Automating Software Changes via Recommendation Systems

Xiaoyu Liu
Southern Methodist University, xiaoyul@smu.edu

Follow this and additional works at: https://scholar.smu.edu/engineering_compsci_etds

Part of the Software Engineering Commons

Recommended Citation
https://scholar.smu.edu/engineering_compsci_etds/11

This Dissertation is brought to you for free and open access by the Computer Science and Engineering at SMU Scholar. It has been accepted for inclusion in Computer Science and Engineering Theses and Dissertations by an authorized administrator of SMU Scholar. For more information, please visit http://digitalrepository.smu.edu.
AUTOMATING SOFTWARE CHANGES
VIA RECOMMENDATION SYSTEMS

Approved by:

__________________________________________
Dr. LiGuo Huang, Associate Professor

__________________________________________
Dr. Jennifer Dworak, Associate Professor

__________________________________________
Dr. Sukumaran Nair, Professor

__________________________________________
Dr. Vincent Ng, Professor

__________________________________________
Dr. Jeff Tian, Professor
AUTOMATING SOFTWARE CHANGES
VIA RECOMMENDATION SYSTEMS

A Dissertation Presented to the Graduate Faculty of the
Lyle School of Engineering
Southern Methodist University
in
Partial Fulfillment of the Requirements
for the degree of
Doctor of Philosophy
with a
Major in Computer Science
by

Xiaoyu Liu
(B.S., Nanjing University of Posts and Telecommunications, China)
(M.S., Southern Methodist University)

December 21, 2019
ACKNOWLEDGMENTS

I would like to thank my advisor, LiGuo Huang and my committee member, Vincent Ng, then many colleagues in the Department of Computer Science and Engineering at Southern Methodist University. I am also very grateful to my parents for being so supportive on me.
As the complexity of software systems is growing tremendously, it came with increasingly sophisticated data provided during development. The systematic and large-scale accumulation of software engineering data opened up new opportunities that infer information appropriately can be helpful to software development in a given context. This type of intelligent software development tools came to be known as recommendation systems [113].

Recommendation Systems in Software Change (RSSCs) share commonalities with conventional recommendation systems: mainly in their usage model, the usual reliance on data mining, and in the predictive nature of their functionality [113]. So a major challenge for designing RSSCs is to automatically and accurately interpret the highly technical data stored in software repositories. Realizing such challenge, many RSSCs are proposed. However, existing works either rely on large amount of historical data or suffers from low accuracy, which makes the existing systems hardly to be applied practically.

In this dissertation we develop techniques that aid developers in overcoming one major challenge in software development and evolution: automated software change. We introduce various of recommendation systems: We start with proposing a new approach and tool (namely, CHIP) to predict software actual change impact set by leveraging two kinds of code dependency: call and data sharing dependency. For this purpose, CHIP employs novel extensions (dependency frequency filtering and shared data type idf filtering) to reduce false positives. Then in order to help
developers to understand software change intents, we propose AutoCILink to automatically identify code to untangled change intent links with a pattern-based link identification system (AutoCILink-P) and a supervised learning-based link classification system (AutoCILink-ML). In addition, to help developers in finding appropriate API to adopt, we propose an approach called RecRank, which applies a novel ranking-based discriminative approach leveraging execution path features to improve Top-1 API recommendation. Furthermore, to resolve license restriction conflicts introduced in software change, we illustrate Automatic License Predictor (ALP), a novel learning-based method and tool for predicting licenses as software changes. We illustrate that the techniques presented in this dissertation represent significant advancements in software development and evolution through a series of empirical evaluation that demonstrate the effectiveness of these approaches and the benefits they provide developers.
**TABLE OF CONTENTS**

<table>
<thead>
<tr>
<th>LIST OF FIGURES</th>
<th>xii</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIST OF TABLES</td>
<td>xiv</td>
</tr>
</tbody>
</table>

**CHAPTER**

1. **INTRODUCTION** ................................................................. 1
   1.1. Motivation ................................................................. 1
   1.2. Research Context: Recommendation Systems in Software Change .... 2
   1.3. Contributions and Outline ........................................... 4

2. **RELATED WORKS** ................................................................. 7
   2.1. Work Related to Software Actual Change Impact Set Prediction ... 7
       2.1.1. Mining Software Repositories Based Analysis ............... 7
       2.1.2. Textual Analysis Techniques .................................... 7
       2.1.3. Dynamic Analysis Techniques .................................. 8
       2.1.4. Structural Analysis Techniques ............................... 8
   2.2. Work Related to Changed Source Code and Change Intent Linking .. 9
       2.2.1. Analysis of commits and changed source code .............. 9
       2.2.2. Linking commits to software artifacts ...................... 10
   2.3. Work Related to API Recommendation .................................. 10
       2.3.1. Code Suggestion based on Mined Software Repositories ..... 10
       2.3.2. Code Suggestion Using Statistical Models ................... 11
       2.3.3. Code Suggestion based on Code Structure ................... 11
   2.4. Work Related to Predicting Software Licenses for Changed Source Code ...................................................... 12
       2.4.1. Software license Identification and Classification ......... 12
2.4.2. License Adoption and Evolution ........................................ 13

3. TOWARDS AUTOMATED SOFTWARE CHANGE – SUPPORTING SOFTWARE ACTUAL CHANGE IMPACT SET PREDICTION ... 15

3.1. Background & Motivation .................................................... 17

3.1.1. Studies on Call Dependency .......................................... 17

3.1.2. Studies on Program Dependence Graph (PDG) and Evolutionary Couplings Capturing ................................. 18

3.1.2.1. Program Dependence Graph (PDG) ............................... 19

3.1.2.2. Evolutionary Coupling ............................................... 20

3.1.3. Motivating Example ..................................................... 21

3.2. Fundamentals of Data Sharing Dependency ............................ 24

3.2.1. Definitions of Data Sharing Dependency ............................ 24

3.2.2. Capturing Data Sharing Dependency ................................ 24

3.2.2.1. Field Access .......................................................... 25

3.2.3. Field Modification ........................................................ 25

3.2.4. Parameter Passing ...................................................... 25

3.2.5. Cross Events ............................................................. 25

3.3. Design of CHIP ............................................................... 26

3.3.1. Usage Scenario ........................................................... 26

3.3.2. CHIP Framework ......................................................... 27

3.3.3. Basic Actual Change Impact Set Predictor ......................... 29

3.3.4. Extension I: Dependency Frequency Filter ....................... 33

3.3.5. Extension II: Inverse Data Frequency (idf) Filter .............. 34

3.4. Empirical Evaluation ........................................................ 36

3.4.1. Experiment Design ....................................................... 37
3.4.1.1. Datasets ................................................. 37
3.4.1.2. Evaluation Metrics .................................... 39
3.4.1.3. Time efficiency Compared to PDG ................. 40
3.4.2. Experimental Results ..................................... 41
3.5. Conclusion .................................................. 57
3.6. Bibliographical Notes ....................................... 58
4. TOWARDS AUTOMATED SOFTWARE CHANGE – RECOMMENDING CHANGED SOURCE CODE AND CHANGE INTENT LINKS ... 59
4.1. Motivating Examples ....................................... 62
4.2. Approach ..................................................... 64
4.2.1. Overview .................................................. 64
4.2.2. Text preprocessing ....................................... 64
4.2.2.1. Untangling change intents ......................... 64
4.2.2.2. Building enriched untangled change intents ...... 65
4.2.2.3. Extracting terms and changed entities .......... 66
4.2.3. Pattern-based Link Identification System (AutoCILink-P) . . . 66
4.2.3.1. Link identification using regular expression (AutoCILink-P-R) ............................................. 66
4.2.3.2. Link identification using vocabulary similarity (AutoCILink-P-V) ......................................... 68
4.2.4. Supervised learning-based link classification system (AutoCILink-ML) ............................................. 69
4.2.4.1. Creating training instances ......................... 70
4.2.4.2. Training and applying link classifier ............ 73
4.3. Empirical Evaluation ......................................... 73
4.3.1. Data preparation ......................................... 73
6. TOWARDS AUTOMATED SOFTWARE CHANGE – RECOMMENDING LICENSES FOR CHANGED SOURCE CODE

6.1. Background & Problem Definition

6.1.1. Software License Background

6.1.2. Problem Statement

6.2. Data Preparation

6.3. Baseline Approaches

6.3.1. Ninka

6.3.2. Caller-Callee (CC)

6.3.3. Previous Version (Prev)

6.4. Our Approach

6.4.1. Step 1: Building the Basic ALP System

6.4.2. Step 2: Modeling the Previous License

6.4.3. Step 3: Adding New Knowledge Sources

6.4.3.1. Extracting features from software documents

6.4.3.2. Extracting features from co-changed files

6.4.4. Step 4: Modeling Conflicts

6.5. Empirical Evaluation

6.5.1. Experimental Setup

6.5.2. Results and Discussion

6.6. Summary

6.7. Bibliographical Notes

7. CONCLUSION & FUTURE WORK

7.1. Concluding Remarks
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Venn diagram of class-level call dependencies in CHIP (CHIP-Call) versus class-level PDG in jEdit</td>
<td>19</td>
</tr>
<tr>
<td>3.2</td>
<td>An example of actual change impact set prediction</td>
<td>21</td>
</tr>
<tr>
<td>3.3</td>
<td>An example of code snippet</td>
<td>21</td>
</tr>
<tr>
<td>3.4</td>
<td>Examples of data sharing dependencies</td>
<td>26</td>
</tr>
<tr>
<td>3.5</td>
<td>An example data sharing dependency set generated from Figure 3.4</td>
<td>27</td>
</tr>
<tr>
<td>3.6</td>
<td>Overview of software actual CHange Impact Set Prediction (CHIP)</td>
<td>28</td>
</tr>
<tr>
<td>3.7</td>
<td>An example of comprehensive set of dependencies with call dependencies (solid links) and data sharing dependencies (dash links) for the basic predictor</td>
<td>32</td>
</tr>
<tr>
<td>3.8</td>
<td>Trimmed call dependencies (solid links) and data sharing dependencies (dash links) after applying the $idf$ filter on Figure 7</td>
<td>36</td>
</tr>
<tr>
<td>3.9</td>
<td>-a-d. Trend analysis on four systems in class-level prediction: F2-score (%) of prediction by $P_{cd}$+ext, $P_d$+ext, $P_c$, $P_{pdg}$ and $P_e$ under difference $idf$ threshold</td>
<td>47</td>
</tr>
<tr>
<td>3.10</td>
<td>-a-d. Trend analysis on four systems in method-level prediction: F2-score (%) of prediction by $P_{cd}$+ext, $P_d$+ext, $P_c$, $P_{pdg}$ and $P_e$ under difference $idf$ threshold</td>
<td>47</td>
</tr>
<tr>
<td>4.1</td>
<td>An example of how software developers link changed source code files to untangled change intents</td>
<td>60</td>
</tr>
<tr>
<td>4.2</td>
<td>An example of $AutoCILink$-P “greedy” matching</td>
<td>63</td>
</tr>
<tr>
<td>4.3</td>
<td>$AutoCILink$ system workflow</td>
<td>64</td>
</tr>
<tr>
<td>4.4</td>
<td>Average percentage of linked changed source code files per untangled change intent count across all 19 projects</td>
<td>72</td>
</tr>
<tr>
<td>5.1</td>
<td>A code snippet</td>
<td>88</td>
</tr>
</tbody>
</table>
5.2. An API recommendation example from Eclipse .......................... 89
   a. a. Child graph ............................................. 89
   b. b. Parent graph ............................................. 89
5.4. Parent-child graph example ........................................... 89
5.5. Learning curves of API recommendation approaches on the entire
test set ............................................................ 113
6.1. Example illustrating licensing incompatibility ....................... 115
6.2. An example of imported code modules and licenses of FluentList.java .. 125
6.3. A snippet of a software document ..................................... 130
# LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1. Partial execution trace log of jEdit</td>
<td>19</td>
</tr>
<tr>
<td>3.2. Partial execution trace log of jHotDraw</td>
<td>22</td>
</tr>
<tr>
<td>3.3. Predicted Actual Change Impact Set with Class <em>RoundRectangleFig-ure</em> by Basic Predictor</td>
<td>32</td>
</tr>
<tr>
<td>3.4. Top 5 Objects with the Lowest <em>idf</em></td>
<td>34</td>
</tr>
<tr>
<td>3.5. Predicted Actual Change Impact Set with Class <em>RoundRectangleFig-ure</em> by Basic Predictor + <em>idf</em> Extension</td>
<td>37</td>
</tr>
<tr>
<td>3.6. Characteristics of Four Open Source Systems</td>
<td>38</td>
</tr>
<tr>
<td>3.7. Time Efficiency of <em>P</em>cd+ext and <em>P</em>pdg on Four Systems</td>
<td>40</td>
</tr>
<tr>
<td>3.10. Prediction Count Before and After Extensions Applied on <em>P</em>cd</td>
<td>44</td>
</tr>
<tr>
<td>3.11. Four Systems at Class-level: Paired T-test Results, Mean and Confidence Interval (CI) of F2-score by <em>P</em>cd+ext vs. <em>P</em>c, <em>P</em>cd+ext vs. <em>P</em>pdg and <em>P</em>cd+ext vs. <em>P</em>e</td>
<td>48</td>
</tr>
</tbody>
</table>
3.15. Five Actual Change Impact Scenarios at Method-level: Precision (P), Recall (R) and F2-score (F2) by $P_{cd}+ext$-overall, $P_{cd}+ext$-FA, $P_{cd}+ext$-FM, $P_{cd}+ext$-PP, $P_{cd}+ext$-CE and $P_c$ (Part I) ............... 53

3.16. Five Actual Change Impact Scenarios at Method-level: Precision (P), Recall (R) and F2-score (F2) by $P_{cd}+ext$-overall, $P_{cd}+ext$-FA, $P_{cd}+ext$-FM, $P_{cd}+ext$-PP, $P_{cd}+ext$-CE and $P_c$ (Part II) ............... 54

3.17. Five Actual Change Impact Scenarios at Class-level: Paired T-test Results, Mean and Confidence Interval (CI) of F2-score by $P_{cd}+ext$ vs. $P_c$ ............................................................ 55

3.18. Five Actual Change Impact Scenarios at Method-level: Paired T-test Results, Mean and Confidence Interval (CI) of F2-score by $P_{cd}+ext$ vs. $P_c$ ................................................ 56

4.1. Dataset from 19 open source projects .................................. 74

4.2. Evaluation results of AutoCILink and baseline approaches ............ 78

4.3. Paired t-test results on accuracy of AutoCILink-ML vs other approaches accuracy ................................................ 78

4.4. Evaluation results of AutoCILink-ML and AutoCILink-P ............... 80

4.5. Feature ablation results in average accuracy ........................... 81

4.6. Paired t-test results on AutoCILink feature selection ................. 81

4.7. Distribution of misclassified pairs of changed source code and enriched untangled change intents ......................... 83

5.1. Probability scores of candidate APIs (Part I) ....................... 91

5.2. Probability scores of candidate APIs (Part II) ....................... 92

5.3. Dataset statistics ....................................................... 104

5.4. Re-implemented and original Gralan results ........................... 105

5.5. Evaluation results of API recommendation systems (Part I) ........ 107

5.6. Evaluation results of API recommendation systems (Part II) ........ 108

5.7. Evaluation results for different model-feature combinations .......... 110

5.8. Feature ablation results of RecRank ................................. 112

6.1. Annotation examples .................................................. 120

6.2. Dataset statistics ....................................................... 122

a. Overall statistics ....................................................... 122
b. Per-license frequencies ........................................... 122

c. License change statistics ........................................ 122

6.4. Five-fold cross-validation results. The strongest result in each column is boldfaced. .................................................. 138

6.5. Five-fold cross-validation results of systems on Ninka-detectable, Ninka-undetectable, and conflict instances ................................. 140

6.6. Examples of errors made by ALP2-Ranker1 .......................... 143


1.1. Motivation

The growing complexity of software has caused software development to become more and more challenging and knowledge intensive, especially during software change process when some software data modified in some cases multiple times a day [59], which costs a significant amount of time and human efforts. In addition, such rapid change of software requires developers continuously learn how to use new technology and new ideas. However, as Robillard et al. [113] mentioned, we have long since reached the point where the scale of the information space facing a typical developer easily exceeds an individual’s capacity to assimilate it. Software developers and other technical knowledge workers must now routinely spend a large fraction of their working time searching for information, for example, to understand existing code or to discover how to properly implement a feature.

The existing way to solve these challenges, such as training and effective interpersonal communication, are limited by the high cost and low effectiveness. Considering that, intelligent supports on software change are urgently demanded.

Specific challenges in software engineering often stem from difficulties for consuming significant time and human efforts. For instance, when source code changes, it is hard for developers to ensure the consistent changes of its dependent entities (i.e., classes, methods, and variables) when systems evolve and developers change these entities to add new features, retire existing features, or fix bugs. In addition, it is even harder to ensure the compatibility of licenses for the changed source code. According to Vendome et al. [128], existing methods and tools for license prediction ([33, 46, 127, 129]) are insufficient for the challenge of predicting licenses in the presence of software changes. For instance, Ninka [46], a state-of-the-art license detection
method, uses regular expressions to predict licenses by detecting the presence of license copyright and terms in the header comments of the source code files (i.e., file header). Hence, Ninka has no problem with (independently) detecting the original license adopted by the source code file before the code change. However, Ninka cannot detect file dependencies because it cannot take into account the license restriction imposed by the newly imported code module. Failure to do so deprives Ninka of its ability to address the license compatibility issues that arise from code changes.

Furthermore, once source code changes are made in the same commit but for different intents (i.e., tangled changes [53]), it is time-consuming, labor-intensive, and requires a great deal of experience to link changed source code to change intents—developers either ignore or forget implementation details behind the changes when different change intents are aggregated in one single commit message [32].

In software change, Application Programming Interface (API) recommendation is common not only for changed code but also newly added code. For API recommendation, APIs are provided by the library’s designers [99] to enable developers to access the functionality of a code library. However, it takes efforts for developers to be familiar with the capabilities provided by a large number of APIs in the library and pick the correct API for development tasks. Otherwise, usage of incorrect APIs may introduce failure and bugs in the program.

This dissertation presents an in-depth analysis of these key issues and solutions for supporting software changes via recommendation systems.

1.2. Research Context: Recommendation Systems in Software Change

Recommendation Systems in Software Change (RSSCs) are defined as a software application that provides information items estimated to be valuable for a software change task in a given context. Different from other intelligent software change supporting systems, RSSCs’ focus is on providing information as opposed to other services such as build or test automation [113]. Specifically, the information items that can be recommended include but not limited to: source code, software documents (e.g., commit messages, issue report, pull request), tools, commands, and operations.
Many existing recommendation approaches are proposed in supporting software changes, such as applying dynamic analysis [4,8,18,71,101,102,145] and static analysis [15,48,50,51,54,64,79,84,106,143,146] to predict actual change impact set; using API usage patterns [11,40,60,126] or statistical learning [26,77,96,97] to recommend API; using information retrieval (IR)-based approaches [17,22,98,121,137] to generate links between changed code and software documents; among the others. However, those key tasks for change and maintenance lack effective industrial-grade solutions to support developers and testers. Specifically, dynamic analysis approaches for actual change impact set prediction heavily relies on a complete system execution profile, which is usually difficult to acquire and only available late during development (after a majority of the system has been implemented) due to the unavailability of complete tasks. Some static analysis approaches for actual change impact sets prediction require access to a long history of code changes to capture the extent to which software artifacts were changed together, namely, evolutionary couplings. Meanwhile, for API recommendation, the usage of API usage patterns, such as in [97], is sensitive to the presence of overlapping features and irrelevant features. Specifically, if two features encode overlapping information (e.g., two features are computed based on the same API in the preceding context), it will undesirably amplify the importance of this API in the prediction process, thus possibly harming model performance. Irrelevant features (i.e., features that are largely not predictive of the target API), too, could be harmful. For source code to change intent linking challenge, applying Information Retrieval (IR)-based approaches such as Latent Semantic Indexing (LSI), Vector Space Model (VSM), association-based approach cannot adequately address this challenge because the changed entities extracted from source code (e.g., identifier, comments, string literals) could be very different from what is described in change intents described in commit messages and other related software documents (e.g., issue reports, pull requests, etc.)

Thus, this dissertation presents four recommendation systems to support one major challenge: automated software change. Specifically, a software actual change impact set recommendation system that leverages both call and data sharing de-
pendency to recommend co-changes of known software changes, a source code to change intent linkage system that combines a pattern-based phase and a supervised learning-based phase, an API recommendation system that applies a novel ranking-based discriminative model by leveraging API usage path features, and a software license recommendation system to recommend licenses as software change.

1.3. Contributions and Outline

In summary, the core thesis of this dissertation is: Automating the process of representing and relating software artifacts in source code and natural language allow for more efficient and effective software change. To investigate this thesis, we develop and empirically evaluate models and approaches that aid in automating the development and change in software. In particular, these approaches focus on (i) supporting software actual change impact set prediction, (ii) improving API recommendation accuracy, (iii) identifying links between changed source code files and untangled change intents, and (iv) predicting licenses for changed source code.

