# Untangling Java Code Changes

**Xiaowei Chen**

**A Thesis**

**in**

**The Department**

**of**

**Computer Science and Software Engineering**

**Presented in Partial Fulfillment of the Requirements**

**for the Degree of**

**Master of Applied Science (Software Engineering) at**

**Concordia University**

**Montréal, Québec, Canada**

**July 2021**

**© Xiaowei Chen, 2021**

CONCORDIA UNIVERSITY
School of Graduate Studies

This is to certify that the thesis prepared

By: **Xiaowei Chen**

Entitled: **Untangling Java Code Changes**

and submitted in partial fulfillment of the requirements for the degree of

**Master of Applied Science (Software Engineering)**

complies with the regulations of the University and meets the accepted standards with respect to originality and quality.

Signed by the Final Examining Committee:

________________________ Chair
*Dr. Jinqiu Yang*

________________________ Examiner
*Dr. Weiyi Shang*

________________________ Examiner
*Dr. Jinqiu Yang*

________________________ Supervisor
*Dr. Emad Shihab*

Approved by ________________________
Lata Narayanan, Chair
Department of Computer Science and Software Engineering

____________ 2021 ________________________
Mourad Debbabi, Dean
Faculty of Engineering and Computer Science

# Abstract

Untangling Java Code Changes

Xiaowei Chen

Pull requests are a critical approach for developers to collaborate in software development, which also initiates the following code review and integration. However, tangled pull requests can be introduced into version control systems via committing unrelated or multi-purposes code changes in one single pull request, which have been found to bring a negative impact on code recommendation systems and bug prediction models in the previous research. In this thesis, we conduct a case study on 640 pull requests among 8 popular open-source Java projects from GitHub. Through manual analysis, we find that 47% of the pull requests are tangled. In order to further understand the characteristics of tangled pull requests, we perform a qualitative annotation and classify the reasons for tangled pull requests. We find that 75% are tangled because bug fixing, feature improvement, or new feature adding are committed with an update in test code. The remaining 25% of pull requests are tangled because developers commit two or more unrelated bug fixing, feature improvement, test code adding or modifying, new feature adding, feature improvement and bug fixing, and other combinations inside one pull request. We also propose an approach to predict whether a pull request is tangled or not with an AUC of 0.87. Furthermore, we also predict whether two lines of code changes belong to the same task, which achieves an AUC of 0.74.

# Dedication

*To my grandparents, parents, husband and best friend Rong.*

# Acknowledgments

I would like to thank my supervisor Dr. Emad Shihab for giving me the opportunity to be part of Data-driven Analysis of Software (DAS) Lab, as well as his endless support and patience, without whom nothing of this would be possible. Thank you for always encouraging me, and giving me all kinds of help to achieve my academic success. You are not only the supervisor during my master's study, but also the mentor to help me achieve a better version of myself. Your passion, immense knowledge and plentiful experience have inspired me all the time.

I extend my gratitude to the professors with whom I worked closest for my degree and research, Dr. Bram Adams, Dr. Nikolaos Tsantalis, Dr. Frédéric Godin and Dr. Xin Xia. It was a delight to follow your teachings. Thank you for all the valuable discussions and guidance, which expand my knowledge. In the same manner, I would like to thank my thesis examiners, Dr. Weiyi Shang and Dr. Jinqiu Yang for taking time to review my thesis and giving valuable feedback on it.

I would also like to acknowledge the unaccountable support I received from Dr. Rabe Abdalkareem, Dr. Mohamed Aymen Saied and Dr. Diego Costa. Thank you for always listening to me and always being willing to provide excellent feedback and guidance. Thank you Dr. Marouane for constructing the dataset with me.

To my lab colleagues and friends, Suhaib, Giancarlo, Mohamed, Ahmad, Hosein, Mahmood, Mouafak, Juan, Abbas, Mahsa, Patrick, Khaled, Nicholas, Olivier, Jinfu, Kundi, Zhenhao, Zishuo, Max, Sophia, Sara, Gui, Mehran, Triet, Bo, Sadegh, Isabella, Javier and everyone else in our peer research labs. Thank you for working alongside me during this journey. I wish you all the best in your prosperous future.

Last but not least, I would like to thank my husband for his unconditional love and

support. You are truly a sincere and iridescent person. I am happy to have you in my life.

# Related Publicaiton

The following publications are not directly related to the material in this thesis, but were produced in parallel to the research performed for this thesis.

- **Xiaowei Chen**, Rabe Abdalkareem, Suhaib Mujahid, Emad Shihab, Xin Xia . " Helping or not helping? Why and how trivial packages impact the npm ecosystem". Empirical Software Engineering (2020)

# Contents

**List of Figures** x

**List of Tables** xi

**1 Introduction** 1

1.1 Introduction to the Research Domain . . . 1

1.2 Contributions . . . 5

1.3 Outline . . . 6

**2 Related Work** 7

**3 Untangling Java Code changes** 12

3.1 Case Study Setup . . . 12

3.1.1 Pull request based software development . . . 12

3.1.2 Atomic and Tangled Pull Requests . . . 13

3.1.3 Data Collection . . . 13

3.1.4 Selecting the Studied Systems . . . 14

3.1.5 Manually Untangling Code Changes . . . 17

3.2 RQ-1: How prevalent are tangled pull requests? . . . 19

3.2.1 Motivation . . . 19

3.2.2 Approach . . . 19

3.2.3 Results . . . 19

3.3 RQ2: Can we effectively predict whether or not a pull request is tangled? . . . 22

3.3.1 Motivation . . . 22

3.3.2 Approach . . . 22
3.3.3 Results . . . 25

**4 Discussion** **28**
4.1 Can we effectively predict the tasks of tangled pull requests? . . . 28

**5 Threats to Validity** **34**
5.1 Internal Validity . . . 34
5.2 External Validity . . . 34

**6 Conclusions and Future Work** **36**
6.1 Conclusions . . . 36
6.2 Future Work . . . 37

**Bibliography** **39**

# List of Figures

Figure 3.1 **Commits for an atomic pull request example.** . . . . . . . . . . . 13
Figure 3.2 **Commits in a tangled pull request example.** . . . . . . . . . . . 14
Figure 3.3 **Distribution of the amout of commits inside pull requests.** . 15
Figure 3.4 **Important features in predicting whether a pull request is tangled or not when using Undersampling and Random Forest.** . . 26
Figure 4.1 **Important features in predicting whether two lines of code changes are in the same task when using Undersampling and Random Forest.** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 32

# List of Tables

Table 3.1 **Overview of the Studied Systems.** . . . . . . . . . . . . . . . . . . . 16

Table 3.2 **Number of closed and filtered pull-requests per project.** . . . . 16

Table 3.3 **Task Definition.** . . . . . . . . . . . . . . . . . . . . . . . . . . . 18

Table 3.4 Most frequent tangled tasks observed on the 301 tangled pull requests. 20

Table 3.5 **Selected features for the prediction of tangled and atomic pull requests.** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 23

Table 3.6 **Confusion matrix.** . . . . . . . . . . . . . . . . . . . . . . . . . . 24

Table 3.7 **Performance of Machine Learning algorithms under different re-sampling techniques for predicting whether a pull request is tangled or not.** . . . . . . . . . . . . . . . . . . . . . . . . . . 25

Table 3.8 **Odds ratio of features for the prediction of tangled and atomic pull requests using undersmapling and Logistic Regression.** . . . . . 26

Table 4.1 **Overview of the pair-wise code metrics.** . . . . . . . . . . . . . . . 29

Table 4.2 **Performance of Machine Learning algorithms under different re-sampling techniques for predicting whether line and line should be together.** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 32

Table 4.3 **Odds ratio of features for the prediction of whether two lines of code changes should be together using Logistic Regression and Undersampling.** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 33

# Chapter 1

# Introduction

## 1.1 Introduction to the Research Domain

Code review is an important mechanism in software development, which helps improve code quality and avoid introducing bugs. Tao et al. [1] pointed out in an exploratory study in industry that understanding code changes is a practice so fundamental that it happens frequently in software development. Previous research found that reviewers are more likely to give feedback to the code they understand, and small changes are easier for reviewers to understand [2].

However, developers often address multiple issues (e.g., bug fixes, new feature addition, code refactoring) within a single composite commit, which is called tangled commits. Herzig and Zeller [3] confirmed that tangled commits are not a theoretical concept, they found that up to 15% of bug fixing commits on subject projects are bundled with other tasks. Tangled commit would be easier to understand if they are decomposed to their individual development issues. Hence, practitioners need tools that can help identify and untangle tangled commits [1].

Aside from the problem that tangled commits cause to the understanding of code, code recommendation systems rely on the assumption that commits contain unique tasks. Recommendation systems are applied to many different software engineering tasks. From identifying defects in the code [4][5], finding bugs that are likely to be re-opened [6][7][8] to assigning maintenance tasks to developers based on their expected experience [9][10].

Commits that contain multiple changes related multiple tasks can deeply affect the accuracy of such recommendation systems by introducing noise in the data.

