of the results were judged to be manual interventions, mainly due to the presence of many controversial "refactorings" in the test data, where the content of the refactorings had been changed. The method names were renamed and the contents of the methods changed, completely losing the correspondence between the removed and added parts of the refactoring diff, leading to errors in the diff feature matching network.

Compared to the single model-based result checker and the diff feature matching network, the optimised solution combines their strengths and weakens their drawbacks when relied upon individually. The result checker based on the diff structure feature model is a significant improvement in precision. In the event of a false prediction, the match result of the diff feature matching network will act as insurance that the result can be imported into the manual intervention, avoiding false negatives as much as possible. Similarly, the improvement in recall by the diff feature matching network is significant. If a true refactoring is judged by the matching network to be a mismatch, the diff structural feature model will also import that result into the manual intervention.

For the C and JavaScript tests, we replaced the diff structure feature model with the model of the corresponding programming language. The results show that the detection performance of the optimised solution is remarkable, as in the java test, with a consistent 100% precision along with a boosted recall, which also shows that the optimised solution is also effective in the detection for other programming languages.

Running time We trained the model using a computer with a 2.3 GHz Intel Core i5 processor and 8 GB of 2133 MHz LPDDR3 memory. The average matching time per candidate for the diff feature matching network in python was 245 ms and the average matching time for the diff structure feature model was 275 ms.

6.5.2 Threats to Validity

For this section the validity threat mainly comes from the data used in the evaluation, clearly `ExtractMethod` and `MoveClass` are far more numerous than the other types, possibly implying a greater weighting on the overall performance when the optimised solution detects them. Although there was less data for the other types tested, the optimised solution detection performance still showed a significant improvement, and the training set is in the same order of magnitude for the various refactoring types when training the model, and the final model accuracy was converging to 1, showing that the detection results can be trusted. For the C and JavaScript tests, there is less test data.

6.5.3 Challenges and Limitations

The solution, while incorporating the advantages of RefDiff's parsing match and both models, also inherits two limitations.

Limitation 1: RefDiff's CST only focuses on large code structure nodes, so `Field` cannot be parsed out and refactoring types about `Field` cannot be detected. We are trying to improve the parsing and matching algorithm, and we have confirmed that we can use `Field`-related refactoring type diffs as training labels for both models (*MoveField*, *RenameField*, etc.) with high recognition accuracy. In addition, some refactoring types are implemented with purely added or removed code (*Introduce Assertion*, *Remove Setting Method*, etc.), and there is no correspondence between the added and removed part of diff, so the *Diff Feature Matching Network* cannot match them. However, *Diff Structure Feature Model* can still work, as their special structure features are still present and recognizable.

Limitation 2: *Diff Structure Feature Model* cannot classify nested refactorings such as *Extract and Move Method*, *Rename and Move Class*, etc. Since the structure feature model classifies refactorings according to the specificity of the structure of each refactoring type, confusion arises between the diff structure of some nested refactoring types and the structure of a single refactoring type, e.g. the diff structures of *Extract and Move Method* and *ExtractMethod* are the same, but *Rename and Move Class* is different from the structure of *MoveClass* or *RenmeClass*. This phenomenon can cause confusion during model training, which needs to be studied in more depth.

6.6 RefDiff-Model with Random Forest Model

In order to try to solve the problem that RefDiff-Model cannot detect nested refactorings, we tried to fuse Chapter 3's random forest probability model with RefDiff-Model, replacing the rules for manually defining refactoring types, and finally implementing a function that supports nested refactoring detection.

6.6.1 Implementation

Artificial abstraction features are simple abstractions of the information contained in a node. Information about a refactoring candidate node includes: NodeType, Namespace, parentNode, NodeName, Locations, Relationships, etc. This information allows the refactoring type to be determined based on a manual definition. But instead of using the traditional way of definition, we abstract this information into an array, and the label of each array is the corresponding refactoring type, as shown in the Figure 6.12, using the java code node as an example.