In Chapter 2 we discuss work related to (i) software actual change impact set prediction, (iii) changed source code files to untangled change intents linking, (iii) software licensing for changed source code, and (iv) API recommendation.

As the first step towards automated software change, in Chapter 3 an automated tool, CHange Impact set Predictor (CHIP), is proposed to provide developers with change impact set predictions. To capture both call and data sharing dependencies across software classes and methods, a dynamic mechanism is used. In addition, a dependency frequency filter and a novel data type inverse document frequency (idf) filter are used to reduce false positives in prediction (ie, wrong change impact set) while maintaining the recall. Our experiment results show that code dependencies complement each other in the actual change impact set prediction. Furthermore, data sharing dependency is particularly useful in specific change impact scenarios including “move refactoring”, “remove classes and statement”, “bug fixing”, “functional improvement”, and “code replacement”. These evaluation results not only demonstrate the usefulness of our approach but also help to better understand the benefits
of data sharing dependency with respect to call dependency for change impact set prediction.\textsuperscript{1}

A further investigation into automated software change by understanding change intents, in \textbf{Chapter 4} we develop a new tool to identify links between changed source code files and untangled change intents. First, we have annotated a new corpus with links between untangled change intents and changed source code from 19 projects and make this corpus publicly available to stimulating research on this challenge. Then, we leverage some recurrently used patterns to design a pattern-based link identification system (a.k.a. AutoCILink-P). These patterns are designed with regular expressions that (1) extract changed entities (i.e., classes that encapsulate data and behavior [36]) from changed code files and (2) compute the similarity between these changed entities and the terms that appear in untangled change intents in commit messages as well as in other related software documents (i.e., issue reports and pull requests). To further our attempt to automatically generate links from code changes to untangled change intents, we have developed a supervised learning based link classification system (a.k.a., AutoCILink-ML) using novel features specifically designed for this challenge.\textsuperscript{2}

To support source code suggestion in automated software change, in \textbf{Chapter 5} we present a novel discriminative ranking approach that employs a novel kind of features based on usage paths to automatically recommend top-1 APIs based on the top-10 API candidates suggested by Gralan. In an evaluation on eight large-scale open source projects, RecRank outperforms APIREC with respect to two evaluation metrics, top-1 recommendation accuracy and mean reciprocal rank (MRR), a commonly used metric for evaluating ranking tasks in information retrieval, achieving the state-of-the-art results.\textsuperscript{3}

\textsuperscript{1}CHIP was published at the Journal of Software: change and Process (JSEP) [74].
\textsuperscript{2}AutoCILinks was published at the 34th IEEE International Conference on Software Maintenance and change (ICSME 18) [75].
\textsuperscript{3}RecRank was published at the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE 18) [76].
To solve the problem of license restriction conflicts as software changes, in Chapter 6 we devise a novel learning based method and supporting tool called Automatic License Prediction (ALP) for automatically predicting source code file-level licenses for code changes. Leveraging the recent successes of machine learning methods in empirical SE research, we propose a learning-based ALP system. At the core of ALP are four key ideas: (1) exploiting a rich set of features extracted from its inline text; (2) modeling the license of the previous version of the file; (3) exploiting features extracted from the associated software documents and co-changed files; and (4) identifying and resolving incompatibilities.4

In Chapter 7 we offer general conclusions that summarize the contributions of this dissertation and discuss two major avenues for future research motivated by the outcomes of the work described in this dissertation.

4ALP will be published at the 34rd ACM/IEEE International Conference on Automated Software Engineering (ASE 19).
Chapter 2
RELATED WORKS

As mentioned, recommendation systems in software development and evolution provide information based on given context to solve some software development and evolution challenges. This chapter examines related work about each specific domain in our two major challenges mentioned in Chapter 1: software actual change impact set prediction, changed source code files to untangled change intents linking, software licensing for changed source code, and API recommendation.

2.1. Work Related to Software Actual Change Impact Set Prediction

2.1.1. Mining Software Repositories Based Analysis

This class of research uses data mining approaches on historical code change repositories to detect frequent impact set patterns. For instance, Zimmerman [146], Mondal [89] and Moonen [90] all use association rules based mining on CVS logs for detecting evolutionary coupling among source code entities. Ying [143] uses the similar approach to identify files that frequently change together. Steff [122] performs change impact analysis by analyzing historical change couplings. CHIANTI and its application [111] do impact analysis by analyzing two versions of an application and decomposes their difference into a set of atomic change. This kind of approach relies on the quality of software historical repositories. If insufficient historical data is available (such as new project or project with incomplete repository), mining software repository based techniques are inapplicable. Moreover, there are changes that contradict the frequent impact set patterns.

2.1.2. Textual Analysis Techniques
Some research work applies information retrieval (IR) techniques on textual data such as comments and/or identifiers in the source code. Gethers [47] presents a change impact analysis technique on textual change request. Kagdi [63] extracts conceptual coupling by comparing the similarity in source code using IR based techniques. Has-san [50] predict change propagation on code textual semantic measures. Some recent works also study change recommendations for bug fixing. Park [104] present their approach not only uses commit history data but also uses the data of supplementary patch to investigate the omission errors reduction in change recommendation. And in their work presented in [103], researchers also indicate that change recommendation for multi-bug fixing has a higher level of severity and tend to be harder to recommend. Xia [139] proposes an approach called SUPLOCATOR to recommend methods that need to be changed for bug fixing based on relationships among code such as method invocation, containment, inheritance, historical changes, content similarities and name similarities. Most of those techniques require developers to encode the implicit relations in the comments and/or identifiers and hence the quality of the change prediction depends on the quality of the encoding. When such kind of data is unavailable or in low quality, those approaches are limited.

2.1.3. Dynamic Analysis Techniques

Major dynamic impact analysis techniques include Dynamic Slicing, CoverageImpact and PathImpact. Dynamic Slicing [4,145] analyzes the change impact by extracting slice from an execution trace. CoverageImpact [3,18,19,101] leverages field data to perform change impact analysis. PathImpact or similar approaches, on the other hand, performs impact analysis on whole profiling [70,71,102], which, however, is difficult to acquire. Moreover, most of these techniques focus on tracing individual system execution paths and focus on method calls [16] while ignoring data (e.g., fields or variables) shared across execution paths.

2.1.4. Structural Analysis Techniques
A number of previous works [9,21,66,115,115,125,135,142] focus on the analysis of structural dependence, most of which leverage the call dependencies among entities (most notably methods and classes) and Program Dependence Graph (PDG) as indicators for change impact set. Other works, such as [52], perform CIA for software architecture evolution relying on architecture models described by architecture description languages (ADLs).

2.2. Work Related to Changed Source Code and Change Intent Linking

2.2.1. Analysis of commits and changed source code

Moreno et al. [91] examine the changed source code in commits and use them to automatically generate release notes with patterns. Hinton et al. [56] discover software release patterns from documents in commits. Several approaches [22,31,73,108] are proposed to automatically generate commit messages from changed source code. Cortes-Coy et al. [31] and Linares-Vasquez et al. [73] present automated tools to generate natural language commit messages by extracting entities and commit stereotypes from changed source code. Different from the code to untangled change intent linking task, their research are built upon the assumption that commit messages can’t reflect all intents of source code changes. Buse et al. [22] present an automatic approach to describe source code modifications using symbolic execution and summarizations. Rastkar et al. [108] propose an approach to describe the motivation of source code changes with multi-document summarization technique. D’Ambros et al. [32] present a tool to augment commits with visual contexts of changes. Herzig et al. [53] analyze the impact of source code changes and find that up to 15% of bug fixes consist of multiple tangled changes. Dias et al. [35] propose a novel approach to group related changed source code together. Similarly, Kreutzer et al. [68] propose an approach to automatically cluster changed source code. Murphy-Hill et al. [94,95] find that 30% of commits contain code refactorings mixed with other code changes. Barnet et al. [14] introduce an automatic technique for decomposing changesets according to code reviews. Kawrykow et al. [65] and Kirinuki et al. [67] present approaches to de-
tect non-essential code differences and tangled changes by mining historical changes. Some related works [56,107] analyze large commits and small source code changes.

2.2.2. Linking commits to software artifacts

Nguyen et al. [98] propose an approach based on text matching and association patterns to find links between bug reports and changed source code. They not only extract text features from bug reports but also from changed source code. Wu et al. [137] present ReLink to automatically recover missing links between bug reports and changed files by text similarity, time intervals, bug owners and change committers. Sun et al. [125] utilize non-source documents in commits to find missing commit-issue links with a VSM model. Le et al. [72] generate links between issue and commits with enriched commit messages that summarize the intents and detailed modifications in commits. Sliwerski et al. [121] propose an approach to identify links between issues and commits and then analyze if commits induce further fix. Bird et al. [17] present LINKSTER to provide query interfaces to locate possible links between issues and commits.

2.3. Work Related to API Recommendation

2.3.1. Code Suggestion based on Mined Software Repositories

Bruch et al. [20] adapt the k-nearest-neighbor algorithm to find method calls to recommend for particular objects. Robbes et al. [112] use change history such as code insertion and modification to improve code completion. Hou et al. [60] present a way of grouping API proposals from historical data for better code completion. Hill et al. [55] build a tool to automatically complete a method by cloned code. Asaduzzaman et al. [10] and Zhang et al. [144] both use parameter filtering to do code recommendation. Omar et al. [100] uses an interactive way to generate code. Reiss et al. [110] and Stolee et al. [124] use semantic search to map retrieved code into what is asked for by users. Thung et al. [126] present an approach that learns from records of other changes made to software systems and compares the textual description of
the requested feature with the textual descriptions of various API methods. Wang et al. [134] propose two quality metrics (succinctness and coverage) for mined usage patterns. Xie et al. [140] present MAPO to generate patterns by mining and ranking frequent sequences in each cluster according to the similarity heuristics of source code such as method names. Most of these approach rely on a large number of software historical repositories. This kind of approach is not applicable when such repository is not available.

2.3.2. Code Suggestion Using Statistical Models

Gu et al. [49] adapt a neural language model named RNN Encoder-Decoder, which encodes a word sequence (user query) into a fixed-length context vector, and generates an API sequence based on the context vector. White et al. [136] apply the RNN-LM model on lexically analyzed source code to code suggestion. Allamanis et al. [5] present NATURALIZE, which learns coding conventions to suggest natural identifier names and formatting conventions. They also apply the binomial model to retrieve source code snippets given a natural language query and retrieve natural language descriptions given a source code query [6]. Maddison et al. [78] use Probabilistic Context Free Grammar (PCFG)-based model to represent source code. McMillan et al. [85] propose a combination of association between queries and functions model and navigation behavior of programmers model to retrieve and visualize relevant functions and their usages. Chan et al. [26] perform API recommendation based on the textual similarity between code and query phrases. CodeHow [77] expands the query with the APIs and performs code retrieval by applying the Extended Boolean model, which considers the impact of both text similarity and potential APIs on code search. MULAPI [141] recommends feature related API from feature request documents.

2.3.3. Code Suggestion based on Code Structure

Asaduzzaman et al. [11] recommend API by ranking the similarities between code contexts and the context of the target API method call. Raychev et al. [109] extract
indexed sequences of method calls and use a statistical language model to find the highest ranked sentences to synthesize a code completion. Mou et al. [93] propose a tree-based convolutional neural network (TBCNN) on AST tree structure to detect code snippets of certain patterns. Holmes et al. [58] present an approach for locating relevant code that is based on heuristically matching the structure of the code under development to the example code. Saul et al. [117] use a random walk approach on a subset of a callgraph in order to recommend source code. Ekoko et al. [37] propose an approach that leverages the structural relationships between APIs to discover inaccessible API methods or types. McMillan et al. [86] recommend source code examples by querying against API calls and documentations about code structural information. Moritz et al. [92] present an approach to recommend API usage by representing software as a Relational Topic Model. Fowkes et al. [40] propose a probabilistic algorithm to find the most informative and parameter-free API call patterns.

2.4. Work Related to Predicting Software Licenses for Changed Source Code

2.4.1. Software license Identification and Classification

Techniques have been introduced to automatically identify and classify software licenses. Tuunanen et al. [127] proposed ASLA, a tool aimed at identifying licenses in FOSS systems. German et al. [46] proposed a tool called Ninka to identify license statements by inputting text files and output license names and versions using pattern matching approach. Di Penta et al. [33, 34] proposed approaches to automatically identify licenses of jar files via combined code search and textual analysis. Vendome et al. [129] applied a machine learning approach to detect exceptions of software license. Hoffmann et al. [57] analyzed actual license choice and correlated project growth from ten years of open source projects and discovered closed analytical models. Stewart et al. [123] found business-friendly open source licenses had a correlation with project success. Alspaugh et al. [7] developed a meta-model to analyze the
interaction of licenses from the viewpoint of software architecture. Mlouki et al. [88] investigated licenses violations and the evolution of these violations over time in Android ecosystem. German et al. [44] detected license inconsistencies in code clones between Linux and other OpenBSD and FreeBSD. While all the existing works developer to tackle the license identification problem, none of them proposed an approach or developed tool to predict software licenses at file-level changes.

2.4.2. License Adoption and Evolution

Di Penta et al. [34] studied the licensing evolution on six open sources systems and found that licenses version and type changed during software evolution. Manabe et al. [80] studied licenses on FreeBSD, OpenBSD, Eclipse, and ArgoUML evolution and discussed characteristics of license evolution. German and Hassan [45] built a model to investigate specific licenses about applicability, advantages and disadvantages. German et al. [43] conducted an empirical study on binary packages of the Fedora-12 Linux distribution to understand and audit licensing consistency between packages and source files and claimed that it was challenging to audit licensing issues. German et al. [44] also studied the cloned code fragments between the Linux Kernel and two distributions of BSD and concluded code migration is caused by additional restriction of software license. Wu et al. [138] found that license could be inconsistent among cloned files. Vendome et al. [130] conducted a survey with developers and found that facilitating commercial reuse was a common reason for license changes. They [128] further investigated open source projects to gain insights of causes of license migration. They found that licensing adoption and changes could be triggered by various factors. They also pointed out that there was a lack of traceability of when and why licensing changes were made. Almeida et al. [80] conducted a survey that posed development scenarios involving three popular open source licenses and found that developers struggled when multiple licenses were involved and developed a tool to recommend when adoption and evolution of license is needed. Sen et al. [118] explored factors that affected the choice of a license for a project through analysis of open source project artifacts. These studies discussed the challenges and importance
of predicting software licenses in software changes. However, none of the studies proposed a method or developed tools to predict licenses for software changes.
Chapter 3

TOWARDS AUTOMATED SOFTWARE CHANGE – SUPPORTING SOFTWARE ACTUAL CHANGE IMPACT SET PREDICTION

As we mentioned in Chapter 1, during software evolution, developers must ensure that dependent classes or methods are changed consistently [50]. Since developers find it hard to manually identify such dependent classes or methods, they benefit from automated support. We thus speak of the Change Impact Analysis (CIA) that guides developers [50] to make changes. It identifies set of classes or methods that repeatedly change together to ensure a consistent and complete change. This set is called the software actual change impact set. Traditional Change Impact Analysis, which aims at estimating the actual change impact set of a system due to a proposed change, is performed manually by developers. Due to the known complexity of the manual identification of actual change impact sets, developers would greatly benefit from automated predictions (a.k.a., automated CIA).

The state-of-the-art approaches of predicting the actual change impact set employ two main types of automated CIA approaches, namely dynamic and static analysis [4, 8, 18, 71, 101, 102, 145]. However, most of these techniques trace individual system execution paths and focus on method calls while ignoring data dependencies (e.g., fields or variables) shared across different execution paths. Static analysis [15, 48, 50, 51, 54, 64, 79, 84, 106, 143, 146] predicts actual change impact set by either (1) Mining code change repositories from previous changes in software repositories; or (2) Textual analysis through extracting conceptual dependencies (coupling) via the analysis of comments and/or identifiers in source code [71]. But as software evolves, older impact set changes could become outdated and possibly misleading. Besides, the assumption of actual change impact set follow the same patterns as documented in the history is not always true because software changes do not always impact the rest of the system following the same patterns. Meanwhile, textual analysis approaches
require developers to encode the implicit actual change impact set from the comments and/or identifiers and hence the quality of the change prediction depends on the quality of the encoding. To counter these weakness, another kind of static analysis method is proposed: structural analysis. Structural analysis approach leverages simply call dependencies among classes or methods (most notably method calls), program dependence graph (PDG) [24, 42] or employ program slicing [146] as indicators for the actual change impact set. Among them, PDG incorporates two kinds of code dependence other than call dependency: (1) control dependence representing the control flow relationships of the program and (2) data flow dependence representing the data flow relationships of the program. Program slicing [146], similar to PDG, addresses the computation of effects among program points by traversing data flow and control flow. It is interesting to note that very few approaches consider data sharing in predicting actual change impact sets. We believe data sharing is another important indicator for actual change impact set, which is overlooked by the state-of-the-art change impact analysis. Our working assumption is that class/method data sharing dependencies are a vital complement to call dependencies in fully understanding source code and how they are affected by changes.

To improve the state-of-the-art, we designed and implemented CH-ange Impact set Prediction (CHIP) framework. To demonstrate that more expressive data sharing dependencies are useful for the actual change impact set prediction, we compare our approach with PDG. In addition, we compare our approach with the state-of-the-art approach using evolutionary couplings extracted with association rule.

As a prerequisite, our approach requires an initial set of changes, which are the initial changes made by developers. This initial set of classes is likely a subset of classes that need changing because the developer may not yet understand the complete impact of the changes. The initial set of changes is thus a subset of all changes needed. Regardless, our approach will analyze call and data sharing dependencies on the initially changed class(es) to identify additional, dependent classes that likely need changing as well. Our approach then recommends these dependent classes that likely need changing as well. Our approach then recommends these dependent classes.
to the developer who decides on whether or not additional changes are necessary. Our approach then repeats the dependency analysis with every change iteration, thus refining the change prediction.

The main contributions of this chapter are:

- A dynamic mechanism to capture both call and data sharing dependencies across software classes and methods, which is then used to frame a novel basic CHIP system. Our experiment results show that code dependencies complement each other in the actual change impact set prediction. Furthermore, data sharing dependencies are particularly useful in specific change impact scenarios including “move refactoring”, “remove classes and statement”, “bug fixing”, “functional improvement” and ”code replacement”.

- A change impact set predictor with dependency frequency filter and a novel data type inverse document frequency (idf) filter to reduce false positives in prediction (i.e., false change impact sets) while maintaining the recall. We show that these filters significantly improve F2-score of CHIP compared to state-of-the-art approaches.

- An automated tool that provide developers with change impact set predictions.

### 3.1. Background & Motivation

To build the basis of this dissertation towards automated software change and source code suggestion, in this section we first discuss the studies on call dependency, PDG and evolutionary coupling. Then we present an example to motivate our CHange Impact set Prediction (CHIP) framework.

#### 3.1.1. Studies on Call Dependency

Call dependency are extracted by detecting method calls among classes and methods. Some previous approaches [9, 21, 66, 115, 115, 125, 135, 142] leverage call dependency among entities to support actual change impact set prediction. Sound static
analysis approaches for capturing call dependency can guarantee correctness through overestimation but typically produce a large number of false positives (wrong predictions) [139] and consume a significant amount of time or resources when the analysis is performed on the whole software system. To avoid such problem, CHIP uses dynamic analysis to capture actually observed call dependency. This eliminates false positives and a reasonable test coverage ensures a high degree of completeness. Next we describe how System Runtime Profiling is used to capture call dependencies.

(1) **System Runtime Profiling:** To capture high quality execution traces in a Java system, CHIP leverages a tool built on JVMTI (Java Virtual Machine Tool Interface) which was developed in our previous work [41]. This tool provides a way to inspect the state of the Java system and control its execution while the system runs on the Java Virtual Machine (JVM). This tool can query and record the special events that are generated by JVM including “method entry” and “method exit”. To ensure the correctness and completeness of captured dependencies, all functions according to requirements and use cases documents of each subject system must be executed with our instrumented runtime profiler. The execution traces are stored to be analyzed in the subsequent steps for dependencies generation.

(2) **Call Dependency Generator:** Note that CHIP’s call dependency generation approach can be generalized to any language platform in which execution traces can be captured. We use Java as an example to illustrate the process of capturing call dependencies. Call dependencies are captured by traversing all records in “method entry” and “method exit” records among the JVM events. This generator traces the events of each thread separately. Method X calls method Y if both classes are registered in the callback functions of “method entry” and “method exit”.
Table 3.1: Partial execution trace log of jEdit

<table>
<thead>
<tr>
<th>Order</th>
<th>Method</th>
<th>JVM Events</th>
<th>Thread ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>OptionsDialog.init()</td>
<td>method exit</td>
<td>27058272</td>
</tr>
<tr>
<td>2</td>
<td>StyleTableModel.getTableCellRender()</td>
<td>method entry</td>
<td>27058272</td>
</tr>
</tbody>
</table>

Figure 3.1: Venn diagram of class–level call dependencies in CHIP (CHIP-Call) versus class–level PDG in jEdit

3.1.2. Studies on Program Dependence Graph (PDG) and Evolutionary Couplings Capturing

In this section, we highlight the findings of previous studies examining PDG and evolutionary couplings described at the beginning of Chapter 3 and how to automatically generate them. Meanwhile, we outline the limitations of the change impact set prediction approaches using PDG and evolutionary couplings as well as the differences between call dependency and PDG.