To address the problems caused by tangled commits, researchers have proposed various tools to untangle or decompose code changes in a single commit [1][3][2][11][12]. Barnett at al. [13] considered lightweight code dependency (e.g, definition and uses information) to cluster code changes. Wang et al. [12] try to mine more complex and comprehensive relationships (e.g, code similarity) between programs. However, the clusters yielded by the tools (CLUSTERCHANGES and CoRA) contained many trivial groups. Dias et al. [11] record all the code change modifications of developers, then group these modifications based on their algorithm. Similarly, Hayashi et al. [14] develop a tool (ChangeMacroRecorder) to extract the edit history, then group edits based on time and syntactical information. However, their tool yields too fine grained edits groups and it is likely to introduce noisy modifications or edits into version control systems, which could require more effort for developers in the code review process. Herzig and Zeller [3] propose a voting mechanism to untangle code changes via group code changes with the highest confidence value, which is voted by data dependency, call graph, file association in the commit history, number of lines between code changes, and package segment difference. While the features they introduced are of a more simple nature, the yielded tangled commits are composed of atomic commits. Arima et al. [15] confirmed that code changes contribute to the same task (e.g, bug fixing, feature addition) can be inappropriately partitioned in Java projects, thus the combination of atomic commits are not always tangled commits. Most of the previous work has their limitations, our research tries to make further progress on untangling Java code changes.

Our goal is to help developers avoid introducing tangled changes or pull requests (PR) into the version control systems, to mitigate the problems caused by the complexity of tangled changes. We focus on PRs because they are the de-facto standard for collaborative development in open source projects. Our work focuses in proposing tools that help identify and untangle tangled PRs, so developers can work on atomic PRs that are easier to understand and review. Furthermore, atomic PRs can also be better employed in recommendation systems that work at the pull request level. Compared to previous work, our work has three main differences: 1) We are the first to study the entanglement of code

changes at the PR level. The PR description provides the context for reviewers to understand the changes without digging into change details [16] 2) We published the large dataset of manually curated and classified set of 640 PRs from eight of the most popular open-source Java projects from GitHub. Our dataset includes the classification of PRs in tangled and atomic (not tangled), as well the type of tasks tangled in the PRs. 3) We introduce a rich set of features to use machine learning classification models to identify tangled PRs. The set includes features extracted from program slicing, refactoring detection tools [17][18][19], Abstract Syntax Tree(AST), program dependency, diff of lines, and other features related to static analysis.

To accomplish our goal, we face three main challenges:

- To the best of our knowledge, there is no publicly available dataset of tangled pull requests. Hence we need to manually collect and curate a large dataset of atomic and tangled pull requests. This data is of paramount importance to train classification models that can help developers at identifying tangled PRs.

- There were no previous studies working on predicting tangling pull requests. Hence, we have to define a comprehensive set of pull request features (e.g., number of files, functions, hunks, tree node operations are touched inside a PR, etc.) that are able to capture the characteristics of tangled PRs.

- After predicting tangled pull requests, we aim to provide a model that can cluster the code that belongs to the same task. Hence, we need to explore what kinds of code-level features should be introduced into our model to complement pull request features (e.g., data dependency, call graph, file association, number of lines between code changes).

In this thesis, we conduct experiments on eight popular open-source Java projects from GitHub to evaluate the performance our algorithms. We focus on studying the prevalence and characteristics of multi-commit tangled PRs, i.e., PRs that contain more than one commit. We found that (1) our algorithm is effective at predicting whether a PR is tangled or not with an AUC of 0.87. (2) our algorithm can predict whether two lines of code changes are in the same task with an AUC of 0.74. Moreover, we found that (3) the most popular combinations

of tasks that are committed in one single PR are fixing bugs for different features, improving different existing features, and adding or modifying test codes for different features. In particular, we answer the following research questions:

**RQ1** *How prevalent are tangled pull requests?*

We manually investigate 640 multi-commit pull requests from 8 popular Java repositories in GitHub and find that 47% (301) of the PRs are tangled, 53% of them are atomic (not tangled). When inspecting the type of tasks included in a tangled PRs, we find that 35.1% of PRs are tangled because developers change the test suite when fixing a bug, updating a feature. This is considered good practice of software development, as developers ensure that the test suite remains up-to-date with changes in the codebase. For example, fixing a bug and adding or modifying test code, adding a new feature or improving an existing feature (e.g, performance improvement[20][21]) and adding or modifying test code and so on. On the other hand, 11.9% of the PRs are tangled because they include two unrelated tasks which can hinder the understanding of the change (bad practice). For example, fixing two unrelated bugs or improving two different existing features, are examples of tangled pull requests caused by discouraging development practices.

**RQ2** *Can we effectively predict whether or not a pull request is tangled?*

Based on the manual investigation in RQ1, we construct a dataset to train models that can help identify tangled pull requests. As we are interested in untangling PRs that should not be tangled, we only consider the tangled pull requests that were caused by discouraging development practices (bad practices). Our resulting dataset is unbalanced, as there are more atomic pull requests than tangled ones. Hence, we employ data balancing strategies using Oversampling and Undersampling techniques in the training data.

We found that the best model trained by Random Forest algorithm while undersampling the training dataset, achieving the AUC of 0.87. The most important features in our models are the number of tree node operations, functions and the

number of characters of the description body when contributors proposed a PR. The number of files, the number of characters of the description title, the number of hunks and number of days it took to close a PR also have an impact on models' performance.

To measure the impact of the changes in the feature values in the predicted class, we calculate the odds ratio of each feature. We find that the number of tree node operations, functions, whether a PR contains both source file and test file, the number of characters inside a PR description title, the number of days it took to close a PR have an odds ratio greater than one.

**Discussion** *Can we effectively predict the tasks of tangled pull requests?*

Instead of predicting all the code changes related to a specific task, we predict whether two lines of code changes are in the same task. We considered code change pairs that are labeled to the same task should be together. Based on the ground truth of Chapter 3.2, related code change pairs are much less frequent than unrelated code pairs (pairs that do not belong to the same task). Thus, we employ oversampling and undersampling techniques. We found that the Logistic Regression model to achieve the best performance, with AUC of 0.74, regardless of the sampling technique.

We use the odds ratio to measure the impact of metric values on the predicted class. We found that if the two lines of code changes belong to the same file, the likelihood of them belong to the same task increases by 1540%, compared to those that are not. Followed by the functions of these two code changes belong to have a call or being called relationship, and the number of commonly related functions.

## 1.2 Contributions

The contributions of this thesis are as follows:

- We publish the first curated dataset on tangled pull requests. We manually classified 640 PRs as tangled or atomic PR, and label every line of code change with its related task. The high-quality data set will be shared publicly [22].

- We propose a novel method to predict whether a PR is tangled or not, which involves features from different dimensions.

- We propose a new method to predict whether two lines of changes belong to the same task, which leverage features from static program analysis(e.g, program slicing, AST node operations, refactoring method detection, code dependency, the number of common variables, methods and classes, etc).

## 1.3 Outline

The remainder of the thesis is organized as follows. Chapter 2 presents the related work on untangling/decomposing code changes that has been published in recent years. Chapter 3 describes our case study setup and presents the results of our untangling Java code changes algorithms. Chapter 4 discusses the implications of our results. Chapter 5 discusses threats to the validity of our findings. Finally, Chapter 6 draws conclusions.

# Chapter 2

# Related Work

Herzig and Zeller. [3] proposed the first algorithm to untangle Java code changes. They introduce five voters, including data dependency, call graph, file association in the commit history, number of lines between two changes, and package segments difference. The aggregated score and the number of classifications will decide whether code changes belong to the same task. When collecting their dataset, they combined atomic commits to make tangled ones, thus the correct classifications of code changes are these atomic ones. Their algorithm achieved a precision rate of 77%, while recall rate is not mentioned.

To confirm the existence of tangled commits and their negative impact on bug prediction models, they manually classified commit messages related to at least one solved issue report of five open-source Java projects and found that up to 15% of them are tangled. They used their algorithm to untangle commits marked as tangled in the manual work and found that on average 16.6% of all the source files are incorrectly related to bug fixing issues, which revealed tangled commits have a bad impact on bug prediction models.

Tangled commits are frequent and had a negative impact on bug prediction models. They suggest that version archive miners should notice this and take advantage of similar untangling algorithms to classify code changes, thus alleviate the impact of tangled changes on mining models.

Dias et al. [11] developed the first tool to untangle fine-grained code changes written in Smalltalk and published a dataset of untangling commits. Tangled commits not only make code review, reversion, and integration harder, but also decrease the reliability of history

analysis results. Their approach is to use a machine learning model to calculate the likelihood that two lines of code changes should be committed together, with a threshold of 0.25. Using the code classifications from two developers during daily development for four months, their model achieved an AUC value of 0.98. The best algorithm observed was Random Forest and the most significant metrics were time difference, number of modifications between two changes, and whether they modified the same file. Six developers were asked to evaluate their tool, on average 73.5% of all the code changes were successfully classified. Developers thought fine-grained can introduced noisy intermediate modifications, which makes it hard to check the code. They developed a tool called EpiceaUntangler to untangle fine-grained code changes in Smalltalk, which are evaluated by six new developers with a median success rate of 91%.