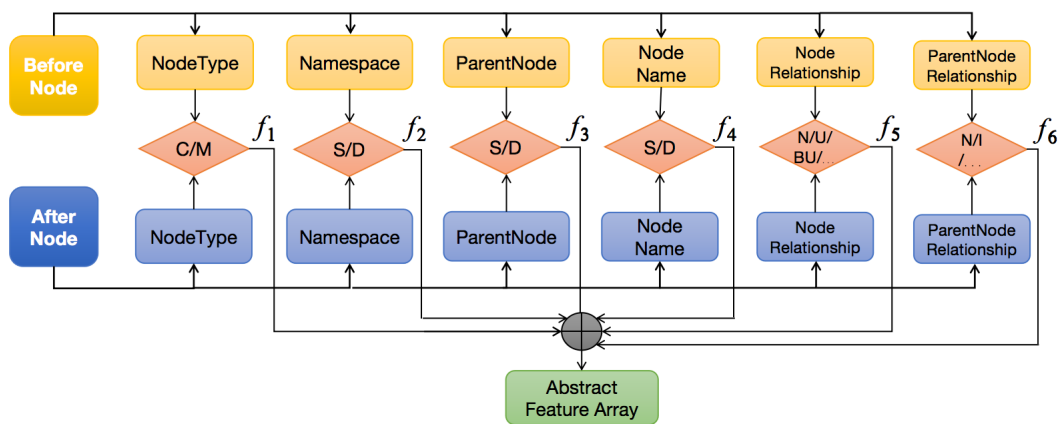


FIGURE 6.12: Process of Features Extraction

We extract the node type, the package the node is in, the class the node is in, the node name, the node relationship and the node parent's relationship directly from the node information of the matched refactoring candidate nodes BeforeNode and AfterNode.

- Feature f_1 is to confirm that the node is a Class node or a Method node, as only nodes of the same type can match each other, and this feature identifies the type of class-level or method-level at which refactoring occurs.
- Feature f_2 is the description of the package in which the node is located that can be obtained based on Namespace, which can determine whether the packages in which two nodes are located are the same or different, and is used to determine whether the refactoring has been moved or replaced in a package.
- Feature f_3 is the parent of the node, in the case of a method node the description of the class in which it is located, and in the case of a class node the description of the package in which it is located, which can be used to determine whether the refactoring has caused the node to move.
- Feature f_4 is the node signature, which is used to determine whether the signatures of the two matching nodes have changed.

- Feature f_5 is a description of the node relationships, which can be SUB-TYPE etc. between class nodes, and USE or be USE between method nodes.
- Feature f_6 is similar to f_5 and is the relationship between parentnodes.

These six features can form an array of abstract features ($f_1, f_2, f_3, f_4, f_5, f_6$), which can contain almost all the abstract information for determining candidate node pairs, and can support the detection of at least ten method-level refactoring types (as Ref-Diff supports ten refactoring types: Move Method, Extract Method, Inline Method, Rename Method, Move Class, etc.).

This design can significantly reduce the amount of work required by researchers to define each refactoring. When dealing with refactorings in other programming languages, the detection of other languages can be done by simply replacing the abstract array model of the corresponding programming language. Since the data structures of all nodes are similar regardless of the programming language being processed, the representation of the abstract features of the refactoring candidate nodes is the same, despite the different declarative languages of the nodes obtained from the different parses.

6.6.2 Train Abstract Features

According to the approach in section 5, each refactoring abstract feature array consists of six abstract feature value (the feature arrays correspond to the Table 6.6), each labelled with the corresponding refactoring type. This list of assignments shows the values assigned to each abstract feature, with the following details:

- Node Type. When the candidate is method-level, it is assigned a value of 1.0; when it is class-level, it is assigned a value of 1.5, or 1.7 if the class node type declaration is different.
- Namespace. 2.0 if namespace is the same, 2.5 if different.
- Parent Node. 3.0 if Parent Node is the same, 3.5 if different.
- Node Name. 4.0 if Node Name is the same, 4.5 if different.
- Node Relationship. 5.0 if there is no relationship between nodes. 5.3 if BeforeNode uses AfterNode, 5.7 if the opposite is true. 5.5 if BeforeNode is a subclass of AfterNode.
- Parent Node Relationship. 6.0 if there is no relationship between ParentNodes. 6.3 if BeforeNode's parentNode is a child of AfterNode's parentNode, and 6.7 if the opposite is true.

For example, if one were to interpret an array of abstract features in light of java structural properties, RenameMethod [1.0, 2.0, 3.0, 4.5, 5.0, 6.0], this array implies that this refactoring candidate node pair is composed of two method nodes, both nodes are in the same package, both nodes are in the

TABLE 6.6: Abstract Feature Assignment

Refactoring Type	Abstract Feature Assignment
NodeType	Method-level(1.0) Class-level(1.5(S), 1.7(D))
Namespace	Same(2.0) Different(2.5)
ParentNode	Same(3.0) Different(3.5)
NodeName	Same(4.0) Different(4.5)
Node Relationship	USE(5.3) Be USE(5.7) SUBTYPE(5.5)
ParentNode Relationship	SUBTYPE(6.3) Be SUBTYPE(6.7)