3.1.2.1. Program Dependence Graph (PDG)

PDG [24, 42] incorporates two kinds of code dependence other than call dependency: (1) control dependence representing the control flow relationships of the program and (2) data flow dependence representing the data flow relationships of the program. However, it does not consider data sharing relations among classes and methods in predicting actual change impact set.

To generate PDG, a tool called “DUA-Forensics” leveraged in [116] is applied.
For point-to analysis, “DUA-Forensics” finds a set of classes or methods in locations pointed to by a known class or method variable in context- and flow- insensitive way. To ensure the correctness and completeness of dependencies, it exploits not only inter-procedural (i.e., across methods) but also intra-procedural dependencies such as exception control dependencies.

**Difference between call dependency and PDG:** The main difference between call dependency extracted by CHIP (CHIP-Call) and PDG is: PDG is extracted using static analysis approach, while CHIP-Call is extracted using dynamic analysis approach. PDG incorporates transitive calling queries by *define* and *use* the same value. For example, PDG considers a dependent relation between method A and method B if a value *defined* by A is *used* by B. However, PDG may not include method calls induced by aggregation of multiple JVM events analyzed by the JVM runtime profiling tool. To explain the difference between CHIP-Call and PDG, we use jEdit system as an example. Table 3.1 shows a partial execution trace log with three execution trace records captured by CHIP. In Table 3.1, we found a calling dependency between methods `OptionsDialog.init()` and `StyleTableModel.getTableCellRender()` induced by aggregation of two execution records in two different JVM events (e.g., method exit and method entry). Furthermore, Figure 3.1 illustrates a Venn diagram that shows the overlap and distinction between class–level CHIP-Call and class–level PDG. Both dependencies are captured from the jEdit system that we will discuss in Section 3.4. As we can see both approaches share 471 dependencies. However, CHIP-Call includes 1866 additional call dependencies not captured by PDG and PDG has 999 data/control flow dependencies not included in CHIP.

### 3.1.2.2. Evolutionary Coupling

Evolutionary coupling couples classes or methods co-changed together in code change history. Previous works [89, 90, 146] use association rules based mining on CVS logs for detecting evolutionary couplings among source code entities. However, if insufficient historical data is available (such as new project or project with incomplete repository), mining software repository based techniques are inapplicable. Besides,
there are changes that contradict the frequent impact set patterns.

To replicate previous works, in capturing evolutionary couplings we apply association rule based on itemset mined from historical change commits. We mine the SVN logs that were generated before the commits in test set. Details are shown in Table 3.6. We choose the same support value (=1) as in [146], to capture a more comprehensive set of evolutionary couplings.

3.1.3. Motivating Example

Figure 3.2 shows an example of an actual change impact set prediction on an
Table 3.2: Partial execution trace log of jHotDraw

<table>
<thead>
<tr>
<th>Methods</th>
<th>Accessed variable name</th>
<th>Accessed variable reference ID</th>
<th>JVM Events</th>
</tr>
</thead>
<tbody>
<tr>
<td>RoundRectangleFigure.createHandles()</td>
<td>-</td>
<td>-</td>
<td>method exit</td>
</tr>
<tr>
<td>RoundRectangleRadiusHandle.init()</td>
<td>-</td>
<td>-</td>
<td>method entry</td>
</tr>
<tr>
<td>RoundRectangleFigure.setArc()</td>
<td>roundrect</td>
<td>19330724</td>
<td>-</td>
</tr>
<tr>
<td>AbstractAttributedFigure.setAttribute()</td>
<td>attributes</td>
<td>19330724</td>
<td>-</td>
</tr>
</tbody>
</table>

early version jHotDraw following the CIA process. A developer starts to change the code based on a change request (i.e, an item that can be either a bug or a request for enhancement [25]). This change request starts with a short description: More options on adjusting the width and height properties of the arc of round rectangle figure. The developer first changes the class RoundRectangleFigure as it seems an obvious choice. However, the developer does not know whether the change is limited to this class or also propagated to other dependent classes that require changes as well. Since this is an early version generated at the early lifecycle of the software development, the developer doesn’t have the access to sufficient historical data either. Table 3.2 shows the execution traces of three classes (RoundRectangleFigure, RoundRectangleRadiusHandle and AbstractAttributedFigure) in jHotDraw captured by our system runtime profiling tool. If we merely consider call dependencies and PDG, the system would predict that class RoundRectangleRadiusHandle would be the only affected class because in Figure 3.3 and Table 3.2 it is obvious that the constructor method (i.e., init()) in class RoundRectangleRadiusHandle is called in
method `createHandles()` of class `RoundRectangleFigure`. According to [23], data dependencies in PDG are extracted by transitive calling of method queries by value define and use. For example, if method M0 is required to be changed. Since M0 calls M1 and M4, and also transitively M2, M3 (both via M1), and M5 (further via M2), the change impact set will not only include M0 but also M1, M4, M2, M3, and M5. Class `AbstractAttributedFigure` would not be predicted as in impact set since there is neither a call dependency nor data dependencies captured by PDG between classes `RoundRectangleFigure` and `AbstractAttributedFigure`. However, in the version control history of jHotDraw, we find that class `AbstractAttributedFigure`, often changed together with `RoundRectangleFigure`, is also listed in the actual change impact set submitted by developer in the commit history. The reason is that there was data sharing between `RoundRectangleFigure` and `AbstractAttributedFigure`. The difference between the data sharing dependencies and data dependencies in PDGs is that data sharing dependencies can occur by aggregation of multiple JVM events, no matter whether they are transitive or not. For example, in the code snippet shown in Figure 3.3, there is a data sharing dependency between `RoundRectangleFigure` and `AbstractAttributedFigure` since the field `roundrect` accessed in `RoundRectangleFigure.setArc()` is eventually accessed by `AbstractAttributedFigure.setAttribute()` as Object value in the `HashMap` typed field `attributes`. Both fields (i.e., `roundrect` and `attributes`) use the same data indicated by sharing the same reference ID (19330724) recorded in the execution trace. This execution trace record is shown in Table 3.2. It is relatively harder to manually observe that in code snippet that there is a data sharing occurs between classes `RoundRectangleFigure` and `AbstractAttributedFigure` because a field `roundrect` accessed in `RoundRectangleFigure.setArc()` is also accessed in `RoundRectangleRadiusHandle.setAttribute()` as a field defined as `attributes`. Fields `roundrect` and `attributes` point to the same piece of data in the memory identified by reference ID 19330724, even though they are declared different variable names. This example tells us actual change impact set prediction probably depends not only on call, data and control flow relations between classes but also their data sharing relations. Therefore, we will investigate whether and how much data sharing dependencies can
support software actual change impact set prediction as a complement to call and control flow dependencies.

### 3.2. Fundamentals of Data Sharing Dependency

To provide sufficient context to the reader, in this section we introduce the fundamental concept of data sharing dependency as well as how to capture data sharing dependency.

#### 3.2.1. Definitions of Data Sharing Dependency

A method-level data sharing dependency is defined as 2 methods reading or manipulating variables that point to the same data stored in the same (physical) memory location no matter whether the variables holding the pointers are identical or not. A class-level data sharing dependency is the aggregation of method-level dependencies. Both kinds of data sharing dependencies are useful, since shared data are often accessed through references or chains of references, unlike other code structural dependencies (ie, call dependencies and control dependencies), data sharing dependencies can only be captured through runtime profiling analysis.

#### 3.2.2. Capturing Data Sharing Dependency

We capture method-level data sharing dependencies via dynamic analysis from execution traces. Since data sharing dependencies are captured between two methods, class-level data sharing dependencies are simply the aggregation of method-level dependencies between two classes. This step is realized by two components in the CHIP framework: (a) system runtime profiling (Section 3.1.1); and (b) data sharing dependency generator (discussed next).

Similar to call dependencies, data sharing dependencies can be automatically generated via analyzing the events in the execution trace database. Among all JVM events, we find that “method entry”, “method exit”, “field access” and “field modification” are the ones closely related to data sharing relations. We have developed our own data sharing dependency generation method and tool to capture
class-level data sharing dependencies. This implementation is currently limited to Java systems but should be easily extensible to other languages. Our tool incorporates four variants of data sharing dependency generator for analyzing different JVM events: “field access” events, “field modification” events, “method entry” and “method exit” events, as well as the situation of crossing over different events. The algorithms for capturing method-level data sharing dependencies are available at https://www.dropbox.com/s/wp9pfceptubw9g7p/DataDep.pdf?dl=0.

3.2.2.1. Field Access

Data sharing dependency in a “field access” (FA) event is captured when two classes access the same field object (variables sharing the same reference ID) through a “field access” event. In JVM each object is assigned with a unique hash code as an identifier (note: this is different from the objects’ non-unique hash code). If different parameters refer to the same object, then they are assigned with the identical identifier.

3.2.3. Field Modification

For data sharing dependency in “field modification” (FM), the process is almost the same as in FA except that this object is modified in “field modification” event. For the object being modified, its unique identifier shall be updated by a new one.

3.2.4. Parameter Passing

Data sharing dependency for Parameter Passing (PP) is captured through “method entry” and “method exit” events. Both events provide a way to inspect variables (including parameters and return value) created or received by a method. Similarly, if the objects share the same unique identifier, they are thus identical. The return value is differentiated by its attribute “CurMReturnValue” as a unique identifier. The class that initially accesses this return value is the receiver.

3.2.5. Cross Events
In this case, data sharing dependency is captured when it is observed across any of the four different JVM events: “field access”, “field modification”, “method entry” and “method exit”. If affected data are identical based on their unique identifiers, both classes are considered data sharing dependent on each other even though they may reach objects in different JVM events.

Figure 3.4 shows seven examples involving the data sharing dependencies captured in four different situations in jHotDraw. Based on Figure 3.4 data sharing dependency generator constructs a data sharing dependency set as shown in Figure 3.5 simply by adding the dependencies as lines between the classes. Specifically, a data sharing dependency link between two classes is added when any method in each class accesses, modifies, or sends/receives the same piece of data as identified by its unique identifier.

3.3. Design of CHIP

3.3.1. Usage Scenario

Let us consider the following scenario: A developer starts by changing identified initial set of classes. No automation supports this first step. Automation comes into play to find out if the developer needs to change other dependent classes (the actual change impact set), and to prevent changes like relocating methods inside
Figure 3.5: An example data sharing dependency set generated from Figure 3.4

a class from affecting the actual change impact set prediction. For this, we are only interested in analyzing and making use of class–level dependencies to predict class–level actual change impact set. Specifically, our approach leverages not only existing code dependencies such as call dependencies but also dependencies based on data sharing, which enhances the existing data flow and control dependencies. Further, we find that the proposed CHange Impact set Prediction (CHIP) framework is particularly useful in predicting actual change impact set in the following commonly observed change activities [61]. We speak of actual impact scenarios:

- **“Moving Refactoring”**: A piece of code is moved from one class to another class.
- **“Remove Class or Statement”**: A class or statement in a class is removed.
- **“Bug Fixing”**: Initial changes are made to fix bugs.
- **“Functional Improvement”**: Initial changes are made to add new features for the system.
- **“Code Replacement”**: A piece of code is replaced.

3.3.2. CHIP Framework
Figure 3.6: Overview of software actual Change Impact Set Prediction (CHIP) framework

Figure 3.6 provides an overview of CHIP framework to automatically predict the actual change impact set in source code with an initial set of changes made by developers. In particular, we will explore whether dependencies based on data sharing captured by our data sharing dependency generator can complement call dependencies in actual change impact set prediction. A supporting tool for all components in the framework has been implemented.

Our framework leverages both call and data sharing dependencies. Figure 3.6 shows the overview of CHIP. The preparation of dependencies is implemented in the “Preprocessing” component (B). In this component, we capture all execution data through an execution trace profiling tool (B-1) and extract both call and data sharing dependencies using the corresponding algorithms (B-2). The “Change Impact set Prediction” component automates the prediction of actual change impact set (C-1) and dependency frequency filtering (C-2) and shared data type \( \text{idf} \) filtering (C-3) with thresholds, in which both thresholds is selected by applying adaptive learning approach. In this step, CHIP makes actual change impact set prediction based on various combinations of code dependencies.

In general, CHIP takes two inputs: (1) a set of methods/classes that the developer
already made the change; and (2) The call and data sharing dependencies captured using the approaches described in Sections 3.1 and 3.2. We have developed two variants of the predictor: (1) basic actual change impact set predictor, (2) two augmented basic predictors with extensions, one with dependency frequency filter and the other with a data type inverse document frequency (idf) filter in order to improve the precision of the basic predictor, shown as Steps C-1, C-2 and C-3 in Figure 3.6.

3.3.3. Basic Actual Change Impact Set Predictor

The basic predictor can predict the actual change impact set at both method and class levels based on the call and/or data sharing dependencies in code. We have developed four variants of the basic predictor based on: (1) call dependencies only (P_c); (2) data sharing dependencies only (P_d); (3) both call and data sharing dependencies (P_cd); (4) PDG (P_pdg); and (5) evolutionary couplings (P_e). P_pdg and P_e are included for comparison with the other CHIP variants (P_d, P_cd) based on data sharing and/or call dependencies. Meanwhile at both method and class level, we consider the effect of the order of changes. When developers make a set of relevant changes in order, we believe that the very first change he/she makes have a leading impact on the subsequent changes in the actual change impact set, while the impacts of subsequent changes are diminishing. Therefore, in our algorithm, we assign a higher weight on the very first change and start actual change impact set predictions based on code dependencies with the very first change. Then for other methods/classes after the very first change, only shared predictions based on a pair of methods/classes are included in the predicted change impact set. Equation (3.1) formalizes the computation of the prediction taking the order of changes into account:

$$P = A_1 + \sum_{i,j=2}^{n} A_i \cap A_j$$  \hspace{1cm} (3.1)

A_1 denotes the prediction made based on the code dependencies with the very first change in an actual change impact set. A_i and A_j are prediction set of ith/jth changed method/class in the given set of changes of size n and A_i \neq A_j.

However, which was the very first one in the actual change impact set from the
commit history is enclosed and unknown to others except developers themselves. Therefore, in our experiment, to determine the first changed method/class in a given change set, we greedily run through all possible first changed method/class.

**Method–level Prediction:** The algorithm predicts the methods that need to be changed together with the method being changed by developer. The following example explains how the basic predictor predicts actual change impact set at method level. The change impact analysis process starts with an initial set of method–level changes that are made by the developer. With an initial set of changes \{m1, m2, m3\}, if m1 is the first method being changed by developer, the predictor \(P_c\) or \(P_d\) will search the call or data sharing dependencies to determine which method has the call or data sharing dependency on m1. If \(P_c\) detects that m4 has call dependency with m1, it will predict m4 as in the actual change impact set. Then for m2 and m3, because of order of changes effect, only their shared predications are considered. For instance, if \(P_c\) detects that m5 has call dependencies on both m2 and m3, it will predict m5 in the actual change impact set. So the actual change impact set of \{m1, m2, m3\} are \{m4, m5\}. Similarly, \(P_d\) detects that m6 has data sharing dependency on m1 and m7 has data sharing dependencies on both m2 and m3, it will predict \{m6, m7\} as the change impact set. \(P_{cd}\) make prediction based on both call and data sharing dependencies. For the aforementioned two examples, \(P_{cd}\) will predict \{m4, m5, m6, m7\} as the change impact set. In method–level prediction, besides direct dependencies, two-step transitive dependencies are also considered taken into account because many changes are caused by two-step transitive impact of initial changes. For instance, if \{m1, m2, m3\} are given methods being changed, \{m4, m5, m6, m7\} are method–level change impact set based on direct dependencies with the given change set. If m8 is detected to have call/data sharing dependency with m4, m8 will also be predicted in the change impact set since m8 has two-step transitive dependency with given changed method via m4.

**Class–level prediction:** The class–level actual change impact set can be derived from the method–level change impact set prediction results. It is a simple aggregation of the method–level results by their owner classes. For example, with the same initial
set of method–level changes \{m1, m2, m3\}, \(P_{cd}\) will predict \{m4, m5, m6, m7\}. Assume m1 belongs to class c1, m2 and m3 belong to class c2, m4 and m5 belong to class c3, m6 and m7 belong to c4 and c5 respectively. We can aggregate the method–level results and make the change impact set prediction at the class level accordingly. If classes \{c1, c2\} are the given set of changes being made by developer, \(P_{cd}\) will predict its actual change impact set as \{c3, c4, c5\}. Algorithm 1 illustrates how the class–level change impact set is predicted.

**Example:** Figure 3.7 shows a portion of combined call and data sharing dependencies from jHotdraw, based on which the basic predictor \(P_{cd}\) is constructed to predict the actual change impact set at the class level based on both call and data sharing dependencies. The solid links represent call dependencies and dashed links represent data sharing dependencies. Based on Figure 3.7 the basic predictor \(P_{cd}\) is able to pre-
Figure 3.7: An example of comprehensive set of dependencies with call dependencies (solid links) and data sharing dependencies (dash links) for the basic predictor.

Table 3.3: Predicted Actual Change Impact Set with Class *RoundRectangleFigure* by Basic Predictor

<table>
<thead>
<tr>
<th></th>
<th>RoundRectangleFigure</th>
<th>Call</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>RoundRectangleRadiusHandle</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>2</td>
<td>BoundsOutlineHandle</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>3</td>
<td>ResizeHandleKit</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>4</td>
<td>AbstractFigure</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>5</td>
<td>AbstractAttributedFigure</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>6</td>
<td>TextAreaFigure</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>7</td>
<td>ConnectionTool</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>8</td>
<td>LineConnectionFigure</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>9</td>
<td>DiamondFigure</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>10</td>
<td>TriangleFigure</td>
<td>N</td>
<td>Y</td>
</tr>
</tbody>
</table>
dict the actual change impact set with class \textit{RoundRectangleFigure}, which are listed in Table 3.3. “Y” indicates a dependency between two classes while “N” indicates no dependency is detected. In this example, three classes \textit{BoundsOutlineHandle}, \textit{ResizeHandleKit} and \textit{AbstractFigure} are predicted in the actual change impact set based on call dependencies, while six classes \textit{AbstractAttributedFigure}, \textit{TextAreaFigure}, \textit{ConnectionTool}, \textit{LineConnectionFigure}, \textit{DiamondFigure} and \textit{TriangleFigure} can only be predicted through data sharing dependencies. The class \textit{RoundRectangleRadiusHandle} is predicted by both call and data sharing dependencies.

3.3.4. Extension I: Dependency Frequency Filter

Nevertheless we notice that both call and data sharing dependencies introduce heavy noise (false positives), which in turn compromises the precision of prediction. One reason is that the methods/classes which have less dependencies on the given change set are treated equally important with the ones which have more dependencies on the given change set. To resolve this issue, we introduce a novel Dependency Frequency Filter extension to the basic predictor using combined call and data sharing dependencies ($P_{cd}$). This extension is devised based on how frequent a method/class is predicted with the given change set. Here is an example. We start from predicting the actual change impact set at the method–level, with an initial set of methods \{m1, m2, m3\} being changed (i.e., given change set) and the dependency frequency filter threshold set as 2, if $P_{cd}$ detects that m4 only has one data sharing dependency with m1, while m5 has totally three call and/or data sharing dependencies with m2 and m3, then m5 will be predicted in the actual change impact set of \{m1, m2, m3\} rather than m4 since m4 only has one dependency link with the given change set, which is below the dependency frequency filter threshold 2. Next, we will make actual change impact set prediction at the class level with dependency frequency filter threshold as 2 as well. Assume m1 belongs to class c1 and both m2 and m3 belong to class c2. If m4 is owned by class c3, then c3 will have one data sharing dependency with c1. If m5 is owned by class c4, c4 will have six call and/or data sharing dependencies with c2. In this case, c3 will be filtered out and not be predicted in the actual change
Table 3.4: Top 5 Objects with the Lowest $idf$

<table>
<thead>
<tr>
<th>Object Combination</th>
<th>Occur</th>
<th>$idf$</th>
<th>Normalized $idf$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 java.lang.String</td>
<td>73024</td>
<td>0.23</td>
<td>0.0</td>
</tr>
<tr>
<td>2 java.sql.ResultSet</td>
<td>51178</td>
<td>0.59</td>
<td>0.032</td>
</tr>
<tr>
<td>3 java.lang.Class</td>
<td>45030</td>
<td>0.72</td>
<td>0.043</td>
</tr>
<tr>
<td>4 java.util.List</td>
<td>41469</td>
<td>0.80</td>
<td>0.051</td>
</tr>
<tr>
<td>5 edu.ncsu.csc.itrust.beans.-PersonnelBean</td>
<td>38645</td>
<td>0.87</td>
<td>0.057</td>
</tr>
</tbody>
</table>

impact set because it only has one dependency link with the given change set \{c1, c2\}, which is below the dependency frequency filter threshold of 2.