Arima et al. [15] first investigated how much and what kind of code changes are inappropriately partitioned in Java projects, and propose an algorithm to detect inappropriately partitioned commits (IP commits). Previous research only considered untangling code changes in a single commit, while disregarding if code changes that contributed to the same task should be committed together in a single commit. IP commits can degrade the performance of repository analysis and understandability of commits. They used the Dijkstra shortest path algorithm to calculate the likelihood of whether two commits belong to the same task, nodes are functions inside two commits, weighted directed paths are whether they are the same method in a different commit, whether one call or be called by the other, and whether two methods in the same class. First, they manually checked 1,174 commit pairs from two Java projects and found 81 to be IP commits. The relationships of IP commit pairs can be classified into 3 types, including correctness, dependency, and cooperation. They trained their model with this manually curated dataset, achieving an F-measures between 71.4% and 74.5%. To evaluate their algorithm, they used it to detect IP commit pairs from 18,619 commit pairs, and then manually confirmed their precision. Using this data set, their algorithm achieved a precision rate of 82.2% and 88.4% separately, while the recall rate is unknown. Their research confirmed IP commits, which should be committed together but scattered in multiple commits do exist, and their algorithm can detect them with an F-measure of 71.4% and 74.5% separately.

Muylaert et al. [23] used program slicing technique to detect and untangle tangled changes. They hypothesized that code changes which belong to the same slicing should be together, thus a commit can be decomposed using program slicing. They first distilled fine-grained code changes in a commit to Abstract Syntax Tree (AST), then create System Dependence Graph (SDG) for every touched file. Code changes that belong to the same slice inside one SDG will be grouped. While Tool TINYPDG can only get Procedure Dependence Graph (PDG) inside a method, they extend it by adding procedure dependence graph of different methods, thus SDG can be generated. When evaluating the tool, they first filtered the data set published by Herzig et al. [3], cause their technique can not deal with or some commits related to two issue reference while only one issue, then 388 commits remained, including 194 single-task commits and 194 multi-task commits. Their technique achieved an F-score around 0.7 for single-task detection and 0.64 for multi-task detection. Generally, the account of clusters after untangled via their method is more than it should be. During the evaluation, some computer scientists found this can help code review. Their technique is able to detect single-task commits and multi-task commits, using the data set after being strictly filtered. Some computer scientists considered their clusters of code changes can help code review, even though the number of clusters is more than it should be.

Sothornprapakorn et al. [24] proposed a tree visualization tool, that can help developers recognize various kinds of tasks before the commit. The tool helps practitioners to submit code changes related to the selected sub-tree at one time. Previous research showed that even though commit one-task code changes is a good habit, tangled commits make it hard for developers to recognize related tasks. They split code changes into refactoring and non-refactoring parts, code changes related to the same refactoring method detected by the tool called RefactoringMiner will group, non-refactoring code changes have a close distance in inter-procedure dependency graph will gather. In order to evaluate their tool, 8 industry developers were asked to untangle code changes with and without the help of the tool, it turned out that the tool can help to understand the purposes of tasks, commit selected code easily and considered to be useful by developers. Their visualization tool can help developers manually untangle code changes.

Hayashi et al. [14] developed a history refactoring tool called HISTOREF, which can

group code changes together, selected code groups can be committed at one time. Single task commit policy is not always followed by developers, this can make patches to projects hard to be reviewed. Meanwhile, manually manage edit histories is cumbersome and error-prone, hence, a tool that can help automatically refactor these edit histories during the editing process is necessary. Their tool untangles code changes by classifying the edit history of the project. First, they get the edit history collected by OperationRecorder or Fluorite, then based on commit time and syntactical information, their algorithm groups these edits. Developers can select several of these groups and commit them. Their tool can help refactor the edict history, but lack evaluation.

Maruyama et al. [25] published a tool called ChangeMacroRecorder (CMR) that can record fine-grained textual changes of a project during the editing process. Textual changes can help to understand the evolution of a project, and it can help developers on later development phases. First, they investigate the reasons for defects of the existing tools. Tools based on built-in document listeners of Eclipse can not catch the edit operations on closed files, the ones that rely on the undo history of Eclipse can not record the edits distributed in different files. CMR guarantees to record these edits by monitoring the local history of the files. The previous tool can not record the modifications that are not made inside Eclipse, so that their records are not consistent, while CMR can detect this irregular discrepancy and record the textual changes. They classify all kinds of edits into change macros, and append these change macros into the textual changes generated, which makes the records easily understandable by humans. Their tool overcame the defects of the previous textual change recorders, and define change macros to make these changes shown in a human-readable way. CMR can provide a more accurate edits history for extend tools than other edit recorders.

Tsantalis et al. [26] compared RefactoringMiner with one of the best tools for detecting refactoring methods, and found that RefactoringMiner achieved a precision rate of 98% and a recall rate of 87%, 10% better than previous works. Refactoring detection algorithms are basic for various applications, thus their accuracy is crucial. But previous tools have exhibited some limitations like requiring similarity thresholds from users, commits must be build successfully, all files of two versions in a project must be provided. While

RefactoringMiner does not have these limitations, it only require git url of a project and two commit id, then output the refactoring methods and their related code information.

Tsantalis et al. [26] conducted the detection on a third-party data set which contains 538 commits in 185 open-source GitHub-hosted projects which has been evaluated by the previous authors. The authors re-validated the data set to guarantee the correctness. They not only compared the precision and accuracy, they also compared the Execution time. They found that RefactoringMiner performed better than related algorithms in a considerably shorter execution time. Their refactoring detecting algorithm is the first one does not require users to provide code similarity thresholds, which achieved a precision rate of 98% and a recall rate of 87%.

Compared with the previous work, we are the first to utilize PR to untangle code changes. PR description on GitHub provide more context for reviewers to understand the changes without digging into change details [16]. Our data set is much richer, which contains 640 PRs from eight of the most popular open - Java projects from GitHub. We introduce a richer set of features, which includes program slicing, refactoring detection [17] [18] [19], Abstract Syntax Tree (AST), program dependency, diff of lines, and other features related to static analysis. Meanwhile, we perform manual work to investigate the tangled reasons (e.g, bug fixing for different features, different features improvement are committed in one single PR).

# Chapter 3

# Untangling Java Code changes

## 3.1 Case Study Setup

In this chapter, we present the background of untangling Java code changes and our case study setup, including the subject selection process, and the methodology of data extraction.

### 3.1.1 Pull request based software development

Open-source software development is highly collaborative. Developers contribute to existing projects to push the development of their favorite software, to include a feature, fix bugs and report issues in the project, and many other reasons. The de-facto method for collaborating on open-source projects is the pull request based software development [27]. In a pull request development paradigm, contributors create a copy of the project they wish to contribute (fork) and submit their code changes in a form of a pull request [27], containing descriptive information of what the code is supposed to accomplish and related artiifacts. Maintainers will review the code and may ask contributors to modify the code to meet the standard of the projects before they merge the pull requests. Once the pull request meets the project requirements, it can be merged by one of the maintainers effectively deploying the contribution into the project.

### 3.1.2 Atomic and Tangled Pull Requests

*Atomic pull request* is a pull request that contains only one task, while a *tangled pull request* contains multiple tasks. For example, a pull request [1] in ElasticSearch contains 3 commits, the purpose of it is to add a null check for CopyExecutionStateStep. As shown in Figure 3.1, the first commit already finished the task, the second one is to modify the location of a variable added in the first commit. Thus, this pull request is an atomic pull request.

A tangled pull request is a pull request that contains more than one task. As shown in Figure 3.2, it is a tangled pull request which is analyzed by our researchers, it contains 4 commits and 2 tasks. The first task is to add a new feature, it is to add support for GID attribute, the second task is to fix a bug. The example shown in Figure 3.2.

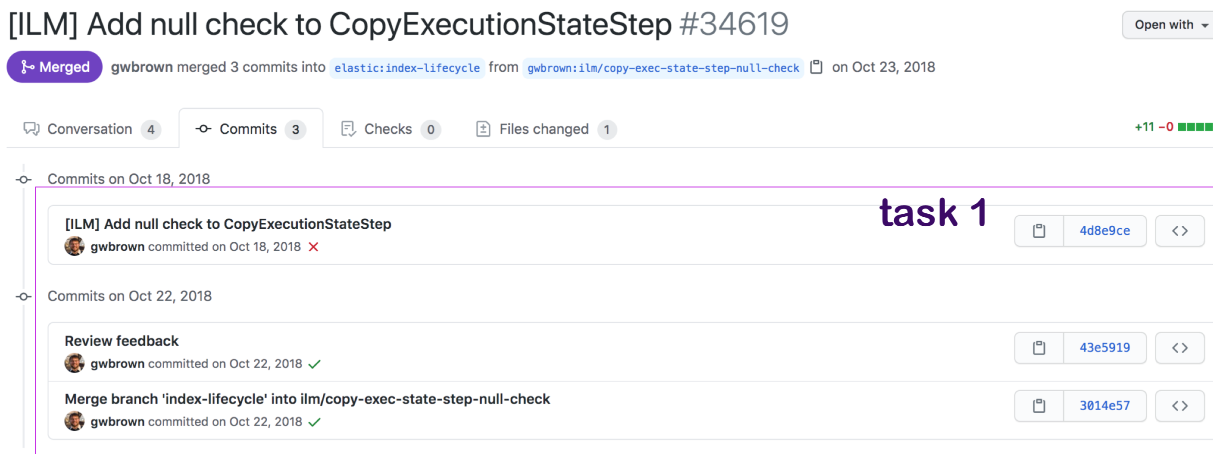


**Figure 3.1: Commits for an atomic pull request example.**

### 3.1.3 Data Collection

In this section, we present the methodology used for collecting the dataset of pull requests from popular software projects (Section 3.1.4) and the process used to manually find and untangle pull-requests (Section 3.1.5).