TABLE 6.7: Feature Arrays Chart

Refactoring Type	Abstract Feature Arrays
Extract Interface	[1.7, 2.0(2.5), 3.0(3.5), 4.0, 5.5, 6.0]
Extract Method	[1.0, 2.0, 3.0, 4.5, 5.3, 6.0]
Extract Superclass	[1.5, 2.0(2.5), 3.0(3.5), 4.0, 5.5, 6.0]
Inline Method	[1.0, 2.0, 3.0, 4.5, 5.7, 6.0]
Move Class	[1.5, 2.5, 2.5, 4.0, 5.0, 6.0]
Move Method	[1.0, 2.0(2.5), 3.5, 4.0, 5.0, 6.0]
Pull Up Method	[1.0, 2.0(2.5), 3.5, 4.0, 5.0, 6.3]
Push Down Method	[1.0, 2.0(2.5), 3.5, 4.0, 5.0, 6.7]
Rename Class	[1.5, 2.0, 3.0, 4.5, 5.0, 6.0]
Rename Method	[1.0, 2.0, 3.0, 4.5, 5.0, 6.0]

same class, both nodes have different node names, both nodes are not related, and both nodes' parents are not specifically related. The feature arrays corresponding to the other refactoring candidates are shown in Table 6.7. In the table, the values in brackets indicate that the array of abstract features we have collected in the training set is not fixed. MoveMethod, for example, the method can be moved to other packages, or to different classes of the same package.

We randomly divided the 4459 training data into two parts according to 80% of the training group (3567) and 20% of the validation group (892). The results were then trained and validated using four machine learning algorithms, KNN, decision tree, random forest and SVM, as shown in Table 6.8.

Model evaluation. The training results for the abstract feature arrays using the four machine learning algorithms all achieved 100% accuracy, indicating that the abstract feature arrays corresponding to the different refactoring types are well differentiated and the trained models are very good at recognising these verification sets.

The results show that it is feasible to train and classify refactoring types using machine learning algorithms, and we recorded the probability when each refactoring type was predicted under random forest model, ExtractMethod (51.0%), InlineMethod (51.0%), PullUpMethod (59.8%), PushDownMethod (59.8%), RenameMethod (51.0%), MoveMethod (100.0% or 67.3%),

TABLE 6.8: Feature Arrays Chart

Refactoring Type	KNN	Decision Tree	Random Forest	SVM
Extract Interface	83	83	83	83
Extract Method	98	98	98	98
Extract Superclass	82	82	82	82
Inline Method	84	84	84	84
Move Class	100	100	100	100
Move Method	93	93	93	93
Pull Up Method	80	80	80	80
Push Down Method	84	84	84	84
Rename Class	91	91	91	91
Rename Method	97	97	97	97
Accuracy	100%	100%	100%	100%

MoveClass (85.7%), RenameClass (69.3%), ExtractSuperclass (69.3%), Extract-Interface (64.3% or 61.5%). Since fixed features correspond to fixed prediction probabilities, we can therefore distinguish between single refactorings and nested refactorings, based on this phenomenon.

Move Method and *Extract Interface* have two results, because these two refactoring types are in actual operation, may be moved and extracted to other packages or still in the original package, resulting in differences in feature extraction, however, this does not affect the correct result of the probabilistic model.

In addition, we chose the six abstract information of the candidates mainly because they are sufficient to support the training of ten refactoring types, although other information of the candidates can be used to increase the dimensionality of the abstract features. For example, there is information such as SimpleName and Parameter in the candidate abstract information, and if they are added to the feature array, they can support the classification ChangeSignature refactoring.

6.6.3 Application in Nested Refactoring

In the test data in the previous section, we found that there are four nested refactoring types:

- Extract and Move Method
- Inline and Move Method
- Rename and Move Method
- Rename and Move Class

We used a random forest model trained on the single refactoring data to detect the feature arrays of nested refactorings and could find that using the model to compute the prediction probabilities obtained from nested refactorings was different from the single refactoring prediction probabilities. The

TABLE 6.9: Nested Refactoring Probability

Java Refactoring Type	Features	Probability
Extract and Move Method	[1.0, 2.0, 3.5, 4.5, 5.3, 6.0]	EM(33.3%),MM(66.6%)
	[1.0, 2.5, 3.5, 4.5, 5.3, 6.0]	EM(39.9%),MM(34.0%)
Inline and Move Method	[1.0, 2.0, 3.5, 4.5, 5.7, 6.0]	IM(33.3%),MM(66.6%)
	[1.0, 2.5, 3.5, 4.5, 5.7, 6.0]	IM(39.9%),MM(34.0%)
Rename and Move Method	[1.0, 2.0, 3.5, 4.5, 5.0, 6.0]	RM(33.3%),MM(66.6%)
	[1.0, 2.5, 3.5, 4.5, 5.0, 6.0]	RM(39.9%),MM(34.0%)
Rename and Move Class	[1.5, 2.5, 3.0, 4.5, 5.0, 6.0]	RC(50.0%),MC(35.7%)

features of both refactoring types that make up the nested refactoring are mapped in the prediction probabilities, and only random forest model from the four machine learning algorithms in the previous section can accomplish this, while the other three algorithms cannot detect nested refactoring without a corresponding training label.