Here we employ adaptive learning to determine the value of the threshold. First, we randomly split the commits of each system five folds. We reserve one fold as an evaluation set and use the other four folds as training sets. We then train on the training set and test the learnt threshold on the held-out evaluation set. We compute the F2-score—a measure that combines precision (fraction of actual changes in prediction) and recall (fraction of actual changes that are predicted) of the prediction results of evaluation set. If a better F2-score is achieved, the threshold is adjusted. This process is repeated. Finally, the optimal threshold corresponding to the highest value of F2-score is chosen.

3.3.5. Extension II: Inverse Data Frequency ($idf$) Filter

We also notice that data sharing dependencies themselves introduce heavy noise (false positives) which also compromises the precision of predictions. The reason for this kind of noise is that all shared data types are considered equally important for predicting the actual change impact set. To tackle this problem, we extend the basic predictor relying on combined call and data sharing dependencies ($P_{cd}$) with a novel shared data type inverse data frequency ($idf$) filter.
For a data sharing dependency across classes, methods in two distinct classes must share data. This data sharing may involve one or multiple variables or parameters and may cover multiple data types. However, not all the data types provide equally useful implication for the actual change impact set. The column “Occur” in Table 3.4 shows how often a data type occurs in all data sharing dependencies generated from the iTrust system. The maximum occurrence of a data type can be the total number of data sharing dependencies (For iTrust, the total number of data sharing dependencies is 92285.) meaning that a data type is shared in every dependency. The minimum occurrence is one meaning that this data type is only shared once. For example, data type `java.lang.String` is shared by classes much more frequently than other data types. A reasonable conjecture is that `java.lang.String` is a commonly shared data type to pass string data across many methods in the iTrust system, which means that this kind of data types is thus too “general”. If a data sharing dependency between two classes is upon a number of “general” data types, this data sharing dependency is probably too “general” to imply an actual change impact set in practice, which should be excluded from our actual change impact set prediction results.

Hence, we borrow the idea of Inverse Document Frequency [13] from information retrieval to define our Inverse-Data-Frequency to weigh the importance of each data type for actual change impact set prediction. Inverse Data Frequency (idf) is the measure of occurrence of a data type across all data sharing dependencies in a system. Specifically, it is defined as:

\[
idf = \log\left(\frac{N}{n_d}\right)
\]  

(3.2)

where N is the total amount of data sharing dependencies and n_d is the occurrence of a data type across all data sharing dependencies. For the purpose of generalizing the idf across different systems, we normalize all the idfs in a system as:

\[
idf_{\text{norm}} = \frac{idf - idf_{\text{min}}}{idf_{\text{max}} - idf_{\text{min}}}
\]  

(3.3)

idf_{\text{min}} and idf_{\text{max}} denote the lowest and highest idf in a system, respectively. The normalization ensures that idf_{\text{norm}} falls between 0 and 1. Since each data sharing
dependency could have more than one shared data type, the *idf* for each data sharing dependency is calculated as \( idf_{\text{accum}} \), which is normalized accumulation of \( idf_{\text{norm}} \) of all shared data types in each data sharing dependency. In principle, *idf* will value rare data types higher than common data types (e.g., `java.lang.String`). The threshold for \( idf_{\text{accum}} \) on each system is selected using adaptive learning similar to Extension I (Dependency Frequency Filter) to determine which data sharing dependences are too “general” to be included in the actual change impact set prediction.

Figure 3.8 shows the trimmed call and data sharing dependencies after applying the *idf* filter on the call and data sharing dependencies of jHotDraw in Figure 3.7. Table 3.5 lists the predicted actual change impact set with class `RoundRectangleFigure` by CHIP with *idf* extension. In this example, there are multiple method–level data sharing dependencies between classes `ConnectionTool` and `RoundRectangleFigure`. If we set the threshold of *idf* as 0.03, `ConnectionTool` is eliminated since among all method–level data sharing dependencies between `ConnectionTool` and `RoundRectangleFigure` the highest *idf* falls below 0.03. Besides, although `RoundRectangleFigure` and `RoundRectangleRadiusHandle` are linked by both call and data sharing dependencies, `RoundRectangleRadiusHandle` is still eliminated because the highest *idf* of data sharing dependencies between `RoundRectangleFigure` and `RoundRectangleRa-
diusHandle` is also below the *idf* threshold.
Table 3.5: Predicted Actual Change Impact Set with Class `RoundRectangleFigure` by Basic Predictor + `idf` Extension

<table>
<thead>
<tr>
<th><code>RoundRectangleFigure</code></th>
<th>Call</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>BoundsOutlineHandle</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>ResizeHandleKit</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>AbstractFigure</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>AbstractAttributedFigure</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>TextAreaFigure</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>LineConnectionFigure</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>DiamondFigure</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>TriangleFigure</td>
<td>N</td>
<td>Y</td>
</tr>
</tbody>
</table>

3.4. Empirical Evaluation

3.4.1. Experiment Design

Our experiments use CHIP to predict actual change impact sets in source code with an initial set of changes. To investigate the effects of data sharing dependencies on actual change impact set prediction, we compare the performance among the CHIP variants including the predictors based on call dependencies (P_c), data sharing dependencies (P_d), combined call and data sharing dependencies (P_cd) and P_cd with frequency and `idf` extensions (P_cd+ext). We also compare the performance of CHIP variants built on data sharing dependencies and the combined call and data sharing dependencies (P_d, P_cd) with PDG (P_pdg) and evolutionary couplings (P_e).

3.4.1.1. Datasets

Our experiments are conducted on four open source Java systems, which have been developed and evolved for a total of over 40 years by hundreds of developers all
Table 3.6: Characteristics of Four Open Source Systems

<table>
<thead>
<tr>
<th>System Version</th>
<th>iTrust 13.0</th>
<th>GanttProject 2.0.9</th>
<th>jHotDraw 7.2</th>
<th>jEdit 4.3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size (KLOC)</td>
<td>43</td>
<td>45</td>
<td>72</td>
<td>109</td>
</tr>
<tr>
<td># of classes</td>
<td>461</td>
<td>475</td>
<td>546</td>
<td>503</td>
</tr>
<tr>
<td>Evaluated commits</td>
<td>c216–c256</td>
<td>Most recent 76 commits to 2.0.9</td>
<td>c518–c798</td>
<td>c7998–c8340</td>
</tr>
<tr>
<td>Mined SVN logs for evolutionary couplings</td>
<td>Since 2009-08-18</td>
<td>Since 2010-12-08</td>
<td>Since 2004-02-01</td>
<td>Since 2006-09-17</td>
</tr>
<tr>
<td># of call dependencies</td>
<td>5954</td>
<td>5055</td>
<td>4550</td>
<td>6463</td>
</tr>
<tr>
<td># of data sharing dependencies</td>
<td>92285</td>
<td>108779</td>
<td>112531</td>
<td>137370</td>
</tr>
</tbody>
</table>

over the world: jEdit 4.3, a mature programmer’s text editor; iTrust 13.0, a medical management system; jHotDraw 7.2, a Java GUI framework for graphics; GanttProject 2.0.9, a cross-platform project scheduling and management system. The choice of these four systems is motivated by the need of: (1) historical change commits (i.e., gold sets of change commits) for evaluation; (2) systems belonging to different problem domain; (3) systems of different sizes that are neither too small nor too large to allow developers to assess dependencies among methods/classes of an entire system; and (4) possibility of capturing high quality execution trace by runtime profiling. Table 3.6 summarizes the characteristics of the four systems.
In class-level prediction, for a commit of size $n$, which contains $n$ changed classes, we randomly pick $i$ classes ($1 \leq i \leq n$) from the commit as the initial set of changes. Similarly in method-level, we randomly pick $i$ methods ($1 \leq i \leq n$) from the commit with $n$ changed methods as the initial set of changes. Then we ask CHIP to predict the $n-i$ actual change impact set in the commit. We exhaustively explore all combinations of size $i$ before increasing the size of the initial change set by 1 and repeat the process. We permute all subsets in a commit as the given change sets greedily since developers could start by changing any subset for a change task. Hence, our evaluation tests all possible scenarios how changes may unfold. Meanwhile, since the order of changes are taken into consideration, we present “best case” defined as the change order with the most optimal recall achieved. “Best case” imitates the actual scenario that developer makes the change. And “overall” permutes all possible scenarios of changes by summarizing results of all possible orders of changes.

Data Preprocessing: The evaluation excludes the testing code (e.g., jUnit tests) because change impact between testing code and system code are usually expected and for this reason are less important [120]. Furthermore, we do not consider changed classes in the commit if the changes are merely: (1) independent formatting changes (e.g., removing an empty line), or (2) API changes without reporting the changed classes. Although many existing impact set prediction studies exclude merging commits or large classes, our study incorporates them all for a more comprehensive study purpose.

To investigate the effects of data sharing dependencies on predicting various change impact scenarios, we classify the commits in four systems based on their change activities documented in commit messages into: “move refactoring”, “remove class or statement”, “bug fixing”, “functional improvement” and “code replacement”, which dominate in those systems.

3.4.1.2. Evaluation Metrics

We measure precision ($P$) recall ($R$) and F2-score of actual change impact set prediction. Here F2-score instead of F-score is used because recall is more important
than precision in actual change impact set prediction. We here use an example to explain how the pair of metrics are measured for an individual prediction experiment. For a commit C containing a set of changed classes \{c1, c2, c3\} and a starting set of changes \{c1\} which is known being changed by developer, if the predicted change impact set are classes \{c1, c2, c4, c5\}, then c1 and c2 are “true positive”, c4 and c5 are “false positive”, c3 is “false negative”. Thus recall is measured as 2/3=66.7%, precision is 2/4=50% and F2-score is \(5*66.7%*50%/(66.7%+4*50%)=62.5\%\). To measure the overall prediction accuracy for an entire system, we sum up the “true positive” \((TP_{total})\), “false negative” \((FN_{total})\), and “false positive” \((FP_{total})\) from all the prediction experiments on individual commits in a system and calculate the overall recall, precision and F2-score as follows:

\[
threshold = \frac{TP_{total}}{TP_{total} + FN_{total}} \quad (3.4)
\]

\[
precision_{total} = \frac{TP_{total}}{TP_{total} + FP_{total}} \quad (3.5)
\]

\[
f2 − score_{total} = \frac{5\times recall_{total} \times precision_{total}}{recall_{total} + 4\times precision_{total}} \quad (3.6)
\]

### 3.4.1.3. Time efficiency Compared to PDG

Table 3.7 shows the time efficiency of CHIP built on data sharing dependencies compared to PDG. Experiments were performed on a computer with Intel Core i5
2.8GHz (configured with one thread and 8GB RAM). In terms of the dependency extraction time, as we expected, extracting all four types of data sharing dependencies costs more time than PDG since data sharing dependencies contains much finer-grained information than the PDG. In all cases, the time of extracting data sharing dependencies from execution traces generated in software testing phase are within affordable 82 minutes. Compared to PDG, it takes almost the same amount of time for CHIP to make each prediction of actual change impact sets.

3.4.2. Experimental Results

This section empirically answers our research questions.

**RQ1.** Do data sharing dependencies complement call dependencies in actual change impact set prediction?

**Results:** The results of performance metrics of call, data sharing dependencies and combined call and data sharing dependencies on the four systems are shown in Table 3.8 (class-level) and Table 3.9 (method-level). Performance metrics are measured in two modes: *best case* and *overall*. As described at the beginning of Chapter 3, since the order of changes are considered. The order is known by developers who made those changes, but it is unknown to researchers. *Best case* is the most likely change order according to the most optimal recall. *Overall* is the measure of all possible orders. Comparing the prediction results of $P_{cd}$ with $P_c$, in class-level predictions recall of $P_{cd}$ outperforms $P_c$ by 3.1%-17.4% overall and 3.4%-7.4% in best case over the four systems. In method-level predictions, recall improves by 3.4%-32.1% overall and 3.7%-7.1% in best case by $P_{cd}$ compared to $P_c$. Adding extension I (Dependency Frequency filter) and extension II (idf filter) to $P_{cd}$, prediction precision is also greatly improved. Table 3.10 shows that for class-level predictions in iTrust, GanttProject, jHotDraw and jEdit false positives are reduced by 97.2%, 93.6%, 72.0% and 85% using $P_{cd}$+ext compared to $P_{cd}$, while true positives are only compromised by only 3.4%, 8.9%, 5.6% and 5.8% correspondingly. Comparing the prediction results of $P_{cd}$ with $P_c$, Table 3.8 and Table 3.9 show that F2-score of $P_{cd}$+ext outperforms $P_c$ by 3.1%-46.3% overall and as much as 35.8% in best case over four systems. In class-
Table 3.8: Four Systems at Class-level: Precision (P(%)), Recall (R(%)) and F2-score (F2(%)) by P_{cd+ext}, P_{cd}, P_{d}, P_{c}, P_{pdg} and P_{e}

<table>
<thead>
<tr>
<th>Systems</th>
<th>Systems</th>
<th>Best Case</th>
<th>Overall</th>
<th>Overall-Single</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>R   P   F2</td>
<td>R   P   F2</td>
<td>R   P   F2</td>
</tr>
<tr>
<td>iTrust</td>
<td>P_{cd+ext}</td>
<td>96.6 40.0 <strong>75.3</strong></td>
<td>90.6 40.7 <strong>72.7</strong></td>
<td>82.8 25.0 <strong>56.6</strong></td>
</tr>
<tr>
<td></td>
<td>P_{cd}</td>
<td>100 1.9  8.8</td>
<td>100 2.4  11.0</td>
<td>100 1.7  7.8</td>
</tr>
<tr>
<td></td>
<td>P_{d}</td>
<td>100 2.1  9.5</td>
<td>100 2.5  11.5</td>
<td>100 1.8  8.2</td>
</tr>
<tr>
<td></td>
<td>P_{c}</td>
<td>96.6 10.7 37.0</td>
<td>82.6 11.5 36.9</td>
<td>72.4 5.9  22.2</td>
</tr>
<tr>
<td></td>
<td>P_{pdg}</td>
<td>96.6 7.9  29.9</td>
<td>82.6 7.1  26.5</td>
<td>72.4 4.6  18.4</td>
</tr>
<tr>
<td></td>
<td>P_{e}</td>
<td>89.7 9.7  33.8</td>
<td>81.2 7.4  27.0</td>
<td>65.5 7.3  25.2</td>
</tr>
<tr>
<td>GanttProject</td>
<td>P_{cd+ext}</td>
<td>90.6 53.5 <strong>79.6</strong></td>
<td>80.5 19.3 <strong>49.3</strong></td>
<td>60.1 10.0 <strong>29.1</strong></td>
</tr>
<tr>
<td></td>
<td>P_{cd}</td>
<td>99.5 7.5  29.0</td>
<td>91.1 5.3  21.6</td>
<td>77.3 3.5  14.9</td>
</tr>
<tr>
<td></td>
<td>P_{d}</td>
<td>99.0 7.6  29.2</td>
<td>89.4 5.5  22.0</td>
<td>74.4 3.5  14.7</td>
</tr>
<tr>
<td></td>
<td>P_{c}</td>
<td>92.1 23.3 57.9</td>
<td>76.2 18.6 47.0</td>
<td>45.8 11.7 28.9</td>
</tr>
<tr>
<td></td>
<td>P_{pdg}</td>
<td>97.0 7.1  27.5</td>
<td>93.0 5.6  22.5</td>
<td>58.6 5.9  21.0</td>
</tr>
<tr>
<td></td>
<td>P_{e}</td>
<td>85.2 23.5 55.9</td>
<td>63.8 22.4 46.6</td>
<td>30.0 9.4  20.1</td>
</tr>
<tr>
<td>jHotDrw</td>
<td>P_{cd+ext}</td>
<td>93.1 22.6 57.4</td>
<td>75.4 11.1 34.9</td>
<td>50.7 9.5  <strong>27.1</strong></td>
</tr>
<tr>
<td></td>
<td>P_{cd}</td>
<td>98.6 8.0  30.1</td>
<td>78.7 8.5  29.7</td>
<td>66.4 4.4  17.4</td>
</tr>
<tr>
<td></td>
<td>P_{d}</td>
<td>96.2 7.8  29.4</td>
<td>76.7 8.5  29.4</td>
<td>61.9 4.2  16.5</td>
</tr>
<tr>
<td></td>
<td>P_{c}</td>
<td>93.1 21.4 55.7</td>
<td>65.8 25.6 50.1</td>
<td>46.0 12.5 30.0</td>
</tr>
<tr>
<td></td>
<td>P_{pdg}</td>
<td>96.3 9.5  34.0</td>
<td>72.4 9.8  31.7</td>
<td>50.9 6.1  20.7</td>
</tr>
<tr>
<td></td>
<td>P_{e}</td>
<td>69.7 48.0 <strong>63.9</strong></td>
<td>57.8 57.6 <strong>57.8</strong></td>
<td>25.5 22.0 24.7</td>
</tr>
<tr>
<td>jEdit</td>
<td>P_{cd+ext}</td>
<td>93.3 24.0 59.2</td>
<td>87.7 13.8 42.3</td>
<td>77.1 12.4 37.8</td>
</tr>
<tr>
<td></td>
<td>P_{cd}</td>
<td>99.0 4.8  20.2</td>
<td>92.5 3.4  14.9</td>
<td>90.5 3.0  13.1</td>
</tr>
<tr>
<td></td>
<td>P_{d}</td>
<td>94.3 4.4  18.6</td>
<td>84.1 3.5  15.1</td>
<td>77.1 2.8  12.3</td>
</tr>
<tr>
<td></td>
<td>P_{c}</td>
<td>95.2 14.0 44.1</td>
<td>89.4 7.2  27.3</td>
<td>82.9 6.8  25.5</td>
</tr>
<tr>
<td></td>
<td>P_{pdg}</td>
<td>91.4 10.7 36.5</td>
<td>87.2 7.0  26.6</td>
<td>73.3 6.3  23.5</td>
</tr>
<tr>
<td></td>
<td>P_{e}</td>
<td>78.1 55.4 <strong>72.2</strong></td>
<td>70.9 43.7 <strong>63.1</strong></td>
<td>46.7 34.5 <strong>43.6</strong></td>
</tr>
</tbody>
</table>
Table 3.9: Four Systems at Method-level: Precision (P(%)), Recall (R(%)) and F2-score (F2(%)) by $P_{cd+ext}$, $P_{cd}$, $P_d$, $P_c$, $P_{pdg}$ and $P_e$