[1] https://github.com/elastic/elasticsearch/pull/34619

Figure 3.2: **Commits in a tangled pull request example.**

### 3.1.4 Selecting the Studied Systems

We want to investigate the occurrence of tangled pull requests on highly collaborative and popular software projects. To that aim, we focus on mining projects hosted on GitHub [28], which is a major software repository platform in the current open-source landscape, with more than 96 million repositories, providing a highly collaborative environment encompassing more than 200 million pull requests to date [29]. We start by selecting the top 50 most-starred projects from GitHub at the time of collecting the data (October 27th, 2018). Stars in GitHub are one of the metrics for a project's popularity [30], hence, highly-starred software projects tend to be popular and are more likely to have higher-quality code and a process for code review.

As the scope of our research is to investigate the occurrence of tangled pull requests, we collect pull requests from the 50 most popular projects in GitHub. In GitHub, pull requests have two statuses: *open*, indicating the pull requests still under work and *closed*, once developers have concluded working on the pull request. We focus on collecting only closed pull requests because they provide us the full context of the collaboration between contributors and maintainers. In addition, we consider in our study the pull requests merged into the master/main branch of the project, as they may be deployed in a released version. To collect the pull requests data, we resort to the official GitHub API (REST) [31].

The distribution of the number of commits inside a pull request is showed in Figure 3.3, we found that most of the pull request contains no more than 5 commits.

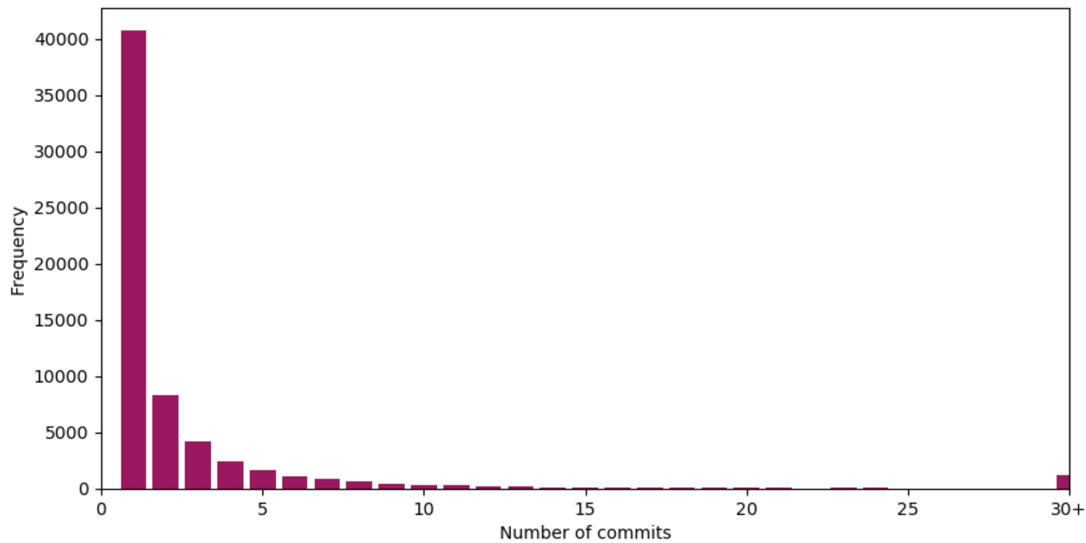


**Figure 3.3: Distribution of the amout of commits inside pull requests.**

We focus our study on multi-commit pull requests, i.e., pull requests that contain two or more commits. In order to explore aspects that lead to tangled PRs and train the models of prediction, we need a high-quality data set. We considered that multi-commit PRs are more likely to be tangled. Hence, we employ a filtration process to select the target pull requests for our study. First, we only include pull requests containing more than one commit. Second, we only consider pull requests that have changed Java files. Third, since we rely on manual analysis to initially infer if a multi-commit pull request is tangled, we have to keep our manual analysis under a reasonable workload. Hence, we excluded very large pull requests from our analysis: pull requests with more than five commits and that changed more than 1,000 lines of Java source code.

We consider eight popular Java projects, and randomly select 80 pull requests from these repositories. As shown in Table 3.1, the selected projects are Dubbo, Jenkins, Libgdx, Netty, Spring Boot, Elasticsearch, RxJava and RealmJava. These 8 projects cover several software domains, from a search engine (ElasticSearch) and database (RealmJava) to game-development framework (Libgdx) and asynchronous API (RxJava). All projects have a

long history of development (more than 7 years), and are still under active development. Furthermore, the selected projects are very popular (above 10k stars), highly collaborative (above 2k pull-requests), and contain a very large code base (avg of 360k LOC).

**Table 3.1: Overview of the Studied Systems.**

| Project | Description | Age | #Stars | KLOC | #Commits |
|---|---|---|---|---|---|
| Dubbo | RPC framework | 8 | 29.9k | 150 | 3,962 |
| Jenkins | Automation server | 13 | 14.3k | 160 | 28,932 |
| Libgdx | Game-development framework | 10 | 16.1k | 280 | 13,882 |
| Netty | NIO client server framework | 12 | 21.4k | 279 | 9,600 |
| SpringBoot | Microservice framework | 7 | 43.1k | 274 | 23,885 |
| Elasticsearch | Search engine | 10 | 45.2k | 1,507 | 49,098 |
| RxJava | Asynchronous programming API | 8 | 40.9k | 283 | 5,593 |
| RealmJava | Mobile database | 8 | 10.7k | 89 | 8,245 |

**Table 3.2: Number of closed and filtered pull-requests per project.**

| Project | # Closed PRs | # Filtered PRs |
|---|---|---|
| Dubbo | 2,160 | 190 |
| Jenkins | 4,235 | 508 |
| Libgdx | 2,708 | 320 |
| Netty | 4,842 | 102 |
| SpringBoot | 3,761 | 101 |
| Elasticsearch | 26,850 | 2,030 |
| RxJava | 3,270 | 345 |
| RealmJava | 2,684 | 159 |

### 3.1.5 Manually Untangling Code Changes

The first goal of our study is to investigate the prevalence of tangled pull requests on the selected eight Java projects. To accomplish that, we first have to manually investigate the occurrence of tangled pull requests on the eight selected projects, and attribute the respective code changes to each task. We randomly select 80 pull requests from each of the 8 selected projects to conduct our manual analysis, encompassing a total of 640 pull requests. Then we manually analyze if a pull request contains multiple tasks using the GitHub web interface of pull requests. The pull request web page contains all the information needed for our analysis, such as the related commits, the development timeline, the suggestions from reviewers after code review, assigned labels, and a conversation section with comments provided by contributors.

To identify how many tasks a pull request contains we perform a thorough manual investigation. Two researchers analyzed the GitHub pull requests web pages independently, including the pull request description, the comments included by contributors and maintainers, and inspect the commit messages and their associated code changes. Given that code refactoring is frequently performed, we make use of the browser plugin Refactoring Aware Commit Review [32] to facilitate the visualization of the changed code. This plugin, based on the tool RefactoringMiner [26], provides a way of visualizing code refactoring in the GitHub commit web page, grouping code changes associated with the same refactoring operation (e.g., move method).

It is well known that commits are not always partitioned [15] at the task level. Developers are likely to push multiple commits to accomplish a single task, improving on previous commits, addressing issues found by maintainers, fixing bugs, introducing new tests. For instance, in a pull request from Dubbo entitled "Optimize RoundRobinLoadBalance" [33], a developer initially issues three commits with the same message, indicating they belong to the same overall task. Once a reviewer asked for an update on the variable names, the contributor then issued the fourth commit to improve the pull request. Therefore, we did not consider the commits as a good indicator for the number of tasks a pull request contains. Instead, we resort to in-depth manual analysis of the code and commit messages to group

**Table 3.3: Task Definition.**

| **Task Type** | **Definition** |
|---|---|
| Add new feature | Implement a new feature in the system. |
| Change documentation | Changes to the project documentation, addition, deletion or improvement. |
| Add test | Add code to test the software. |
| Remove existing feature | Remove existing functionality |
| Fix bug | Fix unexpected problems and unintended behaviors, e.g. memory error, null pointer exception (NPE). |
| Improve existing feature | Make the existing feature support more functionalities, such as optimization. |
| Improve robustness | To prevent unsafe type, such as restrict object type, from Class to generic class. |
| Refactor code | Change the project code without impacting the software functionality, e.g., improve code readability and maintainability |
| Format code | Apply changes in the formatting of the code, without impacting any code logic, e.g., add or delete white space, remove unnecessary brackets. |
| Revert code | Revert code to the previous version. |
| Compiler annotation related | Improve code syntax to eliminate compiler warnings or add compiler annotations. |

multiple commits into a task, whenever necessary.

We also do not consider that changes in the same commit naturally belong to only one task, we still read the code changes carefully. We filtered out one-commit pull requests when collecting dataset, because multi-commit pull requests are more likely to be tangled than one-commit pull requests. For example, a commit [34] in one of the pull requests in netty, the commit message is *More test cases: Round one.* While it contains two test tasks inside, one is *Tests InternalLoggerFactory.getInstance(Class)*, the other is *Paired with #543, this achieves 100% code coverage with tests in UniqueName(class).*

## 3.2 RQ-1: How prevalent are tangled pull requests?

### 3.2.1 Motivation

In this question, we aim to investigate the prevalence of tangled multi-commit pull requests to assess whether this is a real and practical problem. Understanding the prevalence and characteristics of tangled PRs will help us devise better strategies to handle this problem in the review process, contributing to better communication between developers, better effort prioritization and overall improvement of the quality of software delivered.