The random forest model was used to predict the four nested refactoring types contained in the data pool, and the results are shown in Table 6.9, the top two positions of each prediction ranking are the types that make up the nested refactorings.

6.6.4 Model Embedment

The most significant job of the embedding model is to replace the manually defined rules with a random forest model, allowing the processing of abstract information to support the detection of nested refactorings. As shown in Figure 6.13, the embedded model is not only a replacement, but also requires confirmation of Diff Model that support for nested refactoring diff. It has been illustrated in the figure that the Diff Feature Matching Network is supportive of nested refactoring because the code content before and after nesting is the same as single refactorings, despite the change in structure when nesting, so the matching network supports matching of nested diffs.

For the Diff Structure Feature Model, the training labels are all single refactorings, and due to the lack of training data, it is not possible to use the nested refactor types as training labels for the time being. However, we have tried to detect nested refactoring data using a model trained on single refactoring data, and the results show that the model will almost always select the labels containing 'strong features' as the predicted result of nested refactoring. For example, 'package' and 'class' are both keywords and structural features, but in the training data 'package' only appears in the MoveClass, while 'class' will appear in both MoveClass and RenameClass. So when we detected 'Rename and Move Class', the prediction of Diff Structure Feature Model always is MoveClass.

Based on this phenomenon, we propose a hypothesis, $\{RT_s \in RT_{RM} \wedge Matching\}$, that is, if the Random Forest Model determines nested refactoring based on the abstract information of the nodes, the prediction of Diff Structure Feature Model is one of the component types of the nested refactorings, and the Diff Feature Matching Network also outputs a match, then

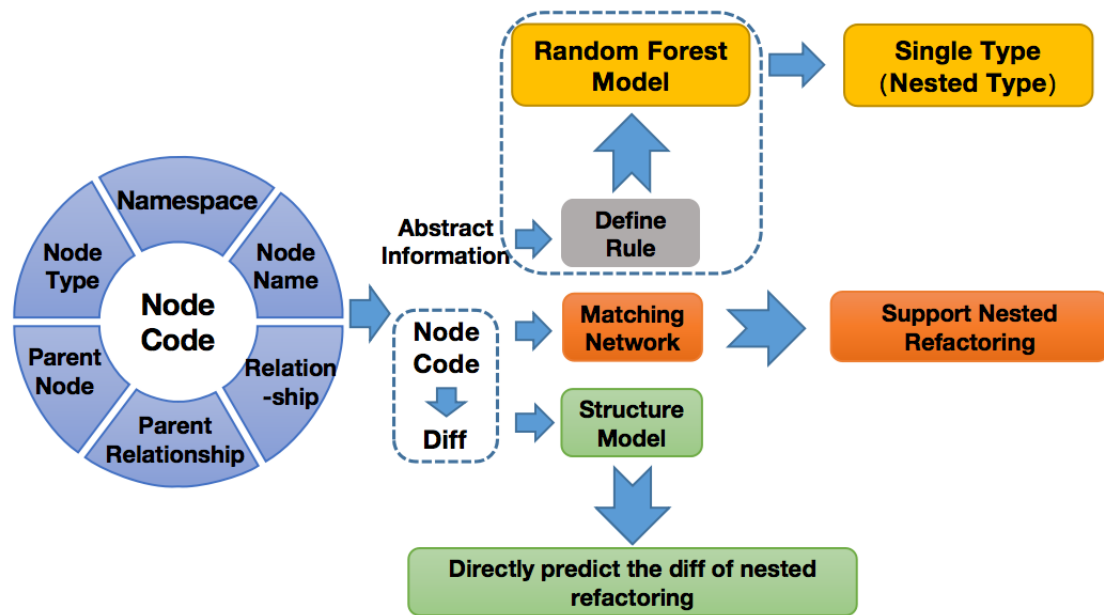


FIGURE 6.13: Model Fusion

the nested refactoring detection holds. Based on this hypothesis, we tested 134 instances of nested restructuring, of which 129 were determined to be nested refactorings and 5 were judged to be manual interventions.