<table>
<thead>
<tr>
<th>Systems</th>
<th>Systems</th>
<th>Best Case</th>
<th>Overall</th>
<th>Overall-Single</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>R  P  F2</td>
<td>R  P  F2</td>
<td>R  P  F2</td>
</tr>
<tr>
<td><strong>iTrust</strong></td>
<td>$P_{cd+ext}$</td>
<td>96.3 100 <strong>97.0</strong></td>
<td>89.2 84.2 <strong>86.0</strong></td>
<td>81.5 100 <strong>84.6</strong></td>
</tr>
<tr>
<td></td>
<td>$P_{cd}$</td>
<td>100 0.8 3.7</td>
<td>97.3 0.7 3.3</td>
<td>96.3 0.7 3.5</td>
</tr>
<tr>
<td></td>
<td>$P_d$</td>
<td>100 0.8 3.7</td>
<td>97.3 0.7 3.5</td>
<td>96.3 0.8 3.8</td>
</tr>
<tr>
<td></td>
<td>$P_c$</td>
<td>96.3 3.7 16.0</td>
<td>86.5 3.0 13.2</td>
<td>81.5 2.7 11.8</td>
</tr>
<tr>
<td></td>
<td>$P_{pdg}$</td>
<td>96.3 2.3 10.6</td>
<td>89.2 1.7 8.0</td>
<td>85.2 1.6 7.2</td>
</tr>
<tr>
<td></td>
<td>$P_e$</td>
<td>96.3 29.5 66.3</td>
<td>86.5 14.7 43.7</td>
<td>81.5 15.1 43.3</td>
</tr>
<tr>
<td><strong>GanttProject</strong></td>
<td>$P_{cd+ext}$</td>
<td>91.5 63.1 <strong>84.0</strong></td>
<td>82.1 58.4 <strong>76.0</strong></td>
<td>60.6 54.1 <strong>59.2</strong></td>
</tr>
<tr>
<td></td>
<td>$P_{cd}$</td>
<td>99.2 3.2 14.0</td>
<td>92.9 2.8 12.3</td>
<td>79.6 2.3 10.3</td>
</tr>
<tr>
<td></td>
<td>$P_d$</td>
<td>99.2 3.3 14.4</td>
<td>92.6 2.8 12.3</td>
<td>79.1 2.3 10.4</td>
</tr>
<tr>
<td></td>
<td>$P_c$</td>
<td>92.1 18.1 50.7</td>
<td>60.9 15.8 38.7</td>
<td>22.0 7.5 15.9</td>
</tr>
<tr>
<td></td>
<td>$P_{pdg}$</td>
<td>96.6 4.7 19.5</td>
<td>73.0 4.8 18.9</td>
<td>39.4 2.8 10.8</td>
</tr>
<tr>
<td></td>
<td>$P_e$</td>
<td>98.1 11.4 38.9</td>
<td>90.0 11.2 37.3</td>
<td>79.9 9.1 31.4</td>
</tr>
<tr>
<td><strong>jHotDrw</strong></td>
<td>$P_{cd+ext}$</td>
<td>91.1 52.8 <strong>79.6</strong></td>
<td>81.5 12.5 <strong>38.8</strong></td>
<td>68.6 12.1 <strong>35.4</strong></td>
</tr>
<tr>
<td></td>
<td>$P_{cd}$</td>
<td>97.8 0.8 4.0</td>
<td>95.3 0.8 3.9</td>
<td>91.8 0.8 3.7</td>
</tr>
<tr>
<td></td>
<td>$P_d$</td>
<td>97.5 0.8 4.0</td>
<td>95.2 0.8 3.9</td>
<td>91.5 0.8 3.7</td>
</tr>
<tr>
<td></td>
<td>$P_c$</td>
<td>92.8 10.1 35.2</td>
<td>81.3 9.0 31.1</td>
<td>68.4 7.5 26.1</td>
</tr>
<tr>
<td></td>
<td>$P_{pdg}$</td>
<td>94.5 2.6 11.7</td>
<td>85.3 2.5 11.1</td>
<td>75.4 2.0 9.2</td>
</tr>
<tr>
<td></td>
<td>$P_e$</td>
<td>90.3 9.9 34.4</td>
<td>85.3 8.5 30.3</td>
<td>76.0 7.4 26.7</td>
</tr>
<tr>
<td><strong>jEdit</strong></td>
<td>$P_{cd+ext}$</td>
<td>92.9 57.0 <strong>82.5</strong></td>
<td>82.9 70.8 <strong>80.2</strong></td>
<td>62.1 43.3 <strong>57.2</strong></td>
</tr>
<tr>
<td></td>
<td>$P_{cd}$</td>
<td>94.2 2.3 10.4</td>
<td>87.8 2.5 11.1</td>
<td>74.3 1.5 7.0</td>
</tr>
<tr>
<td></td>
<td>$P_d$</td>
<td>94.2 2.4 10.7</td>
<td>87.3 2.5 11.2</td>
<td>72.9 1.5 7.1</td>
</tr>
<tr>
<td></td>
<td>$P_c$</td>
<td>94.2 15.5 46.8</td>
<td>84.5 23.2 55.3</td>
<td>65.0 10.3 31.6</td>
</tr>
<tr>
<td></td>
<td>$P_{pdg}$</td>
<td>96.1 9.7 34.6</td>
<td>85.2 15.8 45.4</td>
<td>65.0 6.8 23.9</td>
</tr>
<tr>
<td></td>
<td>$P_e$</td>
<td>92.9 42.3 74.9</td>
<td>81.3 54.2 74.0</td>
<td>57.9 29.3 48.4</td>
</tr>
</tbody>
</table>
Table 3.10: Prediction Count Before and After Extensions Applied on P_{cd}

<table>
<thead>
<tr>
<th>Systems</th>
<th>Systems</th>
<th>TPs (Reduced by)</th>
<th>FPs (Reduced by)</th>
</tr>
</thead>
<tbody>
<tr>
<td>iTrust</td>
<td>P_{cd}</td>
<td>29</td>
<td>1495</td>
</tr>
<tr>
<td></td>
<td>P_{cd}+ext</td>
<td>28(-3.4%)</td>
<td>42(-97.2%)</td>
</tr>
<tr>
<td>GanttProject</td>
<td>P_{cd}</td>
<td>202</td>
<td>2476</td>
</tr>
<tr>
<td></td>
<td>P_{cd}+ext</td>
<td>184(-8.9%)</td>
<td>158(-93.6%)</td>
</tr>
<tr>
<td>jHotDraw</td>
<td>P_{cd}</td>
<td>1096</td>
<td>12662</td>
</tr>
<tr>
<td></td>
<td>P_{cd}+ext</td>
<td>1035(5.6%)</td>
<td>3541(-72.0%)</td>
</tr>
<tr>
<td>jEdit</td>
<td>P_{cd}</td>
<td>104</td>
<td>2055</td>
</tr>
<tr>
<td></td>
<td>P_{cd}+ext</td>
<td>98(5.8%)</td>
<td>310(84.9%)</td>
</tr>
</tbody>
</table>

level prediction, best case P_{cd}+ext ensures recall over 90% as well as precision over 20% while in overall P_{cd}+ext achieves recall larger than 75% while keeping precision greater than 10%. However, in jHotDraw we found that P_{c} achieves better F2-score than P_{cd}+ext, since intra-class dependencies are not employed in class-level change impact set prediction. An intra-class dependency means code dependency between two methods in the same class. Intra-class dependencies within the same class are not leveraged in class-level actual change impact set prediction across different classes. However, for method-level predictions, intra-class dependencies are fully employed. Therefore, Table 3.9 show that at method-level F2-score of P_{cd}+ext outperforms P_{c} by 7.6%-72.8% overall and 33.3%-81.0% in best case over the four systems. In method-level prediction best case P_{cd}+ext achieves recall over 90% as well as precision over 50% while in overall mode, P_{cd}+ext achieves recall larger than 81% with precision greater than 10%. In general, data sharing dependency complements call dependency in actual change impact set prediction.

Meanwhile, it is also interesting to learn the performance of CHIP when a single method or class is changed initially. For class-level prediction, Table 3.8 ("Overall-
single” column) shows that the recall of $P_{cd}$ outperforms recall of $P_c$ by 7.6%-31.5% in all four systems. For method-level prediction, Table 3.9 (“Overall-single” column) shows that $P_{cd}$ improves recall by 9.3%-54.6%. After the extensions was added, the F2-score is improved in the range of 0.2%-34.4% by $P_{cd}$+ext in class-level and 9.3%-72.8% in the method-level compared to $P_c$. Because of the same reason previously mentioned, jHotDraw $P_c$ achieves better F2-scores than $P_{cd}$+ext in class-level actual change impact set prediction.

**Statistical Testing:** To determine whether leveraging call and data sharing dependencies with extensions significantly improves the prediction accuracy over standalone call dependencies, we apply two-tailed paired t-test to determine whether the improvement of F2-score by $P_{cd}$+ext over F2-score of $P_c$ is significant using class-level predictions as an example. Method-level statistical test results are similar. Our null hypothesis is: *There is no difference between F2-score of $P_{cd}$+ext and F2-score of $P_c$.*

Table 3.11 and Table 3.12 show that in all four systems $p<0.0001$, which suggests that F2-scores of $P_{cd}$+ext are significantly different than F2-scores of $P_c$ in those systems in both class-level and method-level predictions. Since the mean of F2-scores of $P_{cd}$+ext is greater than the mean of F2-scores of $P_c$, the statistical test results suggest that F2-score of $P_{cd}$+ext are significantly larger than F2-score of $P_c$ in all three systems (except jHotDraw) with the confidence interval (CI) of 95% under the mean of prediction. For jHotDraw, the mean of F2-score of $P_{cd}$+ext appears to be less than F2-score of $P_c$ since intra-class dependencies are not employed as discussed earlier. In method-level predictions, taking advantage of the intra-class call and data sharing dependencies, the mean of F2-scores of $P_{cd}$+ext are greater than the mean of F2-scores of $P_c$ in all four systems, which suggest that F2-score of $P_{cd}$+ext are significantly larger than F2-score of $P_c$ in all four systems. Thus, the actual change impact set predictor built on combined call and data sharing dependencies outperforms the predictor built on call dependencies only.

**Summary:** With an initial given set of changes, in overall and best cases, averagely CHIP with combined code dependencies ($P_{cd}$) can predict more than 90% of the actual change impact sets in both class-level and method-level on the four systems,
which significantly outperforms CHIP with only call dependencies ($P_c$). Extended with Dependency Frequency and $idf$ filters, precision of CHIP with combined code dependencies ($P_{cd}$) are largely improved with little compromise of recall. $P_{cd}^{+ext}$ achieves significantly better F2-score than $P_c$ with call dependencies only. We conclude that data sharing dependencies complement call dependencies in actual change impact set prediction.

**RQ2.** How effective is the data sharing dependencies compared to traditional Program Dependence Graph and evolutionary coupling in actual change impact set prediction?

**Results:** Table 3.8 (class-level) and Table 3.9 (method-level) show that CHIP based on combined call and data sharing dependencies with both Dependency Frequency and $idf$ filter extensions ($P_{cd}^{+ext}$) generate significantly better prediction results as compared to that built upon PDG ($P_{pdg}$). F2-score is improved by 22.7%-52.1% in best case and 3.1%-46.3% overall in class-level. F2-score is improved even further by 47.8%-86.4% in best case and 27.6%-78.0% overall in method-level.

When comparing with $P_e$ based on evolutionary couplings in class-level predictions, $P_{cd}^{+ext}$ generate better prediction results than $P_e$ in iTrust and GanttProject. F2-score is improved by 41.5% and 23.7% in best case and 45.7% and 2.7% overall in class level predictions. At method level prediction, $P_{cd}^{+ext}$ achieves even better prediction results than $P_e$ in all four systems, where F2-scores are improved by 7.5%-45.2% in best case and 6.2%-42.3% overall. Figure 3.9 and 3.10 also show that in all four systems, by setting dependency frequency threshold as none, the actual change impact sets predictions made by $P_{cd}^{+ext}$ can achieve increasingly better F2-scores than $P_{pdg}$ at both class and method-levels with increasing $idf$ threshold. Figure 3.9 and 3.10 also show that as $idf$ threshold increases, $P_{d}^{+ext}$ can achieve even better F2-score than $P_{cd}^{+ext}$. However, our goal is to achieve an optimal precision while maintaining recall at high level (above 75% at class-level prediction, above 80% at method-level). In this case, we think $P_{cd}^{+ext}$ is more valuable than $P_{d}^{+ext}$.

For the case that only a single class is given as the initial given set of changes, $P_{cd}^{+ext}$ outperforms $P_{pdg}$ in class-level by improving F2-score by 6.4%-38.2%. Fur-
Figure 3.9: -a-d. Trend analysis on four systems in class-level prediction: F2-score (%) of prediction by $P_{cd+ext}$, $P_{d+ext}$, $P_c$, $P_{pdg}$ and $P_e$ under difference idf threshold

Figure 3.10: -a-d. Trend analysis on four systems in method-level prediction: F2-score (%) of prediction by $P_{cd+ext}$, $P_{d+ext}$, $P_c$, $P_{pdg}$ and $P_e$ under difference idf threshold

thermore, in method-level prediction, F2-score is improved by 26.2%-77.4%. Compared to $P_e$, $P_{cd+ext}$ achieves better F2-scores for all systems except jEdit in class-level prediction with the F2-score improved by 2.4%-31.4% (class-level) and 8.8%-41.3% (method-level).

**Statistical Testing:** To determine whether combined call and data sharing dependencies with extensions significantly improves the prediction over CHIP with PDG and evolutionary couplings, we apply two-tailed paired t-test to determine whether the improvement of F2-score by $P_{cd+ext}$ over F2-score of $P_{pdg}$ and $P_e$ is significant. Our null hypothesis is: (1). There is no difference between F2-score of $P_{cd+ext}$ and F2-score of $P_{pdg}$; (2). There is no difference between F2-score of $P_{cd+ext}$ and F2-score of $P_e$. Table 3.11 and Table 3.12 show that in all systems $p<0.01$, which suggests that F2-scores of $P_{cd+ext}$ are significantly different than F2-scores of $P_{pdg}$ and $P_e$ in those systems in both class level and method level predictions. Moreover, since the
<table>
<thead>
<tr>
<th></th>
<th>p-value</th>
<th>Mean of F2-score and CI</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>P_{cd}+ext vs. P_{c}</td>
<td>P_{cd}+ext vs. P_{pdg}</td>
</tr>
<tr>
<td>iTrust</td>
<td>&lt;0.0001</td>
<td>&lt;0.0001</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Gantt Project</td>
<td>&lt;0.0001</td>
<td>&lt;0.0001</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>jHotDraw</td>
<td>&lt;0.0001</td>
<td>&lt;0.0001</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>jEdit</td>
<td>&lt;0.0001</td>
<td>&lt;0.0001</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Table 3.12: Four Systems at Method-level: Paired T-test Results, Mean and Confidence Interval (CI) of F2-score by $P_{cd}ext$ vs. $P_c$, $P_{cd}ext$ vs. $P_{pdg}$ and $P_{cd}ext$ vs. $P_e$

<table>
<thead>
<tr>
<th></th>
<th>p-value</th>
<th>Mean of F2-score and CI</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$P_{cd}ext$ vs. $P_c$</td>
<td>$P_{cd}ext$ vs. $P_{pdg}$</td>
</tr>
<tr>
<td>iTrust</td>
<td>&lt;0.0001</td>
<td>&lt;0.0001</td>
</tr>
<tr>
<td>Gantt</td>
<td>&lt;0.0001</td>
<td>&lt;0.0001</td>
</tr>
<tr>
<td>Project</td>
<td>&lt;0.0001</td>
<td>&lt;0.0001</td>
</tr>
<tr>
<td>jHotDraw</td>
<td>&lt;0.0001</td>
<td>&lt;0.0001</td>
</tr>
<tr>
<td>jEdit</td>
<td>&lt;0.0001</td>
<td>&lt;0.0001</td>
</tr>
</tbody>
</table>
mean of F2-scores of P_{cd+ext} are greater than the mean of F2-scores of P_{pdg} and P_e, the statistical test results suggest that F2-score of P_{cd+ext} are significantly larger than F2-score of P_{pdg} and P_e in all four systems with the confidence interval (CI) of 95% under the mean of predictions at both class and method levels. Our actual change impact set predictor built on combined call and data sharing dependencies outperforms the state-of-the-art approaches using PDG and evolutionary couplings.

**Summary:** Extended with Dependency Frequency and idf filters, given an initial set of changes, CHIP with combined code dependencies (P_{cd+ext}) can achieve significantly better F2-score than that with PDG only over all four systems. When comparing with CHIP with evolutionary couplings only in method-level predictions, by employing intra-class code dependencies, P_{cd+ext} can achieve significantly better predictions with F2-score improved in all subject systems as well.

**RQ3.** The combined call and data sharing dependencies (P_{cd+ext}) improve predictions in different change impact scenarios as compared to standalone call dependencies (P_c)?

**Results:** Table 3.13, 3.14, 3.15, and Table 3.16 show how P_{cd+ext} with specific kind of data sharing dependencies only (FA, FM, PP, CA) or with all four kinds combined (all) can contribute to five commonly encountered change impact scenarios. In general, they all achieve significantly better F2-scores than P_c. The results also show that for each of the five change impact scenarios, data sharing dependencies captured in specific JVM events better contribute to the F2-score of prediction than others. For example, in “remove class or statement” scenario, removal of code is sometimes caused by the removal of software features. Thus in data sharing dependencies from field access JVM events (FA) related to the removed code, the accessed data probably needs to be changed as well. This change will then propagate to the classes that need to access that piece of data. Table 3.13 shows that P_{cd+ext} with data sharing dependencies captured in FA (P_{cd+ext-FA}) improves F2-score by 20.9% in best case and 21.1% overall compared to P_c in predicting the actual change impact sets due to class or statement removal. Also in “move refactoring” scenario, the moved code may be used in new features at a new location other than the old location where it
Table 3.13: Five Actual Change Impact Scenarios at Class-level: Precision (P), Recall (R) and F2-score (F2) by $P_{cd}+\text{ext-overall}$, $P_{cd}+\text{ext-FA}$, $P_{cd}+\text{ext-FM}$, $P_{cd}+\text{ext-PP}$, $P_{cd}+\text{ext-CE}$ and $P_c$ (Part I)

<table>
<thead>
<tr>
<th>Scenarios</th>
<th>CHIP Variants</th>
<th>Best Case</th>
<th>Overall</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>R  P    F2</td>
<td>R  P    F2</td>
</tr>
<tr>
<td><strong>Move refactoring</strong></td>
<td>$P_{cd}+\text{ext-overall}$</td>
<td>97.8 10.0 35.5</td>
<td>68.8 20.2 46.5</td>
</tr>
<tr>
<td></td>
<td>$P_{cd}+\text{ext-FA}$</td>
<td>83.7 14.6 42.9</td>
<td>62.9 22.9 46.6</td>
</tr>
<tr>
<td></td>
<td>$P_{cd}+\text{ext-FM}$</td>
<td>84.8 16.0 <strong>45.5</strong></td>
<td>65.8 20.6 45.8</td>
</tr>
<tr>
<td></td>
<td>$P_{cd}+\text{ext-PP}$</td>
<td>94.6 10.0 35.2</td>
<td>67.3 21.1 46.8</td>
</tr>
<tr>
<td></td>
<td>$P_{cd}+\text{ext-CE}$</td>
<td>97.8 10.6 36.9</td>
<td>64.2 22.9 <strong>47.2</strong></td>
</tr>
<tr>
<td></td>
<td>$P_c$</td>
<td>89.1 8.8 31.5</td>
<td>61.3 14.6 37.3</td>
</tr>
<tr>
<td><strong>Remove class or statement</strong></td>
<td>$P_{cd}+\text{ext-overall}$</td>
<td>77.8 10.6 34.3</td>
<td>67.3 10.2 31.8</td>
</tr>
<tr>
<td></td>
<td>$P_{cd}+\text{ext-FA}$</td>
<td>77.8 14.8 <strong>42.0</strong></td>
<td>67.3 18.3 <strong>43.8</strong></td>
</tr>
<tr>
<td></td>
<td>$P_{cd}+\text{ext-FM}$</td>
<td>77.8 13.0 39.0</td>
<td>67.3 17.1 42.4</td>
</tr>
<tr>
<td></td>
<td>$P_{cd}+\text{ext-PP}$</td>
<td>77.8 10.2 33.4</td>
<td>67.3 10.0 31.4</td>
</tr>
<tr>
<td></td>
<td>$P_{cd}+\text{ext-CE}$</td>
<td>74.1 13.1 38.3</td>
<td>66.0 10.7 32.4</td>
</tr>
<tr>
<td></td>
<td>$P_c$</td>
<td>77.8 5.4 21.1</td>
<td>67.3 6.2 22.7</td>
</tr>
</tbody>
</table>
Table 3.14: Five Actual Change Impact Scenarios at Class-level: Precision (P), Recall (R) and F2-score (F2) by \( P_{cd}+\text{ext-overall} \), \( P_{cd}+\text{ext-FA} \), \( P_{cd}+\text{ext-FM} \), \( P_{cd}+\text{ext-PP} \), \( P_{cd}+\text{ext-CE} \) and \( P_c \) (Part II)

<table>
<thead>
<tr>
<th>Systems</th>
<th>CHIP Variants</th>
<th>Best Case</th>
<th>Overall</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Bug fixing</strong></td>
<td>( P_{cd}+\text{ext-overall} )</td>
<td>85.4 25.2 57.8</td>
<td>62.2 33.9 53.3</td>
</tr>
<tr>
<td></td>
<td>( P_{cd}+\text{ext-FA} )</td>
<td>90.0 24.4 58.5</td>
<td>66.9 30.1 <strong>53.8</strong></td>
</tr>
<tr>
<td></td>
<td>( P_{cd}+\text{ext-FM} )</td>
<td>68.2 32.3 55.8</td>
<td>62.0 34.2 53.3</td>
</tr>
<tr>
<td></td>
<td>( P_{cd}+\text{ext-PP} )</td>
<td>84.0 27.2 <strong>59.2</strong></td>
<td>63.4 30.2 52.0</td>
</tr>
<tr>
<td></td>
<td>( P_{cd}+\text{ext-CE} )</td>
<td>89.4 24.9 58.9</td>
<td>65.7 30.2 53.2</td>
</tr>
<tr>
<td></td>
<td>( P_c )</td>
<td>96.3 21.8 57.3</td>
<td>76.7 18.4 47.0</td>
</tr>
<tr>
<td><strong>Functional improvement</strong></td>
<td>( P_{cd}+\text{ext-overall} )</td>
<td>81.7 26.8 58.0</td>
<td>60.1 50.3 57.9</td>
</tr>
<tr>
<td></td>
<td>( P_{cd}+\text{ext-FA} )</td>
<td>82.2 31.5 <strong>62.2</strong></td>
<td>61.3 50.2 <strong>58.7</strong></td>
</tr>
<tr>
<td></td>
<td>( P_{cd}+\text{ext-FM} )</td>
<td>72.1 32.6 58.0</td>
<td>59.0 50.3 57.0</td>
</tr>
<tr>
<td></td>
<td>( P_{cd}+\text{ext-PP} )</td>
<td>76.6 29.3 57.9</td>
<td>58.9 50.3 56.9</td>
</tr>
<tr>
<td></td>
<td>( P_{cd}+\text{ext-CE} )</td>
<td>80.7 27.8 58.5</td>
<td>60.1 51.5 58.1</td>
</tr>
<tr>
<td></td>
<td>( P_c )</td>
<td>92.9 23.0 57.7</td>
<td>69.1 26.4 52.2</td>
</tr>
<tr>
<td><strong>Code replacement</strong></td>
<td>( P_{cd}+\text{ext-overall} )</td>
<td>88.3 41.4 72.0</td>
<td>61.7 50.3 59.0</td>
</tr>
<tr>
<td></td>
<td>( P_{cd}+\text{ext-FA} )</td>
<td>84.4 50.9 <strong>74.6</strong></td>
<td>65.4 50.5 <strong>61.8</strong></td>
</tr>
<tr>
<td></td>
<td>( P_{cd}+\text{ext-FM} )</td>
<td>70.3 34.1 58.0</td>
<td>57.8 50.7 56.2</td>
</tr>
<tr>
<td></td>
<td>( P_{cd}+\text{ext-PP} )</td>
<td>70.3 81.1 72.2</td>
<td>60.9 50.2 58.4</td>
</tr>
<tr>
<td></td>
<td>( P_{cd}+\text{ext-CE} )</td>
<td>89.8 39.9 71.9</td>
<td>61.7 50.5 59.1</td>
</tr>
<tr>
<td></td>
<td>( P_c )</td>
<td>99.2 34.0 71.8</td>
<td>73.9 27.3 55.1</td>
</tr>
</tbody>
</table>
Table 3.15: Five Actual Change Impact Scenarios at Method-level: Precision (P), Recall (R) and F2-score (F2) by P_{cd}+ext-overall, P_{cd}+ext-FA, P_{cd}+ext-FM, P_{cd}+ext-PP, P_{cd}+ext-CE and P_c (Part I)