### 3.2.2 Approach

We start to investigate the prevalence of tangled PRs by manually inspecting a sample of pull requests. Two annotators, with experience in Java development for more than three years and four experience in other programming languages, classify 640 pull requests into tangled or atomic (not tangled) independently. We evaluate the agreement between annotators using the cohen's Kappa inter-rater reliability level [35]. We obtain an agreement value of 0.89, which indicates near perfect agreement between the two annotators. In cases of disagreement, the two annotators discussed to better explain their chosen category and reach consensus. In the rare cases of persistent disagreement, we involve a third annotator to be the tie-breaker.

After the classification of the sample of 640 pull requests into tangled or atomic, the two annotators jointly categorize the kinds of tasks associated with the tangled pull requests. The annotators used an open card-sort method [36], where the task categories are created during the labeling process and each new category is discussed among annotators and retroactively applied to previously classified pull requests. We present in Table 3.3, the tasks we encountered in our manual investigation, alongside their short description.

### 3.2.3 Results

From the 640 pull requests we manually investigate, 301 (47%) are tangled, i.e., contain two or more development tasks. As shown in Table 3.4, we break down the pull requests based on

**Table 3.4:** Most frequent tangled tasks observed on the 301 tangled pull requests.

| Group | Category | # PRs | % |
|---|---|---|---|
| Good Practice (74.8%) | Fix bug + add/modify test | 112 | 37.2% |
| | Improve feature + add/modify tests | 54 | 17.9% |
| | Add feature + add/modify tests | 50 | 16.6% |
| Bad Practice (25.2%) | Fix two different bugs | 16 | 5.3% |
| | Improve two different existing features | 10 | 3.3% |
| | Add/modify two different test | 7 | 2.3% |
| | Add two different new feature | 6 | 2.0% |
| | Improve existing feature + fix bug | 5 | 1.7% |
| | Refactor code + fix bug | 4 | 1.3% |

the type of tasks combined in a single pull request. From the 301 tangled pull requests, we identify that the majority 225 (74.8%) can be classified as tangled due to good development tasks. These tangled pull requests combine development tasks (adding a new feature, bug fixing, etc.) with test maintenance (update of software test suites), an encouraged practice in software development to keep tests up-to-date and maintain a healthy test coverage.

From the 301 tangled pull requests, 76 (25.2%) pull requests were identified as tangled due to the discouraged practice of combining unrelated tasks in a single pull request. The most common cause of such tangled pull requests are related to fixing two unrelated bugs, which represents 5.3%, followed by pull requests tangled by two or more different feature updates (3.3%) We also find pull requests that modify two tests (2.3%), include two different features in the software project (2%), include bug fixing and feature updates (1.7%), and contain code refactoring and bug fixing (1.3%). There are other cases, not listed in Table 3.4, which occur in less than 4 pull requests, such as reverting code to the previous version, updating code comments, modifying code related to compiler annotation, reformatting code, and removing an existing feature.

**Examples of PR tangled due to good practices.** Fixing a bug and adding/modifying test code is the most popular category of tangled PRs, which is encouraged in software development. The purpose of the PR 3622 in Dubbo [37] was initially to fix a bug. After merging the code changes, the contributor was asked to add test code to the pull request to

keep the test coverage of the project. The PR 5508 of Libgdx [38] provides an example of improving a new feature while maintaining the test suite by modifying current tests.

**Examples of PR tangled due to bad practices.** Fixing two different bugs is the most popular reason developers combine unrelated tasks in a PR. For example, the PR 220 from Netty [39] contains three bug fixes. The first one is *Properly handled SCTP association shutdown event*, the second is *Supported SCTP Unordered Packets*, the last one is *Corrected written bytes count in SctpSendBufferPool*.

The PR 135 from Libdgx [40] is a good example of a PR containing changes in two distinct software features. The OR describes *two minor enhancements/commits to the particle editor*, the first is *add delta multiplier*, the other one is *simpler new emitters*. Another PR in Libdgx [41] contains two tasks, they are minor updates for OpenGL and OpenAL, as the title of the purpose described *iOS Update for OpenGL + OpenAL (minor) #365*.

We found 10 cases of PRs modifying two unrelated test methods. The PR 544 from Netty [42] adds different test cases related to different software features(e.g., log, network, stringUtil, etc). We found 7 cases of PRs adding two different new features. For example, the purpose of a PR in Libdgx [43] is to add *3dapi various small changes #591*, one is to *add reflection color*, another is to *add pinch zoom for CameraInputController*.

> We find that PRs containing more than one task are very common in multi-commit PRs, with 301 (47%) out of 690 PRs being classified as tangled. The majority of tangled PRs (75%) associate the tasks of fixing bugs, improving/adding features with test maintenance, which is considered a good development practice. Still, a quarter of tangled PRs (25%) combine unrelated tasks in the same PR, such as fixing different bugs and working in two different software features.

## 3.3 RQ2: Can we effectively predict whether or not a pull request is tangled?

### 3.3.1 Motivation

Being able to automatically identify tangled multi-commit pull requests can help practitioners in many ways. First, project maintainers can consider the multi-task aspect of tangled pull requests for a fairer assignment of pull request review. Second, quality assurance engineers have a better indicator of the complexity of a pull request, and how much effort they need to put on it, thus they can work more effectively. To the best of our knowledge, we are the first to predict whether a pull request is tangled or not. We run the model only on the tangled PRs that are considered "bad practices" and the PRs contain only one task.

### 3.3.2 Approach

The prediction of whether a pull request is tangled or atomic can be modeled as a binary classification. We first collect a set of metrics from the PR that we believe are relevant to determine if a PR is tangled. Then, we use two classifier models in our study, Logistic Regression [44] and Random Forest [45]. Given the imbalance of our dataset, we experiment with oversampling the minority class (tangled) and under-sampling the majority class (atomic), when training our models. Next, we describe in detail the process we take to train and test our models.

We present in Table 3.5 the metrics we consider relevant for classifying if a PR is tangled. Intuitively, the more files a pull request touches and the more functions in the touched files, the more likely that this pull request is tangled. The number of hunks and tree node operations a pull request touches are added for the same reason. It is natural to consider that the more interactions between maintainers and contributors for a pull request, the more likely that this pull request is tangled, thus the number of review comments is added. If a pull request costs more time to be closed, needs more text to describe itself, this pull request may be complicated, it is more likely to be tangled, thus metrics duration, title length and body length are included. Whether this pull request is proposed by maintainers

of the repository may matter, because maintainers may have their own style for coding, while contributors that are not maintainers of the repository may not always follow their contribution rules, thus they may commit tangled pull requests. We want to know whether a pull request contains both test files and source files is tangled.

**Table 3.5: Selected features for the prediction of tangled and atomic pull requests.**

| Pull request feature | Description |
|---|---|
| #files | Number of files |
| #funcs | Number of functions |
| #hunks | Number of hunks |
| #gumTreeDiff | Number of tree node operations |
| #reviewCommentsCount | Number of review comments |
| mixType | Whether Contains both test file and normal Java file |
| duration | Number of days between open and closed |
| titleLength | Length of title description |
| bodyLength | Length of body description |
| isAuthorMaintainer | Whether the PR author is a maintainer of the repository |

The metrics being used for the prediction of whether a pull request is tangled or not are shown in Table 3.5, only the number of tree node operations is extracted by the tool GumTreeDiff [46], other metrics are extracted by our script.

For all the 640 PRs, 11.7% of them are tangled because of bad practice, 42.0% of them are atomic. The data set is imbalanced, thus the training dataset is resampled using oversampling and Undersampling techniques separately. We used oversampling technique to randomly replicate tangled pull requests to make it as many as the atomic ones and used undersampling to randomly remove samples from atomic pull requests to make it as few as the tangled ones.

The dataset is split into two parts, 70% of them are randomly selected as training set, the remained 30% are used as a test set. Oversampling and undersampling techniques will be applied to balance the training sets separately. Then, we use the training and test set to

evaluate the performance using Random Forest and Logistic Regression classifiers.

**Table 3.6: Confusion matrix.**

| | Identified As | |
|---|---|---|
| Actual | True | False |
| True | True Positive (TP) | False Negative (FN) |
| False | False Positive (FP) | True Negative (TN) |

- **Precision:** It measures the correctness of our model. Precision refers to the number of code change pairs in the same task that are correctly classified to be together by the model divided by the total number of code change pairs that are predicted to be together by the model. It is defined as:

  $Precision = \frac{TP}{TP+FP}$.

- **Recall (TPR):** It measures the completeness of our model. Recall or True Positive Rate (TPR) is defined as the number of code change pairs in the same task that are correctly classified to be together by the model divided by the total number of code changes pairs that should be together by the ground truth. It is defined as:

  $Recall = \frac{TP}{TP+FN}$.

- **FPR:** The false positive rate is calculated as the ratio between the number of negative events wrongly categorized as positive (false positives) and the total number of actual negative events (regardless of classification). It is defined as:

  $FPR = \frac{FP}{FP+TN}$.

- **AUC:** It measures the performance of our model at distinguishing between classes. AUC is defined area under the curve, and the curve is plotted with TPR against the FPR where TPR is on the y-axis and FPR is on the x-axis.