Bibliography

- Andrew G. Howard, Menglong Zhu etc (Apr. 2017). “MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications”. In: URL: [arXiv:1704.04861](https://arxiv.org/abs/1704.04861).
- Beat Fluri, Michael Würsch etc (Nov. 2007). “ChangeDistilling: Tree Differencing for Fine-Grained Source Code Change Extraction”. In: *IEEE Transactions on Software Engineering* 33, 11, pp. 725–743. URL: <https://doi.org/10.1109/TSE.2007.70731>.
- Benjamin Biegel, Quinten David Soetens etc (2011). “Comparison of Similarity Metrics for Refactoring Detection”. In: *In Proceedings of the 8th Working Conference on Mining Software Repositories (MSR '11)*. ACM, New York, NY, USA, pp. 53–62. URL: <https://doi.org/10.1145/1985441.1985452>.
- Breiman, Leo (Aug. 1996). “Machine Learning”. In: 24(2), 123–140. URL: [doi: 10.1023/A:1018054314350](https://doi.org/10.1023/A:1018054314350).
- (2001). “Random forests”. In: *Machine Learning* 45(1), 5–321.
- Broder, Andrei (1997). “On the resemblance and containment of documents”. In: *SEQUENCES'97: Proceedings of Compression and Complexity of Sequences*, pp. 21–29.
- Costa, Shane McIntosh etc Daniel Alencar da (2017). “A Framework for Evaluating the Results of the SZZ Approach for Identifying Bug Introducing Changes”. In: *IEEE Transactions on Software Engineering* 43, 7, pp. 641–657. URL: <https://doi.org/10.1109/TSE.2016.2616306>.
- D. Janzen, K. D. Volder (2003). “Navigating and querying code without getting lost”. In: *In AOSD*, pp. 178–187.
- Danilo Silva, Joao Paulo da Silva etc (2020). “RefDiff 2.0: A Multi-language Refactoring Detection Tool”. In: *IEEE Transactions on Software Engineering*.
- Danilo Silva, Marco Tulio Valente (2017). “RefDiff: Detecting Refactorings in Version Histories”. In: *In Proceedings of the 14th International Conference on Mining Software Repositories (MSR '17)*. IEEE Press, Piscataway, NJ, USA, pp. 269–279. URL: <https://doi.org/10.1109/MSR.2017.14>.
- Danilo Silva, Nikolaos Tsantalis etc (2016). “Why We Refactor? Confessions of GitHub Contributors”. In: *In Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE '16)*. ACM, New York, NY, USA, 858–870. URL: <https://doi.org/10.1145/2950290.2950305>.
- Danny Dig, Can Comertoglu etc (2006). “Automated Detection of Refactorings in Evolving Components”. In: *In Proceedings of the 20th European Conference on Object-Oriented Programming (ECOOP '06)*, Springer-Verlag, Berlin, Heidelberg, pp. 404–428. URL: https://doi.org/10.1007/11785477_24.

- Danny Dig, Kashif Manzoor etc (May 2008). "Effective Software Merging in the Presence of Object-Oriented Refactorings". In: *IEEE Transactions on Software Engineering* 34, 3, 321–335. URL: doi.org/10.1109/TSE.2008.29.
- Danny Dig, William G. Griswold etc (2014). "The Future of Refactoring(Dagstuhl Seminar 14211)". In: *Dagstuhl Reports* 4, 5, pp. 40–67. URL: <https://doi.org/10.4230/DagRep.4.5.40>.
- E. Balas, M. Padberg (1976). "Set Partitioning: A Survey". In: *SIAM Review* 18, pp. 710–760.
- Emerson Murphy-Hill Max S Danny Dig, William G. Griswold (2014). "The Future of Refactoring". In: *The Future of Refactoring, Dagstuhl Reports* 4, 5, 40–67.
- Everton L. G. Alves, Myoungkyu Song etc (2014). "RefDistiller: A Refactoring Aware Code Review Tool for Inspecting Manual Refactoring Edits". In: *In Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE '14)*. ACM, New York, NY, USA, 751–754. URL: <https://doi.org/10.1145/2635868.2661674>.
- F. Chierichetti, R. Kumar etc (2010). "Finding the jaccard median". In: *In 21st Symposium on Discrete Algorithms (SODA)*, pp. 293–311.
- F. Van Rysselberghe, S. Demeyer (2003). "Reconstruction of successful software evolution using clone detection". In: *In IWPSE '03: Proceedings of the 6th International Workshop on Principles of Software Evolution*. Washington, DC, USA: IEEE Computer Society, p. 126.
- Fabio Palomba, Andy Zaidman etc (2017). "An Exploratory Study on the Relationship Between Changes and Refactoring". In: *In Proceedings of the 25th International Conference on Program Comprehension (ICPC '17)*. IEEE Press, Piscataway, NJ, USA, 176–185. URL: doi.org/10.1109/ICPC.
- Feller, William (1968). "An Introduction to Probability Theory and Its Applications". In: *(Vol 1), 3rd Ed, Wiley, ISBN 0-471-25708-7*.
- Fowler, Martin (1999). "Refactoring: Improving the Design of Existing Code". In: *Addison-Wesley, Boston, MA, USA*.
- Friedman J. Olshen R. Breiman, L. Stone (Jan. 1984). "Classification and Regression Trees". In:
- G. Csurka, C. Dance etc (2004). "Visual categorization with bags of keypoints". In: *Proc. of ECCV International Workshop on Statistical Learning in Computer Vision*.
- G. Salton, M. J. McGill (1986). "Introduction to modern information retrieval". In: *McGraw-Hill*, pp. 293–311.
- G. Soares, R. Gheyi etc (Apr. 2013). "Comparing approaches to analyze refactoring activity on software repositories". In: *Journal of Systems and Software*, 86, 1006–1022.
- Gabriele Bavota, Andrea De Lucia etc (2015). "An Experimental Investigation on the Innate Relationship Between Quality and Refactoring". In: *Journal of Systems and Software*, 1–14. URL: doi.org/10.1016/j.jss.2015.05.024.
- Gabriele Bavota, Bernardino De Carluccio etc (2012). "When Does a Refactoring Induce Bugs? An Empirical Study". In: *In Proceedings of the IEEE 12th International Working Conference on Source Code Analysis and Manipulation (SCAM '12)*, 104–113. URL: doi.org/10.1109/SCAM.2012.20.