<table>
<thead>
<tr>
<th>Scenarios</th>
<th>CHIP Variants</th>
<th>Best Case</th>
<th>Overall</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>R  P  F2</td>
<td>R  P  F2</td>
</tr>
<tr>
<td>Move refactoring</td>
<td>P_{cd}+ext-overall</td>
<td>95.9 11.6  39.2</td>
<td>80.8 20.1  50.3</td>
</tr>
<tr>
<td></td>
<td>P_{cd}+ext-FA</td>
<td>92.5 12.5  40.6</td>
<td>78.7 8.8  30.3</td>
</tr>
<tr>
<td></td>
<td>P_{cd}+ext-FM</td>
<td>92.5 11.8  39.2</td>
<td>78.0 20.1  49.5</td>
</tr>
<tr>
<td></td>
<td>P_{cd}+ext-PP</td>
<td>96.6 11.6  39.2</td>
<td>81.4 20.0  50.5</td>
</tr>
<tr>
<td></td>
<td>P_{cd}+ext-CE</td>
<td>96.6 11.7  39.3</td>
<td>78.7 20.0  49.6</td>
</tr>
<tr>
<td></td>
<td>P_c</td>
<td>96.6 11.6  39.2</td>
<td>80.8 9.9  33.2</td>
</tr>
<tr>
<td>Remove class or statement</td>
<td>P_{cd}+ext-overall</td>
<td>99.0 12.6  41.7</td>
<td>87.0 10.0  34.3</td>
</tr>
<tr>
<td></td>
<td>P_{cd}+ext-FA</td>
<td>98.1 14.9  46.4</td>
<td>77.2 12.5  37.9</td>
</tr>
<tr>
<td></td>
<td>P_{cd}+ext-FM</td>
<td>99.0 17.0  50.3</td>
<td>82.4 13.5  40.8</td>
</tr>
<tr>
<td></td>
<td>P_{cd}+ext-PP</td>
<td>99.0 12.7  42.0</td>
<td>88.6 10.1  34.7</td>
</tr>
<tr>
<td></td>
<td>P_{cd}+ext-CE</td>
<td>99.0 14.2  45.2</td>
<td>85.5 10.6  35.3</td>
</tr>
<tr>
<td></td>
<td>P_c</td>
<td>98.1 8.3  30.9</td>
<td>77.2 8.0  28.4</td>
</tr>
</tbody>
</table>
Table 3.16: Five Actual Change Impact Scenarios at Method-level: Precision (P), Recall (R) and F2-score (F2) by P_{cd}+ext-overall, P_{cd}+ext-FA, P_{cd}+ext-FM, P_{cd}+ext-PP, P_{cd}+ext-CE and P_c (Part II)

<table>
<thead>
<tr>
<th>Scenarios</th>
<th>CHIP Variants</th>
<th>Best Case</th>
<th>Overall</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Bug fixing</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>P_{cd}+ext-overall</td>
<td>87.5 10.3 35.1</td>
<td>84.0 10.2 34.3</td>
</tr>
<tr>
<td></td>
<td>P_{cd}+ext-FA</td>
<td>91.3 10.8 <strong>36.6</strong></td>
<td>87.0 10.1 34.6</td>
</tr>
<tr>
<td></td>
<td>P_{cd}+ext-FM</td>
<td>87.5 10.3 35.1</td>
<td>84.0 10.4 <strong>34.8</strong></td>
</tr>
<tr>
<td></td>
<td>P_{cd}+ext-PP</td>
<td>87.5 10.3 35.0</td>
<td>84.0 10.2 34.4</td>
</tr>
<tr>
<td></td>
<td>P_{cd}+ext-CE</td>
<td>87.5 10.4 35.1</td>
<td>84.0 10.1 34.1</td>
</tr>
<tr>
<td></td>
<td>P_c</td>
<td>88.8 8.3 30.2</td>
<td>85.0 7.9 28.8</td>
</tr>
<tr>
<td><strong>Functional improvement</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>P_{cd}+ext-overall</td>
<td>70.0 11.9 35.4</td>
<td>67.6 10.4 32.1</td>
</tr>
<tr>
<td></td>
<td>P_{cd}+ext-FA</td>
<td>90.0 10.2 <strong>35.2</strong></td>
<td>78.4 10.1 <strong>33.4</strong></td>
</tr>
<tr>
<td></td>
<td>P_{cd}+ext-FM</td>
<td>70.0 10.0 31.8</td>
<td>67.6 10.0 31.4</td>
</tr>
<tr>
<td></td>
<td>P_{cd}+ext-PP</td>
<td>70.0 10.4 32.7</td>
<td>67.6 10.0 31.5</td>
</tr>
<tr>
<td></td>
<td>P_{cd}+ext-CE</td>
<td>70.0 10.9 33.7</td>
<td>67.6 10.4 32.1</td>
</tr>
<tr>
<td></td>
<td>P_c</td>
<td>90.0 6.9 26.3</td>
<td>78.4 4.7 18.9</td>
</tr>
<tr>
<td><strong>Code replacement</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>P_{cd}+ext-overall</td>
<td>72.2 21.7 49.2</td>
<td>56.3 50.0 <strong>54.9</strong></td>
</tr>
<tr>
<td></td>
<td>P_{cd}+ext-FA</td>
<td>73.3 21.5 49.4</td>
<td>57.4 22.7 43.9</td>
</tr>
<tr>
<td></td>
<td>P_{cd}+ext-FM</td>
<td>72.2 22.8 <strong>50.4</strong></td>
<td>56.3 23.4 43.9</td>
</tr>
<tr>
<td></td>
<td>P_{cd}+ext-PP</td>
<td>72.2 21.5 49.1</td>
<td>56.5 23.2 43.9</td>
</tr>
<tr>
<td></td>
<td>P_{cd}+ext-CE</td>
<td>74.4 20.8 49.1</td>
<td>56.8 23.1 43.9</td>
</tr>
<tr>
<td></td>
<td>P_c</td>
<td>87.5 17.8 49.0</td>
<td>64.2 19.4 43.9</td>
</tr>
</tbody>
</table>
Table 3.17: Five Actual Change Impact Scenarios at Class-level: Paired T-test Results, Mean and Confidence Interval (CI) of F2-score by $P_{cd+ext}$ vs. $P_c$

<table>
<thead>
<tr>
<th>Scenario</th>
<th>p-value</th>
<th>Mean of F2-score and CI</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$P_{cd+ext}$ vs. $P_c$</td>
<td>$P_{cd+ext}(%)$</td>
</tr>
<tr>
<td>Move refactoring</td>
<td>&lt;0.0001</td>
<td>58.9 (57.9,59.9)</td>
</tr>
<tr>
<td>Remove class or statement</td>
<td>&lt;0.0001</td>
<td>47.3 (44.0,50.5)</td>
</tr>
<tr>
<td>Bug fixing</td>
<td>&lt;0.0001</td>
<td>55.9 (54.9,57.0)</td>
</tr>
<tr>
<td>Functional improvement</td>
<td>0.000670832</td>
<td>58.7 (58.4,59.0)</td>
</tr>
<tr>
<td>Code replacement</td>
<td>0.00035209</td>
<td>59.4 (59.1,59.7)</td>
</tr>
</tbody>
</table>
Table 3.18: Five Actual Change Impact Scenarios at Method-level: Paired T-test Results, Mean and Confidence Interval (CI) of F2-score by $P_{cd+\text{ext}}$ vs. $P_c$

<table>
<thead>
<tr>
<th>Scenario</th>
<th>p-value</th>
<th>Mean of F2-score and CI</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Move refactoring</td>
<td>0.013787341</td>
<td>48.6 (41.7,55.5)</td>
<td>38.1 (31.3,44.9)</td>
</tr>
<tr>
<td>Remove class or statement</td>
<td>&lt;0.0001</td>
<td>45.0 (35.0,55.0)</td>
<td>25.9 (15.9,35.9)</td>
</tr>
<tr>
<td>Bug fixing</td>
<td>0.000118944</td>
<td>46.8 (41.3,52.3)</td>
<td>29.3 (26.9,31.7)</td>
</tr>
<tr>
<td>Functional improvement</td>
<td>&lt;0.0001</td>
<td>43.8 (37.6,50.0)</td>
<td>21.4 (19.1,23.6)</td>
</tr>
<tr>
<td>Code replacement</td>
<td>&lt;0.0001</td>
<td>60.8 (60.2,61.3)</td>
<td>44.2 (43.9,44.5)</td>
</tr>
</tbody>
</table>
was moved from [83]. The data that accessed or manipulated by the moved code is also very likely to be changed and used in those new features, which will propagate changes to other classes which the moved code has data sharing dependency with. Table 3.15 shows that under “move refactoring” scenario, P<sub>cd</sub>+ext with field modification data sharing dependencies (P<sub>cd</sub>-ext-FM) improves F2-score the most by 14% in best case compared to P<sub>c</sub>. In each kind of scenario, we can find at least one specific kind of data sharing dependencies achieve significantly better F2-scores than P<sub>c</sub>.

**Statistical Testing:** The null hypothesis for RQ3 is: *There is no difference between F2-score of P<sub>cd</sub>+ext with selected type of data sharing dependencies only and F2-score of P<sub>c</sub>.* Table 3.17 and Table 3.18 show that in all five scenarios, p is less than or around 0.01, which suggests that in specific type of data sharing dependencies, F2-score of P<sub>cd</sub>+ext is significantly different than F2-score of P<sub>c</sub> in those scenarios in both class-level and method-level predictions. Moreover, since the mean of F2-scores of P<sub>cd</sub>+ext are greater than the mean of F2-scores of P<sub>c</sub>, the statistical test results suggest that F2-score of P<sub>cd</sub>+ext is significantly larger than F2-score of P<sub>c</sub> in all those scenarios with the confidence interval (CI) of 95% under the mean of predictions.

**Summary:** Under each of the five change impact scenarios, P<sub>cd</sub>+ext with data sharing dependencies extracted from specific JVM events can achieve better F2-score than P<sub>c</sub>.

### 3.5. Conclusion

The chapter presents an automated approach and tool (CHIP) for software actual change impact set prediction built upon combined code structural dependencies. It exploits not only call dependencies but also data sharing dependencies, a more comprehensive data dependency by referencing shared datas ID in memory. To improve the precision of the basic predictor relying on data sharing dependencies (P<sub>d</sub> and P<sub>cd</sub>), we introduce two novel Dependency Frequency and data type *idf* filter extensions to the basic predictor. We have empirically evaluated CHIP in both method-level and class-level actual change impact set predictions on four open source systems. The evaluation results support our hypothesis that data sharing dependencies complement
call dependencies in actual change impact set predictions with consistently improved recall and F-score by as much as 19.2%(class–level) and 48.6%(method–level) in average. The results also indicate that after applying data sharing dependencies with both extension, CHIP consistently improves F2-scores of prediction as compared with the ones built on Program Dependence Graph (PDG) which includes dependencies based on data flow and control flow, as well as the ones built on evolutionary couplings in all four systems. By adding both extensions to CHIP, the combined call and data sharing dependencies improve F2-score of prediction by as much as 46.4%(class–level), 86.4%(method–level) compared to the predictors built on Program Dependence Graph (PDG), and 41.4%(class–level), 45.2%(method–level) compared with the ones built on evolutionary couplings. Moreover, our empirical experiment results show that specific type of data sharing dependencies are particularly useful on predicting certain change impact scenarios.

3.6. Bibliographical Notes

The paper supporting the content described in this Chapter were written in collaboration with the members at Southern Methodist University, Johannes Kepler University, and Nanjing University.

Chapter 4
TOWARDS AUTOMATED SOFTWARE CHANGE – RECOMMENDING CHANGED SOURCE CODE AND CHANGE INTENT LINKS

As discussed in Chapter 3, source code changes are committed through version control systems (VCS) such as Subversion and Git during software development and evolution. When changes are committed, the intents behind these changes are also documented in commit messages by developers. Problems arise, however, when developers overlook the individual change intent (i.e., one segment of change message sentences that is related to one unique task in this commit) in commit messages. This can lead to confusion, omissions and errors when developers validate changes, locate bug reports, (re)assign bug reports and trace changes to other software artifacts [31]. For example, if one developer committed two bug fix tasks that are tangled together, it could be hard for other developers to verify at the code level if both bugs are fixed without knowing which changed source code is related to which task. Therefore, it is critical and non-trivial to extract untangled change intents by slicing a commit message containing aggregated change intents, and identify/recover links between changed source code files and their untangled change intents—we speak of code to untangled change intent links. Herzig et al. [53] highlight the importance of this question by investigating the frequency of tangled changes and how big the impact of tangled changes is. Their investigation results confirm that tangled changes should be avoided. Unfortunately, to the best of our knowledge, no existing research has addressed the task of identifying code to untangled change intent links.

State-of-the-art practices commonly identify links between code changes and untangled change intents manually. However, this process is time-consuming, labor-intensive, and requires a great deal of experience. First, developers may have to review not only commit messages but also other software artifacts such as issue reports and pull requests. Figure 4.1 shows an example from project gmail4j, in which
Figure 4.1: An example of how software developers link changed source code files to untangled change intents

individual change intents described in the segmented commit message of commit \textit{ae66810} need to be linked to the corresponding changed source code files. In this example, to identify the link between the changed source code file and the change intent described in segmented commit message “\textit{Issue 13: quick fix}”, the developer starts by locating \textit{issue 13}. The developer notices that the change involves removing the package \textit{maven-source-plugin}. To learn where exactly this change is made, she will need to go through all changed source code files before realizing that the changed file \textit{ImapConnectionHandler.java} is linked to “\textit{issue 13: quick fix}” since one of its imports \textit{org.apache.maven.plugin} is removed.

Although no one has addressed this linking task, one may argue that Information Retrieval (IR)-based approaches such as Latent Semantic Indexing (LSI), Vector Space Model (VSM), Association-based approach (e.g., [17, 22, 22, 53, 121]) could be applied to generate links between commits and software artifacts (e.g., bug reports, design documents, etc.) by comparing the textual similarity between commits and software artifacts. However, these approaches cannot adequately address our task. The reason is that the changed entities extracted from source code (e.g., identifier, comments, string literals) could be very different from what is described in commit messages and other related software documents (e.g., issue reports, pull requests, etc.). As an example, consider Figure 4.1 again. The terms used to describe the change intent in the commit message (i.e., “quick”, “fix”, “remove”, “plugin”, etc.)
are mostly different than the identifiers used in the changed source code file (i.e., “org”, “apache”, “maven”, “plugin”, “Abstract”, “Mojo”).

Motivated by this observation, we propose AutoCILink, a novel method with a supporting tool for automatically identifying/recovering links between the untangled change intents in segmented commit messages and the changed source code files. To address the issue of one source code file being changed for multiple intents, AutoCILink is designed to relate each changed file to one or more changed intents. Specifically, we have designed two variants of AutoCILink. To simulate the human reasoning process, we propose a pattern-based link identification system (AutoCILink-P) that leverages manually defined patterns to identify links between untangled change intents and changed code files. Motivated by the successful application of machine learning to software engineering tasks, we develop a supervised learning-based link classification system (AutoCILink-ML) to further understand the intents of source code changes and their associations. AutoCILink-ML identifies links between untangled change intents and code changes by encoding the patterns introduced in AutoCILink-P as features (regular expression features and vocabulary features) and employing two novel features: (a) code import, which considers changed imports from committed source code files; and (b) untangled change intent count, which considers the number of untangled change intents in a commit message.

The main contributions of this chapter are:

- **Novel task.** To our knowledge we are the first to examine the task of automatically identifying links between changed source code files and untangled change intents.

- **New resources.** We have annotated a new corpus with links between untangled change intents and changed source code from 19 projects and make this corpus publicly available to stimulate research on this task.

- **Novel pattern-based approach.** We discover that (1) some patterns are recurrently used by developers to trace change intents and further build the links
between changed code files and untangled change intents; and (2) related software documents (i.e., issue reports and pull requests) frequently provide root causes of code changes. We leverage these insights to design a pattern-based link identification system (a.k.a. AutoCILink-P). These patterns are designed with regular expressions that (1) extract changed entities (i.e., classes that encapsulate data and behavior [36]) from changed code files and (2) compute the similarity between these changed entities and the terms that appear in untangled change intents in commit messages as well as in other related software documents (i.e., issue reports and pull requests).

- Novel learning-based approach. To further our attempt to automatically generate links from code changes to untangled change intents, we have developed a supervised learning based link classification system (a.k.a., AutoCILink-ML) using novel features specifically designed for this task.

4.1. Motivating Examples

We motivate the development of AutoCILink via two examples. Figure 4.1 shows an example from project gmail4j about how a developer reasons about the link between the change intent described in the segmented commit message “Issue 13: quick fix” and the changed source code file ImapConnectionHandler.java via related issue report issue 13. To reduce human efforts, if an IR-based approach (e.g., VSM) is employed to automate the task, this link is unlikely to be recovered due to the differences in the entities used in the source code ImapConnectionHandler.java and the terms used in the segmented commit message. AutoCILink-P follows a different way to mimic the human reasoning process. First, commit message ae66810 is automatically segmented into three parts, each of which describes a different intent: (1) “Issue 13: quick fix”; (2) “# moved the declaration of StringBuilder toString inside toString() method”; and (3) “# added missing javadoc to private long startTime instance vari-

\[\text{The terms extracted from “Issue 13: quick fix” are: “issue”,“quick”,“fix”.
The entities extracted from ImapConnectionHandler.java: “org”,“apache”,“maven”,“plugin”,“abstract”,“mojo”,etc.}\]
In addition, AutoCILink relates issue 13 to (1) since (1) clearly indicates that it is relevant to fixing issue 13. Then to find which segmented commit message (i.e., (1), (2), or (3)) is linked to from one of the changed files ImapConnectionHandler.java, AutoCILink-P generates a regular expression using the extracted changed entity “plugin” (i.e., \^.*\s*remove\(.*\)plugin\(.*\)) from ImapConnectionHandler.java and applies it to each segmented commit message and their related software documents. At last, AutoCILink-P finds a link from ImapConnectionHandler.java to segmented commit message (1) by matching the mentioned regular expression in Comment 1 of (1)’s related software document (i.e., issue 13): “I just removed the plugin...”.

However, since regular expressions identify links based on simple word matching between source code and the text in a change intent, they could introduce errors. For instance, Figure 4.2 shows an example from commit 18eb05d in project activiti-crystalball, in which SimulationRun.java can be erroneously linked to the sentence “Performance can be increased by increasing maxWaitTime for job executor too.” (i.e., false positive link in Figure 4.2) that is not the actual change intent by matching the regular expression \^.*\s*increase\(.*\)job\(.*\). This regular expression is generated using the extracted changed entity “job” and it matches with the supporting text in the intent description “...increasing...job executor...” rather than the main topic of the intent (i.e., “…increasing maxWaitTime...”). To overcome this weakness, AutoCILink-ML is developed. By learning from known links, each feature is weighted by AutoCILink-ML, which can effectively reduce the negative effects imposed by regular expressions.
4.2. Approach

4.2.1. Overview

To enable the automated tracing from changed source code files to untangled change intents, we have designed two variants of AutoCILink: pattern-based link identification system (AutoCILink-P) and supervised learning-based link classification system (AutoCILink-ML). The workflow of both systems is shown in Figure 4.3. The subsequent sub-sections elaborate the text preprocessing and methodology employed by the two systems.