### 3.3.3 Results

As shown in Table 3.7, the best model we obtain is the Random Forest trained using Undersampling approach, with the AUC of 0.87. The second best model is generated by Logistic Regression, which get an AUC of 0.85. The other two models trained by the combination of Random Forest and Oversampling, the combination of Logistic Regression and Undersampling both achieve AUC of 0.84. AUC is used to measure the performance of our models, because the value of AUC will not be affected by different thresholds.

**Table 3.7: Performance of Machine Learning algorithms under different re-sampling techniques for predicting whether a pull request is tangled or not.**

| Algorithm | Precision | Recall | AUC |
|---|---|---|---|
| OverSampling+RandomForest | 0.48 | 0.80 | 0.84 |
| OverSampling+LogisticRegression | 0.43 | 0.87 | 0.85 |
| UnderSampling+RandomForest | 0.37 | 1.0 | **0.87** |
| UnderSampling+LogisticRegression | 0.42 | 0.87 | 0.84 |

The significant features of the best model which is trained by Random Forest using the undersampling strategy are shown in Figure 3.4. The most important feature is the number of tree node operations being touched in a PR, followed by the number of functions in the touched files, the number of characters in the description body of a PR, the number of files being touched in a PR. The number of characters in the description title of a PR, the number of hunks being touched, and the number of days it took to close a PR are also important. the number of review comments, whether a PR touched both source file and test file, whether the contributor who proposed the PR is one of the maintainers of the repository also matter.

In addition to getting the most important features by Random Forest, we also analyze the effect of these features by Logistic Regression. To be specific, we use the odds ratio to quantify the effect of a feature in deciding whether a pull request is tangled or not. Odds ratio of a specific feature is the exponent of the coefficient of that feature in Logistic Regression model [7], it indicates the likelihood of a pull request being tangled which is increased by one unit increment of that feature. If the odds ratio is larger than one, it means

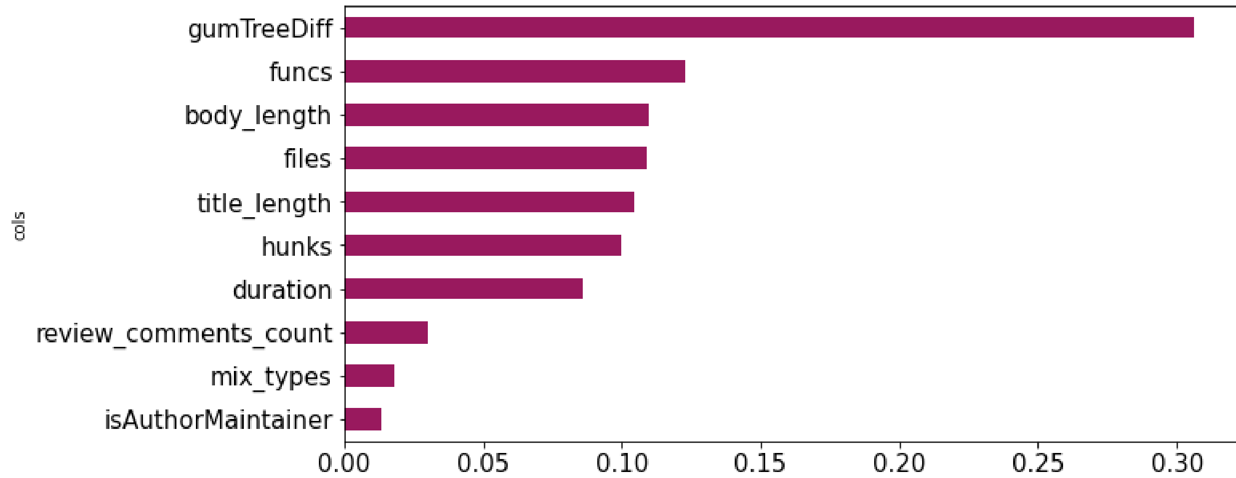


**Figure 3.4: Important features in predicting whether a pull request is tangled or not when using Undersampling and Random Forest.**

**Table 3.8: Odds ratio of features for the prediction of tangled and atomic pull requests using undersmapling and Logistic Regression.**

| Feature | Odds ratio |
|---|---|
| gumTreeDiff | 6.49 |
| funcs | 1.76 |
| mixTypes | 1.27 |
| titleLength | 1.07 |
| duration | 1.03 |
| bodyLength | 0.95 |
| isAuthorMaintainer | 0.91 |
| reviewComments | 0.77 |
| hunks | 0.76 |
| files | 0.46 |

that feature value increased, the more likely a pull request is tangled. As show in Table 3.8, any additional tree node operation increases the likelihood of a pull request being tangled by 549%. Another relevant feature, any additional functions in the touched files of a PR increases the likelihood of a PR being tangled by 76%. If a PR touched both source file and test file, the likelihood of a PR being tangled will increase by 27%, comparing to the PR touched only source files or test files. The PR title also showed to be relevant to our models, showing that any additional character in the title of a PR increases the likelihood of tangled PRs by 7%. Every one more day it took to close a PR, the likelihood of a PR being tangled will increase slightly by 3%.

When the odds ratio of a feature is smaller than one, it means that when feature value increases the likelihood of a PR being tangled will decrease. Every additional character in the description body of a PR decreases the likelihood of it being tangled by 5%. If the contributor who proposed the PR is one of the maintainers of the repository, the likelihood of it being tangled will decrease by 9%, compared to the PR proposed by contributors that are not maintainers of this repository. Every one more review comment for the PR, the likelihood of it being tangled will decrease by 23%. Every one more hunk and every one more file being touched in a PR, the likelihood of it being tangled will decrease by 24% and 54% separately.

> Our best model, Random Forest with undersampling strategy, achieved good performance when predicting tangled PRs, with AUC of 0.87.
>
> The most significant features from the best model are the number of tree node operations and the number of functions being touched in a PR, the number of characters of the description body of a PR, the number of files being touched in a PR and the number of characters in the description title of a PR.
>
> When it comes to the direction of features, we found that any additional tree node operations of the code changes increases the likelihood of a PR being tangled increased by 549%. Any additional functions in the touched files increases the likelihood of tangled PRs by 76%.

# Chapter 4

# Discussion

In this chapter, we discuss our algorithm which predicts whether two lines of code changes belong to the same task.

## 4.1 Can we effectively predict the tasks of tangled pull requests?

In Chapter 3.2, we showed that tangled multi-commit PRs are prevalent and 11.9% of them are due to discouraged practices. In Chapter 3.3 we evaluated a classifier that can help developers at identifying tangled PRs, our model can predict the number of tasks inside a PR with an AUC of 0.87. In this chapter, we discuss a tool that can automatically untangle multi-commit PRs.

We model the problem by focusing on predicting whether two lines of code changes in the same PR should be together. After the prediction, our solution groups the code changes that should be together, we will get several groups of code changes. The code changes in the same group are the code changes that are in the same task which predicted by our algorithm. Thus, instead of predicting all the code changes in the same task at one time, we transfer the question as whether two lines of code changes in the same PR should be together.

For the dataset, we used the tangled PRs that are tangled because of “bad practice”, as classified by our manual work. In our manual work, we also labeled the code changes

with their related task. In this way, we considered code changes in the same task should be together, code changes in different task should not be together.

**Table 4.1: Overview of the pair-wise code metrics.**

| Attribute | Type | Definition | Rational |
|---|---|---|---|
| inSameFile | Boolean | Whether these two lines of code changes in the same file | Our method can only consider code changes inside one file now |
| inSameFunc | Boolean | Whether these two lines of code changes in the same function | If two lines of code in the same function, they maybe contribute to the same functionality |
| inSameHunk | Boolean | Whether these two lines of code changes in the same hunk | Two code changes in the same hunk, means they are continuously changed based on line number |
| inSameRefactorMethod | Boolean | Whether these two lines of code changes in the same refactoring method | Code changes that are detected to be in the same refactoring method to solve a specific code smell should be together |
| inSameSlicing | Boolean | Whether these two lines of code changes in the same slicing | The value of an variable in a line that is affected or can affect the value of another variable in another line, they should be together |
| inSameTreeDiff | Boolean | Whether these two lines of code changes in the same tree node operation | Code changes in the same AST operation between before and after version should be together. |

*Continued on next page*

**Table 4.1 – Continued from previous page**

| **Attribute** | **Type** | **Definition** | **Rational** |
|---|---|---|---|
| commonLocalVarsNum | Integer | Number of common local variables related to these two lines of code changes | Two code changes related to more common local variables, they may have a high cohesion |
| commonFileVarsNum | Integer | Number of common instance variables, static variables related to these two lines of code changes | Two code changes related to more common instance variables or static variables, they may have a high cohesion |
| commonClassesNum | Integer | Number of common classes related to these two lines of code changes | Code changes in a file that are related to other classes, they may have a relationship, like import statement and a statement related to this imported class |
| commonFuncsNum | Integer | Number of common functions related to these two lines of code changes | Code changes that are related to more common functions, they may have similar functionality |
| lineDiff | Integer | Number of code lines between these two code changes if they are in the same file | If two lines of code in the same function, they maybe contribute to the same functionality |

*Continued on next page*

**Table 4.1 – Continued from previous page**

| Attribute | Type | Definition | Rational |
|---|---|---|---|
| callOrCalled | Boolean | Whether the functions these two lines of code changes belong to being called by the other one | Two lines of code belonged to two functions, if one calls the other one, its functionality may be affected by the other one. |

The metrics that may clue whether two lines of code changes should be together are shown in Table 4.1. They are whether these two lines of code changes in the same file, function, hunk, whether they are in the same refactoring method, slicing, same tree node operation. We also consider how many common local variables, instance variables and static variables they have, how many common classes and functions they touched. If they are in the same file, how many lines gap between these two lines of code changes may matters. Whether the functions these two lines belonged to have a call and being called relationship.