- Giuliano Antoniol, Massimiliano Di Penta etc (2004). "An Automatic Approach to identify Class Evolution Discontinuities". In: *In 7th International Workshop on Principles of Software Evolution*, 31–40. URL: <https://doi.org/10.1109/IWPSE.2004.1334766>.
- Gustavo Soares, Rohit Gheyi etc (Apr. 2013). "Comparing Approaches to Analyze Refactoring Activity on Software Repositories". In: *Journal of Systems and Software*, 1006–1022. URL: doi.org/10.1016/j.jss.2012.10.040.
- Hassan Ahmed E., Xie Tao (Nov. 2010). "Software intelligence: the future of mining software engineering data". In: *In Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research (FoSER '10)*, 161–166. URL: [doi:10.1145/1882362.1882397](https://doi.org/10.1145/1882362.1882397). S2CID3485526.
- Ittai Balaban, Frank Tip etc (2005). "Refactoring support for class library migration". In: *In Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 265–279. URL: doi.org/10.1145/1094811.1094832.
- J. Brant W. Opdyke M. Fowler, K. Beck etc (1999). "Refactoring: Improving the Design of Existing Code (Object Technology Series)". In: Jim Buckley, Tom Mens etc (Sept. 2005). "Towards a Taxonomy of Software Change". In: *Journal of Software Maintenance and Evolution: Research and Practice* 17, 5, pp. 309–332. URL: <https://doi.org/10.1002/smr.v17:5>.
- Johannes Henkel, Amer Diwan (2005). "CatchUp!: capturing and replaying refactoringsto support API evolution". In: *In 27th International Conference on Software Engineering*, 274–283. URL: <https://doi.org/10.1145/1062455.1062512>.
- Jurafsky Daniel, H. James Martin (2000). "Speech and language processing : an introduction to natural language processing, computational linguistics, and speech recognition". In: *Upper Saddle River, N.J.: Prentice Hall*.
- K. Prete, N. Rachatasumrit etc (2010a). "Catalogue of template refactoring rules". In: *The University of Texas at Austin, Tech. Rep. UTAUSTINECE- TR-041610, April*.
- (Apr. 2010b). "Catalogue of template refactoring rules". In: *The University of Texas at Austin, Tech. Rep. UTAUSTINECE- TR-041610*.
- Krizhevsky Alex, Sutskever Ilya etc (Apr. 2015). "ImageNet classification with deep convolutional neural networks". In: *Communications of the ACM*, 60 (6): 84–90. URL: [doi:10.1145/3065386](https://doi.org/10.1145/3065386). ISSN0001-0782. S2CID195908774.
- Kyle Prete, Napol Rachatasumrit etc (2010). "Template-based reconstruction of complex refactorings". In: *In Proceedings of the 26th IEEEInternational Conference on Software Maintenance (ICSM '10)*, 1–10. URL: <https://doi.org/10.1109/ICSM.2010.5609577>.
- M. Kim, D. Notkin (2009). "Discovering and representing systematic code changes". In: *IEEE Computer Society, in ICSE '09. Washington, DC, USA*, pp. 309–319.
- M. Kim, D. Notkin etc (2007). "Automatic inference of structural changes for matching across program versions". In: *In ICSE*, pp. 333–343.
- Marco Tulio Valente, Danilo Silva (2017). "Detecting Refactorings in Version Histories". In: *IEEE/ACM 14th International Conference on Mining Software*