![AutoCILink system workflow](image)

Figure 4.3: AutoCILink system workflow

4.2.2. Text preprocessing

As shown in Figure 4.3, the text preprocessing component seeks to accomplish three tasks:

4.2.2.1. Untangling change intents

We untangle change intents in two steps. First, we employ the Stanford Tokenizer from the CoreNLP toolkit [82] to split the commit message into individual sentences. Then each sentence is further segmented if multiple change intents are aggregated in one sentence. The second step can be done when an additive transition word is used².

²We apply an additive transition word list pre-defined in: https://msu.edu/user/jdowell/135/-
For instance, the following sentence in a commit message “enable proguard by default and correct small mistake in debug log” can be further split into two untangled change intents: “enable proguard by default” and “correct small mistake in debug log”. We manually verified that 87% of the intents are correctly untangled by this procedure.

4.2.2.2. Building enriched untangled change intents

We leverage additional software documents related to the intent to build an enriched untangled change intents. If a software document (i.e., an issue report or a pull request) is related to any untangled change intent in a commit, it will be appended as supplemental description to the corresponding intent. Since this relationship is not always explicitly provided during software development and evolution, it is worthwhile to automatically recover the missing relations by performing the following analysis:

**Step 1:** Pairing untangled change intents and software documents. In the example shown in Figure 4.1, AutoCILink groups three untangled change intents in commit ae66810 with Issue 13 report into 3 distinct pairs; **Step 2:** Similarity analysis. For each pair generated in Step 1, we apply Vector Space Model (VSM) [81] to calculate the cosine similarity score using Equation (1).

\[
sim(i, d) = \cos(i, d) = \frac{\sum_{t \in T} \omega(t, i, T) \times \omega(t, d, T)}{\sqrt{\sum_{t \in T} \omega(t, i, T)^2} \times \sqrt{\sum_{t \in T} \omega(t, d, T)^2}}
\]

where \( t \) is a term, \( d \) is a software document, \( i \) is an untangled change intent, \( T \) is a corpus of terms taken from the commit messages and the software documents, \( \omega(t, d) \) is the weight of term \( t \) in software document \( d \), and \( \omega(t, i) \) is the weight of term \( t \) in untangled change intent \( i \). All terms are extracted by tokenizing the corresponding textual content. The weight of each term is calculated using Term Frequency and Inverse Document Frequency (tf-idf) (e.g., \( \omega(t, d, T) = tf(t, d) \times idf(t, T) \)), where \( tf(t, d) \) is the term frequency of term \( t \) in software document \( d \) and Inverse document frequency (i.e., \( idf(t, T) = \log \frac{|T|}{|\{d : t \in d\}|} \)) distinguishes rare terms from those common terms by giving more weight to rare terms in the corpus. Based on the similarity scores calculated in Step 2, AutoCILink determines that a given pair of software document and untangled change intent are related if \( \sim(i, d) > 50\% \), as Bacchelli
et al. [12] indicate that using a 50% threshold yields better results than using other thresholds.

4.2.2.3. Extracting terms and changed entities

Next, AutoCILink extracts (1) terms from an enriched untangled change intents and (2) changed entities from changed source code files. Terms are extracted by tokenizing each untangled change intent and related software documents in the enriched untangled change intent. Since entities can be referenced in many different ways [12], AutoCILink extracts changed entities from changed source code identifiers, inline comments and string literals. Each entity is defined as a single noun word that represents a class that encapsulates data and behavior [36]. Specifically, since developers compound words through camel casing [12] (e.g., ObjectContainer is formed from “object” and “container”) and underscore separator (e.g., TYPE_END_SIMULATION is formed from “type”, “end” and “simulation”), these compounded words must be split into separate entities. In addition, each word in inline comments is tagged with its part-of-speech (POS) tag (a label that is assigned to a word to indicate its syntactic function) using the Stanford Log-linear Part-Of-Speech Tagger from the CoreNLP toolkit [82]. All words tagged with noun POS tags (i.e., NN, NNP, NNS, NNPS) are extracted as entities. Terms and changed entities are not only preprocessed by filtering common English stop words, but also stemmed using Porter Stemmer ( [105,125]) —a widely used algorithm that heuristically converts a word to its stem.

4.2.3. Pattern-based Link Identification System (AutoCILink-P)

As shown in Figure 4.3, the pattern-based link identification system (AutoCILink-P) operates in two steps;

4.2.3.1. Link identification using regular expression (AutoCILink-P-R)

Link identification using regular expression (AutoCILink-P-R) is inspired by our observation that entities are usually named by their roles and responsibilities in system design [36].
Generating regular expressions. Given this observation, for each changed entity extracted from changed source code identifiers, inline comments and string literals (mentioned in Section 4.2.2.3), AutoCILink-P-R generates the regular expressions based on the following two regular expression templates:

(1). `\^\.*\s*(<verb>)(.*?)(<entity>)(.*\))`

(2). `\^\.*\s*<entity>(.*?)(<verb>)(.*\))`

The intuition behind both templates is that in commit messages and software document, the terms that indicate changed entities could be separated from the governing verbs (simulating active voice in template (1) and passive voice in template (2)) by other characters (e.g., empty spaces, coma, etc.) or describing words. For example, in “moved the declaration of StringBuilder toString inside toString() method”, active verb “moved” is separated from terms “String” and “Builder”, which represents changed entities, by a sequence of words that describes “String” and “Builder” (i.e, “the declaration of”).

As shown in both regular expression templates, `<verb>` represents an acting verb in an untangled change intent. We define three kinds of verbs: reserved verbs, synonyms and frequent verbs. Reserved verbs and synonyms are defined by the nine change types in [31]. Each change type represents a kind of change in source code entity roles or responsibilities (i.e., add new features, fix bugs, improve existing features, add code components, modify code components, delete code components, deprecate code components, refactor code components, modify code inline text). All reserved verbs are taken directly from the verbs of various change types. As shown above, there are seven reserved verbs for the nine change types: “add” for add new features and add code components, “modify” for modify code components and modify code inline text, “delete” for delete code components, “deprecate” for deprecate code components, “refactor” for refactor code components, “fix” for fix bugs, and “improve” for improve existing features. To improve the likelihood of a match, we additionally employ the synonyms of these reserved verbs. The synonyms of each reserved verb manually
identified based on its *synset* (i.e., sets of cognitive synonyms) defined in the WordNet [39] lexical knowledge base. Specifically, for each reserved verb, we manually collect all its *synsets* based on the intended meaning of the verb. For example, for reserved verb “add”, we collect all the “synsets” for its intended meaning “*make an addition (to)*”. Note that for each changed entity, we generate regular expressions using all seven reserved verbs from all possible change types and their synonyms.

In addition, *AutoCILink-P-R* leverages additional frequent verbs beyond the seven reserved verbs to generate regular expressions, because we observe that some verbs that are related to the nine change types are neither reserved verbs nor their synonyms. For example, the verb “update” is related to *improve existing features*, which is neither a reserved verb nor a synonym of the reserved verb “improve”. Based on this observation, we mine verbs that most frequently occur in commit messages. Specifically, we first POS-tag each term in each commit message using the Stanford Log-linear Part-Of-Speech Tagger. Then we extract all the terms tagged with verb POS tags (i.e., VB, VBD, VBG, VBN, VBP, VBZ) and rank them based on their term frequency across all commit messages.

*Applying regular expressions.* We apply each generated regular expression on the enriched untangled change intents. If any match of the regular expression is identified in the enriched intents, *AutoCILink-P-R* reports a link between the untangled change intent and the changed source code file that contains the changed entity in the regular expression.

### 4.2.3.2. Link identification using vocabulary similarity (*AutoCILink-P-V*)

Intuitively each changed source code file should be linked to at least one untangled change intent. If *AutoCILink-P-R* described in Section 4.2.3.1 cannot identify any link from the given changed source code file to any untangled change intent, we identify the missing link based on the vocabulary similarity between each changed source code file and enriched untangled change intents (*AutoCILink-P-V*). The motivation behind *AutoCILink-P-V* is that commit messages and software documents typically elaborate code change details with similar textual information with changed source
AutoCILink-P-V first computes the vocabulary similarity between enriched untangled change intents and changed source code files, which includes the vocabulary similarity between untangled change intent terms and changed source code entities (i.e., \( \text{sim}(i,c) \)) and the vocabulary similarity between terms in related software document (i.e., issue report or pull request) and changed source code entities (i.e., \( \text{sim}(d,c) \)):

\[
\text{sim}(i,c) = \frac{\sum_{t \in V, e \in V} \omega(t,i,V) \times \omega(e,c,V)}{\sqrt{\sum_{t \in V} \omega(t,i,V)^2} \times \sqrt{\sum_{e \in V} \omega(e,c,V)^2}}
\]

\[
\text{sim}(d,c) = \frac{\sum_{t \in V, e \in V} \omega(t,d,V) \times \omega(e,c,V)}{\sqrt{\sum_{t \in V} \omega(t,d,V)^2} \times \sqrt{\sum_{e \in V} \omega(e,c,V)^2}}
\]

in which \( t \) is a term, \( i \) is an untangled change intent, \( d \) is a related software document, \( e \) is a changed entity, \( c \) is a changed source code file, \( V \) is a corpus of extracted terms and entities, \( \omega(e,c,V) \) is the weight of changed entity \( e \) in changed source code \( c \) in corpus \( V \). Similar to \( \omega(t,i,V) \) and \( \omega(t,d,V) \), \( \omega(e,c,V) \) is also calculated using the tf-idf approach.

In enriched untangled change intents, one untangled change intent could be related to multiple software documents. As a result, AutoCILink-P-V chooses the max value as the vocabulary similarity between the untangled change intent and changed source code file (i.e., \( \text{sim}(\text{Link}(i,c)) \)), as shown in Equation (4.2), in which the number of related software documents \( (d_x) \) is \( n \):

\[
\text{sim}(\text{Link}(i,c)) = \max \left( \text{sim}(i,c), \bigcup_{x=1}^{n} \text{sim}(d_x,c) \right)
\]

To decide if there is a link or not, we set a threshold. The threshold for each experiment is selected by using a development set (details explained in Section 4.3.2.1). That is, when \( \text{sim}(\text{Link}(i,c)) \) larger than selected threshold, AutoCILink-P-V reports a link between the changed source code file and the untangled change intent.

As we mentioned in Section 4.1, AutoCILink-P may erroneously report certain links between changed source code files and untangled change intents as regular expressions are imprecise. To tackle this problem, we propose the supervised learning-based link classification system.
4.2.4. Supervised learning-based link classification system (AutoCILink-ML)

AutoCILink-ML operates in three steps:

4.2.4.1. Creating training instances

To train a link classifier to identify links, we create training instances for each changed source code and enriched untangled change intent pair in the training dataset. We annotate each training instance as linked or not linked depending on whether there is a link between the pair or not. Each training instance is represented using the following features:

Regular expression features are inspired by link identification using regular expressions described in Section 4.2.3.1, which generates and uses regular expressions to identify code to untangled change intent links. Specifically, we create one binary feature for each such regular expression. Each feature encodes the presence (value=1) or absence (value=0) of the regular expression in the enriched untangled change intent in the training set. In the example shown in Figure 4.2, the untangle change intent in the commit message contains one regular expression feature (i.e., the one encodes the presence of ^.*\s*increase(.*?)<entity>(.*) having the value 1. Unlike in AutoCILink-P, in AutoCILink-ML the learning algorithm will have the flexibility to determine which of the regular expressions to use. For instance, if a regular expression is deemed useless, the learner can simply assign a low weight to the corresponding feature.

Vocabulary features are motivated by link identification using vocabulary similarity in Section 4.2.3.2. Recall that a link is identified when vocabulary similarity score of the changed source code file and enriched untangled change intent pair is greater than 50%. Specifically, we create three types of vocabulary features: vocabulary pair features, vocabulary similarity features and term unmatched features.

We use (TERM, ENTITY) pairs extracted from change intent and code pairs to create vocabulary pair features. As described in Section 4.2.2.3, terms are extracted from enriched untangled change intents and entities are extracted from changed source code files. Each term and entity are paired to create a (TERM, ENTITY) feature, whose
value is 1 if the particular (term, entity) pair appears in the change intent and code pair under consideration. Otherwise, its value is 0. Returning to the example in Figure 4.2, one of the vocabulary pair features created for the untangled change intent ("Implementation of jobs for simulation process engine") and changed source code file (SimulationRun.java) pair is (process, job).

The vocabulary similarity features are created to encode the vocabulary similarity scores between an enriched untangled change intent and a changed source code file. We group the similarity scores into ten ranges: [0, 10%), [10%, 20%), [20%, 30%), [30%, 40%), [40%, 50%), [50%, 60%), [60%, 70%), [70%, 80%), [80%, 90%), [90%, 100%] and define ten binary features on these ten ranges. The feature value is 1 if the similarity score falls in the corresponding range. Otherwise, the value is 0.

Finally, for each changed source code and enriched untangled change intent pair, we create term unmatched features, which encodes the percentage of unmatched noun terms with changed entities in source code (i.e., the percentage of noun terms that do not appear in the set of words derived from the entities). Specifically, AutoCILink-ML first obtains all nouns (i.e., the words tagged by the Stanford Log-linear Parts-Of-Speech Tagger as NN, NNP, NNS, NNPS) from the tokenized segmented commit message. Then it calculates the percentage of noun terms that are unmatched with changed entities. We define ten binary features, each of which encodes the presence (value=1) or absence (value=0) of the calculated percentage value in the range of: [0%,10%); [10%,20%); [20%,30%); [30%,40%); [40%,50%); [50%,60%); [60%,70%); [70%,80%); [80%,90%); [90%,100%]. In the example in Figure 4.2, there are 3 out of 7 (42.9%) noun terms ("time", "job", "executor") in the untangled change intent "Performance can be increased by increasing maxWaitTime for job executor too"\(^3\), which match with the changed entities extracted from SimulationRun.java. Therefore, the feature representing range [50%,60%) will have its value set to 1 since the percentage of noun terms unmatched in changed entities (1-42.9%=57.1%) falls in the range of [50%,60%).

\(^3\)The terms extracted from the untangled change intent after tokenization are: “perform”, “increas”, “max”, “wait”, “time”, “job”, “executor”
Code import features are motivated by the fact that the intent of a change in source code file may be affected by the changes of its imported code modules (i.e., imported class from the same project). We create 10 code import features (by discretizing the [0,1] range into 10 equal-sized intervals, similar to what was done for the term unmatched features described above) to encode the percentage of terms in the corresponding enriched untangled change intent that are unmatched in each changed imported code module’s changed entities. Similar to terms unmatched features, the feature value is 1 if the calculated percentage falls in a specific range defined above.

Untangled change intent count features are inspired by our hypothesis that the less untangled change intents one commit message can be segmented into, the more likely each intent is linked to more changed code files in this commit. For example, when there are ten changed source code files but only one untangled change intent in one commit, most likely all changed source code files are linked to this single intent (i.e., percentage of linked changed code=100%). As another example, if there are ten changed source code files and ten untangled change intents segmented from one commit, the average percentage of linked changed code will be approximately 10% (i.e., 1/10=10%), with each untangled change intent linked with one distinct changed source code file. Figure 4.4 empirically validates our hypothesis based on data from 19 subject projects collected in the data preparation step (Section 4.3.1). Specifically, Figure 4.4 plots the percentage of changed source code files each untangled change intent is linked to on average against the number of untangled change intents in a

Figure 4.4: Average percentage of linked changed source code files per untangled change intent count across all 19 projects
commit message. As we can see, although there are minor fluctuation when untangled change intent count=3 and 4, the overall trend is obvious: the more the untangled change intents a commit message contains, on average the lower percentage of changed source code files is linked to each untangled change intent. Figure 4.4 shows that the percentage drops from 96.8% (when untangled change intent count=1) to 38.4% (when untangled change intent count=6).

Given these results, we create the untangled change intent count features, which are binary features that encode the number of untangled change intents in each commit message. The value of a feature is 1 if the commit message is segmented into the corresponding number of untangled change intents.\(^4\) Otherwise its value is 0. For example, in Figure 4.1 the commit message ae66810 is segmented into four untangled change intents. Therefore, the training instances that AutoCILink-ML creates for each changed source code file and enriched untangled change intent pair in this commit set the value of the feature four untangled change intents to 1.

4.2.4.2. Training and applying link classifier

We employ the support vector machine (SVM) learning algorithm with Radial Basis Function (RBF) kernel as implemented in the libSVM software package [27] to train the link classifier. This binary classifier determines whether the given untangled change intent and changed source code file are “linked” or “not linked”. After training, the resulting classifier can be used to label each test instance. Test instances are created in the same way as the training instances.

4.3. Empirical Evaluation

This section presents the experiment results to answer our research questions proposed in this chapter.

4.3.1. Data preparation

\(^4\)Based on our study across all 19 projects, the number of untangled change intents falls in the range of [1,7]. Thus 7 binary features are created.
We randomly selected 19 open source software projects of different types and domains hosted on GitHub\textsuperscript{5}. The data we collected for our experiments include all change commits, issue reports and pull requests from the 19 projects. Table 4.1 summarizes the details of these projects.

Given the dataset, we created a binary classification task, where we seek to identify each pair of changed source code file and enriched untangled change intent as “linked” or “not linked”. Hence, for system training and evaluation, we manually determined whether there was a link between each pair of enriched untangled change intent and changed source code file, as described below:

4.3.1.1. Coding procedure

Two expert analysts were asked to conduct the coding task. First, one of the coders conducted a pilot study on a subset of all the untangled change intents and changed source code file pairs. The pilot study resulted in a list of preliminary coding criteria to identify links between untangled change intent and source code files. After that, this coder trained the other coder with coding criteria in a session that involved open discussion. To minimize subjectivity of coding, both coders coded all code-intent

\textsuperscript{5}The 19 projects are: activiti-crystalball, androidkickstartr, deprecated-avaje-ebeanorm-api, zoola, SiVi, semaphore manager, gmail4j, java-color-loggers, java-semver, jdf-stacks-client, kornakapi, picketbox-cdi, sonar-scm-stats, sonar-switch-off-violations, xml2csv, cascading-helpers, dasein-cloud-gogrid, elasticsearch-facet-script, fest-guava-assert
pairs in the dataset. Each coded pair was verified by both coders and disagreements were resolved through open discussion.

4.3.1.2. Coding results

We measured the inter-coder agreement between the two coders with Cohen’s Kappa ($k$) [30]. Coders agreed on 86% ($k$=0.683, i.e., moderate agreement [131]) of the cases and then resolved their disagreements via discussion. Based on our analysis, the main causes of the disagreements are omissions and misunderstandings in several cases that the terms found in segmented commit messages (untangled change intents) have different meanings in the changed source code entities. For example, the word initial in untangled change intent “initial Simulator implementation” from project activiti-crystalball “initial” can be understood as “the first time”. However, in the changed source code it actually means “the fundamental”.

4.3.2. Experimental setup

4.3.2.1. Evaluation settings

To evaluate AutoCILink-ML, we apply leave-one-project-out cross validation. In each experiment, we use 17 projects for training AutoCILink-ML, one project for development (i.e., parameter tuning), and the remaining project as our held-out test set. We repeat this experiment 19 times, each time choosing a different project as our held-out test set. This ensures that the entire dataset is used for evaluation. For parameter tuning, we tune SVM’s regularization parameter C. Intuitively, the larger the C value is, the higher the penalty on training error is. We tune C that maximizes the average accuracy on the development set. To evaluate AutoCILink-P, we apply it to the entire dataset. In other words, results of both variants of AutoCILink are obtained on the entire dataset.

4.3.2.2. Baseline Systems

Since this is a new task, there is no existing system that we can employ as a
baseline system. Nevertheless, as mentioned in the introduction, one could conceivably employ IR-based approaches as baselines for our task. Consequently, we employ three IR-based baselines, namely VSM \[72, 125\], LSI \[56\] and the Association-based approach \[98\]. Recall that LSI is based on the Vector Space Model that takes words appearing in a context into consideration. VSM calculates the distance between terms from enriched untangled change intents and entities from changed source code files. Association-based approach learns the associations between terms from enriched untangled change intents and the entities from changed source code. Since each IR-based baseline returns a similarity score for each test instance, we need to employ a threshold to determine whether a test instance should be classified as “linked” or “not linked”. Specifically, a test instance whose similarity score is at least as large as the threshold will be classified as “linked”. Otherwise, it will be classified as “not linked”. To avoid giving an unfair advantage to \textit{AutoCILink} whose learning-based variant employs annotated training data, we use the same amount of annotated training data for identifying the “best” threshold for each IR-based baseline. Specifically, when evaluating an IR-based baseline on a particular project, we apply the threshold that achieves the highest accuracy on the remaining 18 projects.