For more details, whether two lines of code changes are in the same refactoring method is detected by RefacoringMiner [26], whether they are in the same slicing which includes forward and backward slicing is detected by TinyPDG [47]. Other metrics can be analyzed by Understand [48] and our script.

After collecting all these metrics, we found that code changes pairs that are not the same task are much more than the ones that should be together, hence, our dataset is highly imbalanced. To address this problem, we use both oversampling and undersampling strategies. After we balance the data set, two different machine learning algorithms are used to train our model: Random Forest and Logistic Regression. We use them to do the classification, they predict whether two lines of code changes should be together or not.

As shown in Table 4.2, we found that the model which is trained by the combinations of Logistic Regression and two different re-sampling techniques achieved have the best performance, they both achieved an AUC of 0.74. Followed by the combinations of Random

**Table 4.2: Performance of Machine Learning algorithms under different re-sampling techniques for predicting whether line and line should be together.**

| Algorithm | Precision | Recall | AUC |
|---|---|---|---|
| OverSampling+RandomForest | 0.91 | 0.59 | 0.69 |
| OverSampling+LogisticRegression | 0.96 | 0.58 | **0.74** |
| UnderSampling+RandomForest | 0.92 | 0.58 | 0.69 |
| UnderSampling+LogisticRegression | 0.96 | 0.58 | **0.74** |

Forest and two different re-sampling techniques, they also get the same AUC value of 0.69. In our four models, we found that Logistic Regression performed better than Random Forest.

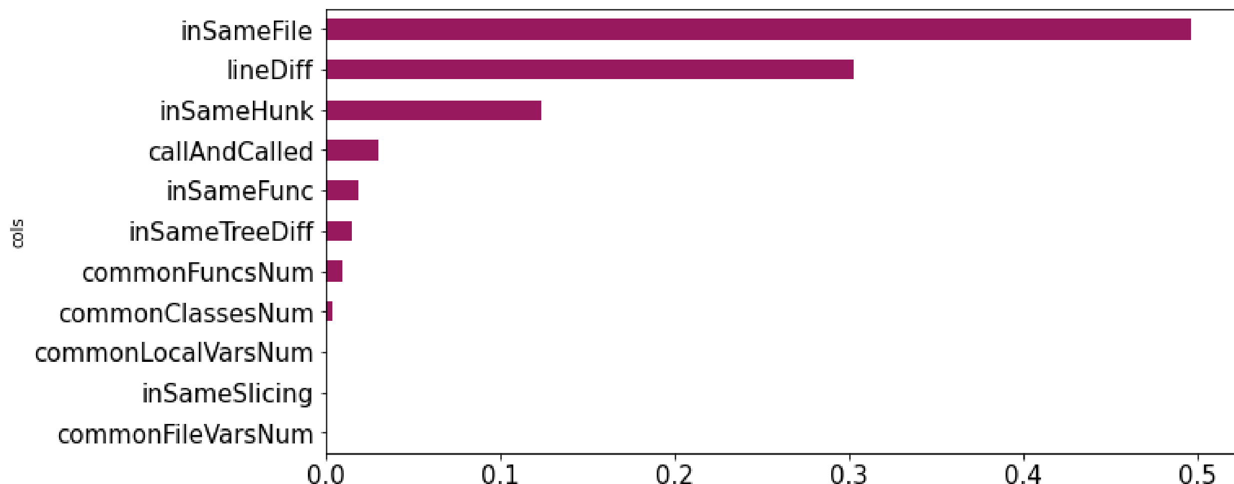


**Figure 4.1: Important features in predicting whether two lines of code changes are in the same task when using Undersampling and Random Forest.**

The important metrics for predicting whether two lines of code changes should be together or not by the model of Random Forest and Undersampling are shown in Figure 4.1. The most important feature is whether they are in the same file, the number of lines between them if they are in the same file and whether they are in the same hunk. Whether the functions they belong to have a call and being called relationships, and whether they are in the same function, whether they are in the same tree node operation are also important. The last two important features are how many common functions, classes they related to.

For the contribution directions of these features, we used odds ratio to analyze them.

**Table 4.3: Odds ratio of features for the prediction of whether two lines of code changes should be together using Logistic Regression and Undersampling.**

| Feature | Odds ratio |
|---|---|
| inSameFile | 16.40 |
| callAndCalled | 2.73 |
| commonFuncsNum | 1.97 |
| inSameHunk | 1.93 |
| commonClassesNum | 1.58 |
| inSameFunc | 1.08 |
| commonLocalVarsNum | 1.02 |
| commonFileVarsNum | 1.00 |
| inSameSlicing | 1.00 |
| lineDiff | 1.00 |
| inSameTreeDiff | 0.40 |

As shown in Table 4.3, we found that most features have the same positive direction, it means the value of these features are increased, the likelihood of line change pairs should be together is increased. If code change pairs that are in the same file, the likelihood of them belong to the same task is increased by 1,540%, compared to those that are not in the same file. If the functions these two lines belonged to have a called and being called relationships, the likelihood of them should be together is increased by 173%, compared to those that do not have. The number of common functions that these two lines related increased by one, the likelihood is increased by 97%. If the code changes are in the same hunk, the likelihood will be increased by 93%. And if they have one more common class related to, the likelihood is increased by 58%, while other features do not contribute a lot. Beyond our expectation, code changes in the same tree node operation, the likelihood of them belong to different tasks are increased by 60%, compared to those are not in the same tree node operation.

# Chapter 5

# Threats to Validity

This chapter discusses the threats to the validity of our study.

## 5.1 Internal Validity

The threats of internal validity may come from the process of building our dataset's ground truth. We understand that manually classifying whether a PR is tangled or not and label lines of code changes with its related task may not be always correct. Considering this limitation, we have the insight of a third developer to resolve the conflicting opinions of the two developers who perform the manual work. As untangling PRs is a time-consuming and labor-intensive task, we take the advantage of public information of a PR from the GitHub web page, which is provided by contributors and maintainers.

## 5.2 External Validity

The main threat for external validity is that our research focuses on a few selected popular open-source Java projects. Our results may not generalize to projects written in different programming languages or projects with different characteristics, e.g., smaller projects developed by few developers. More research is needed to establish a more comprehensive view of the occurrence and characteristics of tangled pull requests in software development. Our study is performed on multi-commit pull requests, our models will be more general if

we add one-commit pull requests into our study.

# Chapter 6

# Conclusions and Future Work

## 6.1 Conclusions

Developers often submit loosely related or unrelated code changes inside one commit, which is called tangled commit, it is harmful to code review and code recommendation systems. Reviewers need to manually split code changes to different contexts, tangled commits increase the difficulty to understand code changes, while reviewers are more likely to leave feedback to the code they understand. Furthermore, the accuracy of code recommendation systems (e.g., future defects, re-opened bugs, assign the task to related developers) is affected by multi-tasks commits. Thus, tangled commits should be avoided.

Most of the previous studies are performed on commits, in this thesis we tackle tangled pull requests. Pull request is a critical approach for developers to collaborate in software development, which also initiates the following code review and integration. Tangled pull requests are pull requests that contain multi-purpose changes inside one single pull request, those are merged into version control systems have the same impact as tangled commits do.

In this thesis, we conduct a case study on 640 pull requests among eight popular open-source Java projects from GitHub. Through a manual identification, we find that 47% of the pull requests are tangled. In order to further understand the behaviors of tangled pull requests, we perform a qualitative annotation and classify the reasons for tangled pull requests. Among the tangled pull requests, we find that 75% of them are good practice, contributors added or modified test code with its related task(e.g, bug fixing, existing feature

improvement, new feature addition). While the remained 25% are bad practice, the most frequent combinations of tangled tasks are fixing two different bugs, improving two different existing features, adding or modifying test code related to different features, adding two different new features, fixing a bug and improving existing feature, refactoring code and fixing a bug.

To help maintainers avoid merging tangled pull requests without being aware, we propose an algorithm to predict whether a pull request is tangled. Our algorithm achieves and AUC of 0.87. Furthermore, we propose another algorithm to predict whether code changes belong to the same task, which achieves an AUC of 0.74.

We truly believe that our algorithms can contribute to avoiding tangled pull requests, which facilitates code review and improve the accuracy of code recommendation systems.

## 6.2 Future Work

We believe that our thesis makes a positive contribution to understanding the problem of tangled pull requests and proposes approaches that can help developers untangle pull requests. However, there are still many challenges that need to be tackled in helping code review and improve the accuracy of code prediction models. We now highlight avenues for future work.

**Investigating contributors and maintainers who proposed the pull requests.** The ground truth of whether a pull request is tangled or not is classified by our researchers. We do believe it would be better to have the insight of contributors and maintainers who proposed the pull requests.

**Developing a tool to untangle code changes automatically.** It will be helpful to develop a tool to untangle code changes automatically before commit, code changes that belong to the same task are inside one commit, while code changes that belong to different tasks are in different commits. Thus, code review will be much easier and bug prediction models will be more accurate.