- Repositories (MSR)*, 291–301. URL: <https://doi.org/10.1109/MSR.2017.14>.
- Martin, Robert C. (2009). “Clean Code: Refactoring, Patterns, Testing, and Clean Code Techniques”. In: *mitp, Frechen*.
- Michele Tufano Fabio Palomba, etc (2017). “There and back again: Can you compile that snapshot?” In: *Journal of Software: Evolution and Process*. URL: <https://doi.org/10.1002/smr.1838>.
- Miryung Kim, Dongxiang Cai etc (2011). “An Empirical Investigation into the Role of API-level Refactorings During Software Evolution”. In: *In Proceedings of the 33rd International Conference on Software Engineering (ICSE '11) ACM, New York, NY, USA*, 151–160. URL: doi.org/10.1145/1985793.1985815.
- Miryung Kim, Matthew Gee etc (2010). “Ref- Finder: A Refactoring Reconstruction Tool Based on Logic Query Templates”. In: *In Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE '10). ACM, New York, NY, USA*, 371–372. URL: <https://doi.org/10.1145/1882291.1882353>.
- N. Tsantalis, V. Guana etc (2013). “A multidimensional empirical study on refactoring activity”. In: *In Proceedings of the Conference of the Centre for Advanced Studies on Collaborative Research (CASCON)*, 132–146.
- Napol Rachatasumrit, Miryung Kim (2012). “. An empirical investigation into the impact of refactoring on regression testing”. In: *In Proceedings of the 28th IEEE International Conference on Software Maintenance (ICSM '12)*, 357–366. URL: doi.org/10.1109/ICSM.2012.6405293.
- Nassif Matthieu, Robillard Martin P (Nov. 2017). “Revisiting turnover-induced knowledge loss in software projects”. In: *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 261–272. URL: [doi: 10.1109/ICSME.2017.64](https://doi.org/10.1109/ICSME.2017.64). ISBN978-1-5386-0992-7. S2CID13147063.
- Nikolaos Tsantalis, Ameya Ketkar etc (2020). “Refactoring Miner 2.0”. In: *IEEE Transactions on Software Engineering*, p. 21. URL: <https://doi.org/10.1109/TSE.2020.3007722>.
- Nikolaos Tsantalis, Matin Mansouri etc (May 2018). “Accurate and Efficient Refactoring Detection in Commit History”. In: *ICSE '18, Gothenburg, Sweden*, pp. 725–743.
- Novais Renato, Santos José Amancio etc (2017). “Experimentally assessing the combination of multiple visualization strategies for software evolution analysis”. In: *Journal of Systems and Software*, 128: 56–71. URL: [doi: 10.1016/j.jss.2017.03.006](https://doi.org/10.1016/j.jss.2017.03.006).
- Peter Weissgerber, Stephan Diehl (2006). “Identifying Refactorings from Source-Code Changes”. In: *In Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering (ASE '06)*, pp. 231–240. URL: <https://doi.org/10.1109/ASE.2006.41>.
- Péter Hegedűs, István Kádár etc (Nov. 2017). “Empirical Evaluation of Software Maintainability Based on a Manually Validated Refactoring Dataset”. In: *Information and Software Technology*. URL: <https://doi.org/10.1016/j.infsof.2017.11.012>.

- Quinlan, J. Ross (1989). "Induction of decision trees". In: *Machine Learning 1*, 1, 81–106.
- (1993). "C4.5: Programs for Machine Learning." In:
- Rabin, Michael O. (1981). "Fingerprinting by random polynomials". In: *Technical Report, Harvard University*, pp. 15–81. URL: https://doi.org/10.1007/11785477_24.
- Ref-Finder. "Ref-Finder". In: (). URL: <https://sites.google.com/site/reffindertool/>.
- Refactoring, All Commits with. "Projects". In: (). URL: [ps://aserg-ufmg.github.io/why-we-refactor/#/allCommits](https://aserg-ufmg.github.io/why-we-refactor/#/allCommits).
- RefactoringCrawler. "RefactoringCrawler". In: (). URL: <http://dig.cs.illinois.edu/tools/RefactoringCrawler/>.
- RefactoringMiner. "Data". In: (). URL: <https://medium.com/@aserg.ufmg/what-are-the-most-common-refactorings-performed-by-github-developers-896b0db96d9d>.
- "RefactoringMiner". In: (). URL: <https://github.com/tsantalis/RefactoringMiner>.
- RefDiff. "RefDiff". In: (). URL: <https://github.com/aserg-ufmg/RefDiff>.
- S. Kusumoto T. Kamiya, K. Inoue (July 2002). "CCFinder: A multilinguistic tokenbased code clone detection system for large scale source code". In: *IEEE Transactions on Software Engineering*, 657–670.
- Serge Demeyer, Stéphane Ducasse etc (2000). "Finding Refactorings via Change Metrics". In: *In Proceedings of the 15th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA '00)*. ACM, New York, NY, USA, 166—177. URL: <https://doi.org/10.1145/353171.353183>.
- Stas Negara, Nicholas Chen etc (2013). "A Comparative Study of Manual and Automated Refactorings". In: *In Proceedings of the 27th European Conference on Object-Oriented Programming (ECOOP'13)*. Springer-Verlag, Berlin, Heidelberg, pp. 552–576. URL: https://doi.org/10.1007/978-3-642-39038-8_23.
- Stephen R. Foster, William G. Griswold (2012). "WitchDoctor: IDE support for real-time auto-completion of refactorings". In: *In Proceedings of the 34th International Conference on Software Engineering (ICSE '12)*, 222–232. URL: doi.org/10.1109/ICSE.2012.6227191.
- Steven Davies, Marc Roper etc. "Catalog of Refactorings". In: (). URL: <https://refactoring.com/catalog/>.
- (2014). "Comparing text-based and dependence-based approaches for determining the origins of bugs". In: *Journal of Software: Evolution and Process* 26, 1, pp. 107–139. URL: <https://doi.org/10.1002/smr.1619>.
- T. H. Cormen, C. E. etc (2001). "Introduction to Algorithms". In: MIT Press.
- T. Kamiya, S. Kusumoto etc (2002). "CCFinder: A multilinguistic token-based code clone detection system for large scale source code". In: *IEEE Transactions on Software Engineering* 28, 654—670.
- Tan Liang, Christoph Bockisch (Oct. 2022a). "Checking Refactoring Detection Results Using Code Changes Encoding for Improved Accuracy". In: *22nd IEEE International Working Conference on Source Code Analysis and Manipulation*.

- Tan Liang, Christoph Bockisch (Dec. 2022b). “Diff Feature Matching Network in Refactoring Detection”. In: *29th Asia-Pacific Software Engineering Conference*.
- (2023). “RefDiff-Model: Model-based Optimization Solution for Refactoring Detection”. In:
- Tomas, Mikolov (2013). “Efficient Estimation of Word Representations in Vector Space”. In: URL: [arXiv:1301.3781](https://arxiv.org/abs/1301.3781).
- Uri Alon, Meital Zilberstein etc (2019). “code2vec: Learning Distributed Representations of Code”. In: *POPL*. URL: [arXiv:1803.09473](https://arxiv.org/abs/1803.09473).
- Van Gurp Jilles, Bosch Jan (Mar. 2002). “Design erosion: problems and causes”. In: *Journal of Systems and Software*, 61(2): 105–119. URL: [doi : 10 . 1016 / S0164-1212\(01\)00152-2](https://doi.org/10.1016/S0164-1212(01)00152-2).
- Volder, K. D. (1998). “Type Oriented Logic Meta Programming”. In: *Ph.D. dissertation, The University of British Columbia*.
- Xi Ge, Quinton L. DuBose etc (2012). “Reconciling Manual and Automatic Refactoring”. In: *In Proceedings of the 34th International Conference on Software Engineering (ICSE '12)*, 211–221. URL: <http://dl.acm.org/citation.cfm?id=2337223.2337249>.
- Xi Ge, Saurabh Sarkar etc (2014). “Towards Refactoring-awareCode Review”. In: *In Proceedings of the 7th International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE '14)*. ACM, New York, NY, USA, 99–102. URL: <https://doi.org/10.1145/2593702.2593706>.
- (2017). “Refactoring-Aware Code Review”. In: *In Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC '17)*, 71–79. URL: doi.org/10.1109/VLHCC.2017.8103453.
- Xufeng Han, Thomas Leung etc (2015). “MatchNet: Unifying feature and metric learning for patch-based matching”. In: *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. URL: [DOI : 10 . 1109 / CVPR . 2015.7298948](https://doi.org/10.1109/CVPR.2015.7298948).
- Zhangyin Feng, Daya Guo etc (2020). “CodeBERT: A Pre-Trained Model for Programming and Natural Languages”. In: *EMNLP*. URL: [arXiv : 2002 . 08155](https://arxiv.org/abs/2002.08155).
- Zhenchang Xing, Eleni Stroulia (2005). “UMLDiff: An Algorithm for Object-oriented Design Differencing”. In: *In Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering (ASE '05)*, ACM, New York, NY, USA, pp. 54–65. URL: <https://doi.org/10.1145/1101908.1101919>.
- (2007). “API-Evolution Support with Diff- CatchUp”. In: *IEEE Transactions on Software Engineering* 33, 12, 818–836. URL: doi.org/10.1109/TSE.2007.70747.
- Zisserman, J. Sivic & A. (2003). “Video Google: A Text Retrieval Approach to Object Matching in Videos”. In: *Proc. of ICCV*.