**Extending to other programming languages.** Our study is performed on popular open-source Java projects, thus our findings are not generalized. Even though Java is popular, there are still other popular programming languages. So, future work will be performed on other programming languages.

# Bibliography


[1] Y. Tao, Y. Dang, T. Xie, D. Zhang, and S. Kim, "How do software engineers understand code changes? an exploratory study in industry," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, 2012, pp. 1–11.

[2] Y. Tao and S. Kim, "Partitioning composite code changes to facilitate code review," in *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*. IEEE, 2015, pp. 180–190.

[3] K. Herzig and A. Zeller, "The impact of tangled code changes," in *Proceedings of the 10th Working Conference on Mining Software Repositories*. IEEE Press, 2013, pp. 121–130.

[4] T. Zimmermann, R. Premraj, and A. Zeller, "Predicting defects for eclipse," in *Third International Workshop on Predictor Models in Software Engineering (PROMISE'07: ICSE Workshops 2007)*. IEEE, 2007, pp. 9–9.

[5] Y. Kamei, E. Shihab, B. Adams, A. E. Hassan, A. Mockus, A. Sinha, and N. Ubayashi, "A large-scale empirical study of just-in-time quality assurance," *IEEE Transactions on Software Engineering*, vol. 39, no. 6, pp. 757–773, 2012.

[6] E. Shihab, A. Ihara, Y. Kamei, W. M. Ibrahim, M. Ohira, B. Adams, A. E. Hassan, and K.-i. Matsumoto, "Predicting re-opened bugs: A case study on the eclipse project," in *2010 17th Working Conference on Reverse Engineering*. IEEE, 2010, pp. 249–258.

[7] ——, "Studying re-opened bugs in open source software," *Empirical Software Engineering*, vol. 18, no. 5, pp. 1005–1042, 2013.

[8] X. Xia, D. Lo, X. Wang, X. Yang, S. Li, and J. Sun, "A comparative study of supervised learning algorithms for re-opened bug prediction," in *2013 17th European Conference on Software Maintenance and Reengineering*. IEEE, 2013, pp. 331–334.

[9] J. Anvik, L. Hiew, and G. C. Murphy, "Who should fix this bug?" in *Proceedings of the 28th international conference on Software engineering*, 2006, pp. 361–370.

[10] X. Xia, D. Lo, X. Wang, and B. Zhou, "Accurate developer recommendation for bug resolution," in *2013 20th Working Conference on Reverse Engineering (WCRE)*. IEEE, 2013, pp. 72–81.

[11] M. Dias, A. Bacchelli, G. Gousios, D. Cassou, and S. Ducasse, "Untangling fine-grained code changes," in *Software Analysis, Evolution and Reengineering (SANER), 2015 IEEE 22nd International Conference on*. IEEE, 2015, pp. 341–350.

[12] M. Wang, Z. Lin, Y. Zou, and B. Xie, "Cora: decomposing and describing tangled code changes for reviewer," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 1050–1061.

[13] M. Barnett, C. Bird, J. Brunet, and S. K. Lahiri, "Helping developers help themselves: Automatic decomposition of code review changesets," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1. IEEE, 2015, pp. 134–144.

[14] S. Hayashi, D. Hoshino, J. Matsuda, M. Saeki, T. Omori, and K. Maruyama, "Historef: A tool for edit history refactoring," in *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, 2015, pp. 469–473.

[15] R. Arima, Y. Higo, and S. Kusumoto, "A study on inappropriately partitioned commits: How much and what kinds of ip commits in java projects?" in *Proceedings of the 15th International Conference on Mining Software Repositories*. ACM, 2018, pp. 336–340.

[16] Z. Liu, X. Xia, C. Treude, D. Lo, and S. Li, "Automatic generation of pull request descriptions," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 176–188.

[17] N. Tsantalis, A. Chatzigeorgiou, G. Stephanides, and S. T. Halkidis, "Design pattern detection using similarity scoring," *IEEE transactions on software engineering*, vol. 32, no. 11, pp. 896–909, 2006.

[18] N. Tsantalis and A. Chatzigeorgiou, "Identification of move method refactoring opportunities," *IEEE Transactions on Software Engineering*, vol. 35, no. 3, pp. 347–367, 2009.

[19] N. Tsantalis, A. Ketkar, and D. Dig, "Refactoringminer 2.0," *IEEE Transactions on Software Engineering*, 2020.

[20] D. Costa, A. Andrzejak, J. Seboek, and D. Lo, "Empirical study of usage and performance of java collections," in *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*, 2017, pp. 389–400.

[21] D. E. D. Costa, C.-P. Bezemer, P. Leitner, and A. Andrzejak, "What's wrong with my benchmark results? studying bad practices in jmh benchmarks," *IEEE Transactions on Software Engineering*, 2019.

[22] "Untangling java code changes manual work link," May 2020. [Online]. Available: https://drive.google.com/file/d/1WTzEugQW0I-PATXxLlEitdkrWafCPdDI/view?usp=sharing

[23] W. Muylaert and C. De Roover, "Untangling composite commits using program slicing," in *2018 IEEE 18th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 2018, pp. 193–202.

[24] S. Sothornprapakorn, S. Hayashi, and M. Saeki, "Visualizing a tangled change for supporting its decomposition and commit construction," in *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*, vol. 1. IEEE, 2018, pp. 74–79.

[25] K. Maruyama, S. Hayashi, and T. Omori, "Changemacrorecorder: Recording fine-grained textual changes of source code," 03 2018, pp. 537–541.

[26] N. Tsantalis, M. Mansouri, L. M. Eshkevari, D. Mazinanian, and D. Dig, "Accurate and efficient refactoring detection in commit history," in *Proceedings of the 40th International Conference on Software Engineering*, ser. ICSE '18. New York, NY, USA: ACM, 2018, pp. 483–494. [Online]. Available: http://doi.acm.org/10.1145/3180155.3180206

[27] GitHub, "Pull request," August 2020. [Online]. Available: https://docs.github.com/en/github/collaborating-with-issues-and-pull-requests/about-pull-requests

[28] Wikipedia, "Github," Nov 2018. [Online]. Available: https://en.wikipedia.org/wiki/GitHub

[29] GitHub, "The state of the octoverse," Nov 2018. [Online]. Available: https://octoverse.github.com/

[30] H. Borges, A. Hora, and M. T. Valente, "Predicting the popularity of github repositories," in *Proceedings of the The 12th International Conference on Predictive Models and Data Analytics in Software Engineering*, 2016, pp. 1–10.

[31] GitHub, "Rest api v3," August 2020. [Online]. Available: https://developer.github.com/v3/

[32] hassan.mansour, "Refactoring aware commit review," October 2019. [Online]. Available: https://chrome.google.com/webstore/detail/refactoring-aware-commit/lnloiaibmonmmpnfibfjjlfcddoppmgd

[33] apache/dubbo, "apache/dubbo/commit/c60c54," Mar 27 2019. [Online]. Available: https://github.com/apache/dubbo/pull/3750/commits/c60c549f4822259ddbbcbaae3efbd179139d320f

[34] netty/netty, "netty/netty/pull/544/commits/6e3b9ed634df77933ccc10e545a2b265bdee4cf2," Aug 20 2012. [Online]. Available: https://github.com/netty/netty/pull/544/commits/6e3b9ed634df77933ccc10e545a2b265bdee4cf2

[35] J. Cohen, "A coefficient of agreement for nominal scales," *Educational and psychological measurement*, vol. 20, no. 1, pp. 37–46, 1960.

[36] S. Fincher and J. Tenenberg, "Making sense of card sorting data," *Expert Systems*, vol. 22, no. 3, pp. 89–93, 2005. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1111/j.1468-0394.2005.00299.x

[37] apache/dubbo, "apache/dubbo/pull/3622," Mar 8 2019. [Online]. Available: https://github.com/apache/dubbo/pull/3622

[38] libgdx/libgdx, "libgdx/libgdx/pull/5508," Mar 8 2019. [Online]. Available: https://github.com/libgdx/libgdx/pull/5508

[39] netty/netty, "netty/netty/pull/220," Mar 5 2012. [Online]. Available: https://github.com/netty/netty/pull/220

[40] libgdx/libgdx, "libgdx/libgdx/pull/135," Dec 2 2012. [Online]. Available: https://github.com/libgdx/libgdx/pull/135

[41] ——, "libgdx/libgdx/pull/365," May 3 2013. [Online]. Available: https://github.com/libgdx/libgdx/pull/365

[42] netty/netty, "netty/netty/pull/544," Aug 21 2012. [Online]. Available: https://github.com/netty/netty/pull/544

[43] libgdx/libgdx, "libgdx/libgdx/pull/591," Sep 14 2013. [Online]. Available: https://github.com/libgdx/libgdx/pull/591

[44] D. G. Kleinbaum, K. Dietz, M. Gail, M. Klein, and M. Klein, *Logistic regression*. Springer, 2002.

[45] L. Breiman, "Random forests," *Machine learning*, vol. 45, no. 1, pp. 5–32, 2001.

[46] J. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus, "Fine-grained and accurate source code differencing," in *ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden - September 15 - 19, 2014*, 2014, pp. 313–324. [Online]. Available: http://doi.acm.org/10.1145/2642937.2642982

[47] YoshikiHigo, “Tinypdg,” October 2019. [Online]. Available: https://github.com/YoshikiHigo/TinyPDG

[48] scitools, “Understand,” October 2019. [Online]. Available: https://scitools.com/features/