We employ two additional baseline systems: the \textit{majority classifier} and the \textit{untangled intent count classifier}. The \textit{Majority classifier} takes a greedy approach in link identification, simply classifying each test instance into the majority class, which for our task means that each test instance will be classified as “linked”. The \textit{untangled intent count classifier} is designed based on the assumption that the less untangled change intents one commit message contains, the more changed source code files each untangled change intent in this commit will be linked to. Specifically, it first assigns a random score between 0 and 1 to each test instance (code-intent pairs). Then, a threshold is employed so that a test instance is classified as “linked” if and only if its score is below the threshold. The threshold is dependent on the number of untangled change intents in the commit message the test instance is associated with. Specifically, the threshold is set as the reciprocal of the count of untangled change intents in one commit message under consideration. For example, the threshold for a commit
contain one untangled change intent commit is set as 1, while the threshold for a
commit containing two untangled change intents is set as 1/2.

4.3.2.3. Evaluation metrics

We compute accuracy, as well as recall, precision, and F1-score on both “linked”
and “not linked” instances. The recall (R) on the “linked”/“not linked” class is de-
ﬁned as the percentage of “linked”/“not linked” instances in the test set that are
correctly classiﬁed (i.e., Recall = \( \frac{TP}{TP+FN} \)). The precision (P) on the “linked”/“not
linked” class is the percentage of code-intent pairs classiﬁed as “linked”/“not linked”
that are indeed correct “linked”/“not linked” classiﬁcations (i.e., Precision = \( \frac{TP}{TP+FP} \)).
The F1-score (F) on the “linked”/“not linked” instances is the harmonic mean of
“linked”/“not linked” recall and precision (i.e., F1-score=\( \frac{2 \times R \times P}{R+P} \)). Finally, we re-
port Accuracy, which is the percentage of code-intent pairs correctly classiﬁed (i.e.,
Accuracy = \( \frac{TP+TN}{TP+FP+FN+TN} \)).

4.3.3. Evaluation results

This section evaluates the effectiveness of AutoCILink by addressing the four
research questions.

RQ1. How effective is AutoCILink in linking changed code to untangled change
intents?

We compare the effectiveness of two variants of AutoCILink systems (i.e., AutoCILink-
ML and AutoCILink-P) with the ﬁve baselines (i.e., the three IR-based systems, the
majority classiﬁer, and the untangled intent count classiﬁer. Table 4.2 shows the F1-
scores of “linked” code-intent pairs, “not linked” code-intent pairs and the accuracy of
each system averaged over the 19 projects using leave-one-project-out cross validation.
As we can see, in terms of average accuracy, both AutoCILink variants (rows 1–2)
outperform the baseline systems (rows 3–7). In particular, AutoCILink-ML achieves
the best average accuracy (81.2%).

Table 4.2 shows several additional interesting results. First, we found AutoCILink-
ML outperforms other systems on not only the “linked” instances but also the “not
Table 4.2: Evaluation results of AutoCILink and baseline approaches

<table>
<thead>
<tr>
<th>Systems</th>
<th>Linked Avg.F1</th>
<th>Not linked Avg.F1</th>
<th>Average Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 AutoCILink-ML</td>
<td>87.4</td>
<td>62.4</td>
<td>81.2</td>
</tr>
<tr>
<td>2 AutoCILink-P</td>
<td>83.4</td>
<td>40.2</td>
<td>74.6</td>
</tr>
<tr>
<td>3 LSI</td>
<td>73.2</td>
<td>32.0</td>
<td>64.3</td>
</tr>
<tr>
<td>4 VSM</td>
<td>78.4</td>
<td>36.2</td>
<td>70.4</td>
</tr>
<tr>
<td>5 Association-based</td>
<td>80.4</td>
<td>7.8</td>
<td>66.5</td>
</tr>
<tr>
<td>6 Majority</td>
<td>82.4</td>
<td>0.0</td>
<td>70.1</td>
</tr>
<tr>
<td>7 Untangled intent count</td>
<td>77.9</td>
<td>55.9</td>
<td>70.0</td>
</tr>
</tbody>
</table>

Table 4.3: Paired t-test results on accuracy of AutoCILink-ML vs other approaches

<table>
<thead>
<tr>
<th>System 1</th>
<th>System 2</th>
<th>p value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 AutoCILink-ML</td>
<td>LSI</td>
<td>&lt;0.0001</td>
</tr>
<tr>
<td>2 AutoCILink-ML</td>
<td>VSM</td>
<td>0.0002</td>
</tr>
<tr>
<td>3 AutoCILink-ML</td>
<td>Association-based</td>
<td>&lt;0.0001</td>
</tr>
<tr>
<td>4 AutoCILink-ML</td>
<td>Majority</td>
<td>0.0920</td>
</tr>
<tr>
<td>5 AutoCILink-ML</td>
<td>Untangled intent count</td>
<td>0.0931</td>
</tr>
<tr>
<td>6 AutoCILink-ML</td>
<td>AutoCILink-P</td>
<td>0.0030</td>
</tr>
</tbody>
</table>
linked” instances. When compared with the five baseline systems, AutoCILink-ML improves the “linked” F1-score by 5–14.2% and the “not linked” F1-score by 6.5–62.4%.

In addition, we observe that AutoCILink-P also achieves better average accuracy than the baseline systems. However, its “not linked” F1-score is lower than that of the Untangled intent count-baseline. We hypothesize that this could be attributed to errors in finding matches via the regular expressions in enriched untangled intents, where many of the “not linked” code-intent pairs are misclassified “linked”. We will examine this hypothesis as part of RQ2.

To determine whether the differences in average accuracy between the best performer (i.e., AutoCILink-ML) system and baseline systems are statistically significant or not, we employ the two tailed paired $t$-test. To show the soundness of choosing the two tailed paired $t$-test, we perform the Shapiro-Wilk normality test [119] with the null hypothesis that the performance of corresponding system is normally distributed. The test result for each system ($p > 0.05$) cannot reject the null hypothesis and the data of each system is normally distributed. For each significance test, our null hypothesis is: there is no performance difference between the two systems under comparison. Following [87], the result of significant test is interpreted as: (1) highly significant if $p < 0.01$; (2) significant if $0.01 \leq p < 0.05$; (3) moderately significant if $0.05 \leq p < 0.1$. Otherwise, the difference is statistically indistinguishable. The significance test results are shown in Table 4.3 (rows 1-5). Each row shows the $p$ value of AutoCILink-ML’s result compared to one baseline system’s result. We conclude from these results that AutoCILink-ML is more accurate than its peers with high or moderate statistical significance.

**RQ2. Which system is more accurate in linking changed code to untangled change intents, AutoCILink-ML or AutoCILink-P?**

The first two rows of Table 5.4 shows the results between the two variants of AutoCILink, namely AutoCILink-ML and AutoCILink-P. As we can see, AutoCILink-ML is considerably more accurate than AutoCILink-P, with average accuracies of 81.2% vs 74.6%. To determine whether this improvement is significant or not, we again em-
ploy a two tailed paired $t$-test on these systems. Row 6 in Table 4.4 shows that the improvement is highly significant. In addition, Table 4.4 shows that AutoCILink-ML outperforms AutoCILink-P by 4% in “linked” F1-score and by 22.2% in “not linked” F1-score.

To gain additional insights, we investigate the performance of AutoCILink-ML and AutoCILink-P in terms of their recall and precision in predicting “linked” and “not linked” instances, respectively. Results are shown in Table 4.4. It is interesting to see that although AutoCILink-P achieves better recall in predicting “linked” instances (90.8%) and better precision in predicting “not linked” instances (68.1%) than AutoCILink-ML, it results in much lower precision in predicting “linked” instances (78.4%) per project and recall (36.3%) in predicting “not linked” instances per project. Again, we hypothesize that this can be attributed to the “imprecision” of the regular expression based system (AutoCILink-P-R), which tends to identify code-intent links by searching for a match in the enriched untangled change intents, compromising its preciseness in pinpointing the actual links. To test this hypothesis, we evaluate the contribution of the regular expressions to AutoCILink-P’s performance. Specifically, we obtained results on the test set by running only “AutoCILink-P-R” (i.e., Step 1 of AutoCILink-P). These results are shown in the last row of Table 4.4. As we can see, in comparison to AutoCILink-ML, AutoCILink-P-R’s higher recall on the “linked” instances and its higher precision on the “not linked” instances provide suggestive evidence for our hypothesis.

<table>
<thead>
<tr>
<th>AutoCILink Model</th>
<th>Linked, Avg</th>
<th>Not Linked, Avg</th>
<th>Average Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>R</td>
<td>P</td>
<td>F1</td>
</tr>
<tr>
<td>ML</td>
<td>81.7</td>
<td>94.0</td>
<td>87.4</td>
</tr>
<tr>
<td>P</td>
<td>90.8</td>
<td>78.4</td>
<td>83.4</td>
</tr>
<tr>
<td>P-R</td>
<td>89.8</td>
<td>67.5</td>
<td>75.8</td>
</tr>
</tbody>
</table>
Table 4.5: Feature ablation results in average accuracy

<table>
<thead>
<tr>
<th>Iter</th>
<th>-Type 4</th>
<th>-Type 6</th>
<th>-Type 3</th>
<th>-Type 2</th>
<th>-Type 1</th>
<th>-Type 5</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Iter 1</td>
<td>72.3</td>
<td>79.4</td>
<td>79.5</td>
<td>80.4</td>
<td>80.4</td>
<td>80.9</td>
</tr>
<tr>
<td>Iter 2</td>
<td>75.8</td>
<td>78.2</td>
<td>78.9</td>
<td>80.0</td>
<td>80.4</td>
<td></td>
</tr>
<tr>
<td>Iter 3</td>
<td>74.1</td>
<td>77.1</td>
<td>78.9</td>
<td>79.7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Iter 4</td>
<td>75.9</td>
<td>79.6</td>
<td>79.7</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Iter 5</td>
<td>63.1</td>
<td>69.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 4.6: Paired t-test results on AutoCILink feature selection

<table>
<thead>
<tr>
<th>System 1</th>
<th>System 2</th>
<th>$p$ value</th>
</tr>
</thead>
<tbody>
<tr>
<td>AutoCILink-Type 4</td>
<td>AutoCILink-ML</td>
<td>&lt;0.0001</td>
</tr>
<tr>
<td>AutoCILink-Type 6</td>
<td>AutoCILink-ML</td>
<td>0.0397</td>
</tr>
</tbody>
</table>
RQ3. Which feature types have the largest impact on the performance of AutoCILink-ML?

Recall that AutoCILink-ML employs six types of features: Type 1: regular expression features; Type 2: vocabulary pair features; Type 3: vocabulary similarity features; Type 4: untangled change intent count features; Type 5: code import features; and Type 6: terms unmatched features. To understand which feature type(s) have the largest impact on the performance of AutoCILink-ML, we perform feature ablation experiments in which we remove the feature types from the system one-by-one.

We show the results of the ablation experiments in Table 4.4, where results are expressed in terms of average accuracy. The top line of the table shows what the system that uses all available features’ score would be if we removed just one of the six feature types. So to see how our system performs if we remove only the untangled change intent count features (Type 4), we would look at the first row of results under the column headed by -Type 4. The number here tells us that the resulting system’s average accuracy is 72.3%. Since AutoCILink-ML (when all feature types are used) achieves an accuracy of 81.2% (see Table 4.5), the removal of the untangled change intent count features costs the complete system 8.9% points in accuracy.

From row 1 of Table 4.5, we can see that removing Type 5 (code import features) yields a system with the best average accuracy in the presence of the remaining feature types in this row. For this reason, we permanently remove the Type 5 features from the system before we generate the results in row 2. We iteratively remove the feature type that yields a system with the best performance in this way until we get to the last line, where only one feature type is used to generate each result.

Since the feature type whose removal yields the best system is always the rightmost entry in a line, the order of column headings indicates the relative importance of the feature types, with the leftmost feature types being the most important to performance and the rightmost feature types being least important in the presence of other feature types. As we can see, the most important features are Type 4 and Type 6, as their removal results in a 18.1% and 12.2% drop in accuracy, respectively.
Table 4.7: Distribution of misclassified pairs of changed source code and enriched untangled change intents

<table>
<thead>
<tr>
<th></th>
<th>Percentage of misclassification</th>
<th>Misclassified as Linked</th>
<th>Misclassified as Not linked</th>
</tr>
</thead>
<tbody>
<tr>
<td>AutoCILink-ML</td>
<td>18.8%</td>
<td>617</td>
<td>175</td>
</tr>
</tbody>
</table>

We conduct the paired \( t \)-test to determine whether the removal of either of these two types of features yields a system that performs significantly worse than the system that employs all six types of features. Similar to RQ1, we conduct the Shapiro-Wilk test to show that the performance data yielded by the system with certain types of features is normally distributed. Results are shown in Table 4.5. Specifically, row 1 compares the system using all six types of features (System 2) with the system that uses all but the Type 4 features (System 1), whereas row 2 compares the system using all six types of features (System 2) with the system that uses all but the Type 6 features (System 1). As we can see from the results, the difference between the two systems in each of these two experiments is statistically significant.

**RQ4. What is the root cause of mistakes made by AutoCILink-ML?**

To shed light on how to improve the performance of AutoCILink-ML in future work, we performed a comprehensive error analysis on misclassified pairs of changed source code and untangled change intents by the two variants of AutoCILink system. Table 4.6 lists the percentage of misclassified instances and the number of test instances misclassified as “linked” and “not linked” by AutoCILink-ML. We notice that one typical error is the misclassification of a “linked” code-intent pair as “not linked” by AutoCILink-ML, which leads to relatively lower precision in predicting “not linked” pairs. Based on our analysis, the main reason for this type of misclassification is can be attributed to the inconsistent definitions/ambiguity of certain terms used in commit messages and their related documents and the same ones used in source code. For example, in the project androidkickstartr, one change intent untan-
gled from the commit message says: “remove the canonical name constants”, where “name” means “variable name”. However, this intent is erroneously linked to changed source code file that contains an entity “name” which means “by which a thing is known”. To reduce this kind of errors, word sense disambiguation would be helpful. In WordNet, English words are grouped into sets of synonyms called “synsets”. As mentioned in Section III-C-1, every meaning of a word is represented in a unique “synset”. In this case, “synsets” features can be added to ensure the unambiguous interpretation of the term and/or entity under its own linguistic context.

4.4. Summary

We proposed the AutoCILink system, which comprises a pattern-based approach and a learning-based approach, to address the novel task of automatically link changed source code files to untangled change intents. In experiments on a newly annotated corpus from the repositories of 19 open source projects with links between changed source code files and untangled change intents, we showed that AutoCILink outperformed all five baseline systems under comparison. AutoCILink-P achieves 83.4% F1-score on “linked” code-intent data, 40.2% F1-score on “not linked” data and an average accuracy of 74.6%. AutoCILink-ML further improves the performance and achieves 87.4% F1-score on “linked” data and 62.4% F1-score on “not linked” data, and an average accuracy of 81.2%.

4.5. Bibliographical Notes

The paper supporting the content described in this chapter were written in collaboration with the members at Southern Methodist University, Nanjing University, and University of Texas at Dallas.

• Liu, X., Huang, L., Li, C., and Ng, V. in Proceedings of the 34th IEEE International Conference on Software Maintenance and Evolution, 2018.

6 Extracted from changed source code identifier packageName
Chapter 5

TOWARDS AUTOMATED SOFTWARE CHANGE – IMPROVING API RECOMMENDATION

As a start, in Chapter 3 we discussed the source code dependency (i.e., call dependency, data sharing dependency and PDG), in which we learnt about the structural relations in source code. With this as a basis, in this chapter we present a way to help developers to choose the correct API for the next line of code. We thus speak of API recommendation. API recommendation is non-trivial because during daily software development, Application Programming Interfaces (APIs) are provided as functional building blocks to program software systems. APIs are classes, methods, and fields provided by the library’s designers [99] to enable developers to access the functionality of a code library. However, developers need to spend a lot of efforts on familiarizing themselves with the capabilities provided by a large number of APIs in the library and pick the correct API for development tasks. For instance, developers need to manually browse a long list of APIs to identify `BufferedWriter.write()`, the API that enables them to efficiently write to a file by buffering the characters in Java memory. As another example, developers have to choose from a list of 67 candidate member methods of `String` in the Java Development Kit (JDK) to identify the appropriate API for converting all of the corresponding characters to upper case (i.e., `String.toUpperCase()`). To address this challenge, many automated API recommendation approaches and tools have been proposed to relieve the burden of developers in understanding and locating APIs, either by taking advantage of API usage patterns [11,40,60,126] or by using statistical learning to recommend the next token [26,77,96,97]. For instance, Gralan uses a statistical language model for API recommendation that relies on features extracted from the preceding context (i.e., the code that has been written so far). The model was trained by collecting statistics on how often a candidate API co-occurs with the APIs in its preceding context [97].
Being a generative model, however, *Gralan* is sensitive to the presence of *overlapping* features and *irrelevant* features. Specifically, if two features encode overlapping information (e.g., two features are computed based on the same API in the preceding context), it will undesirably amplify the importance of this API in the prediction process, thus possibly harming model performance. Irrelevant features (i.e., features that are largely not predictive of the target API), too, could be harmful: while the statistics collected during the training process could to some extent indicate whether a feature is relevant, the multiplicative effect resulting from a large number of irrelevant features in a generative model could overwhelm the positive effect of the relevant features, again harming model performance. Hence, feature engineering is important when employing generative models. Unfortunately, as we will see in the next section, a number of features that *Gralan* employs are by design both overlapping and irrelevant.

More recently, *APIREC* [96], a state-of-the-art API recommendation approach, was proposed by *Gralan*’s authors. *APIREC* makes a key assumption: changes that serve the same higher-level intent of the developers will co-occur more frequently than non-related changes [96]. Hence, by leveraging the regularity and repetitiveness of software changes of a software system, *APIREC* can identify and focus on changes/features that are relevant to API recommendation, thereby reducing the impact of the feature irrelevance problem mentioned earlier. Nevertheless, the applicability of *APIREC* is severely limited by the large number of historical software change repositories it requires. Specifically, not only does it need to be *trained* on 471,730 changed source code files and 113,103 change commits, but it can only be *applied* to files with a similarly long change history.

Our goal in this chapter is to advance the state-of-the-art in API recommendation, specifically by improving the top-1 API recommendation accuracy. In view of the aforementioned limitations of *APIREC*, we desire an approach that does *not* rely on code change history. The design of our system, RecRank, is motivated by a key observation: while *Gralan* is unable to achieve a high top-1 recommendation accuracy, it achieves a reasonably high top-10 recommendation accuracy (73.4–80.6%).
Given this observation, we take the top-10 API candidates identified by Gralan as our starting point and re-rank these candidates so that the correct API surfaces to the top of the list. The question, then, is: how should we re-rank? Recall that a key weakness of Gralan concerns the use of a generative model, which is sensitive to the presence of overlapping and irrelevant features. RecRank is specifically designed to address this weakness. First, RecRank employs a discriminative re-ranker that is trained to re-rank Gralan’s top-10 candidate APIs. The key advantage of a discriminative approach (over a generative approach) is that the former can automatically discriminate relevant from irrelevant features (by assigning high weights to the relevant ones and low weights to the irrelevant ones). Second, we propose a novel kind of features for use in conjunction with our discriminative re-ranker, API usage path based features. These features partially address the feature irrelevance problem and can arguably better capture the linguistic topic of the program expressing the intention of the developer.

In summary, our contribution in this chapter lies in the proposal of RecRank, a novel discriminative ranking approach that employs a novel kind of features based on usage paths to automatically recommend top-1 APIs based on the top-10 API candidates suggested by Gralan.

5.1. Problem Statement and Motivating Examples

5.1.1. Problem Definition

The problem that we aim to solve with our proposed approach is the following:

**Problem Statement:** Improving the state-of-the-art performance of recommending API given its preceding API usage in source code.

Furthermore, this problem can be broken down into two phases, including generating top-10 API candidates and pick the correct one from the top-10 API candidates. In this dissertation, we focus on automatically generating top-10 API candidates and then automatically picking up the correct API. To accomplish this we investigate the ability by using two approaches: (i) an approach proposed in previous work (Gralan)
to generate top-10 API candidates for each recommendation point, and (ii) our proposed approach (RecRank) to find out the correct API from the top-10 candidates.

5.1.2. Motivating Example

We motivate the development of RecRank through the following example. A developer is making change by adding a software function to read text from a .txt file (“input.txt”) and write the processed text to another .txt file (“output.txt”). The code snippet is shown in Figure 5.1, in which the input text file “input.txt” is read using Java Development Kit (JDK) API BufferedReader (line 6) and written to “output.txt” using JDK API BufferedWriter (line 7). A while loop is used to iteratively read each line of the input text file (line 11). Now this developer needs to decide what API should be used in line 12 to write to the “output.txt” file. Modern Integrated Development Environment (IDE) tools, such as Eclipse, provide a list of APIs for developers to choose. This list of methods and fields is usually ranked in alphabetical order since it simply shows all member methods/fields of the calling API. Figure 5.2 shows that Eclipse recommends 16 APIs for line 12 in Figure 5.1. Note that these member methods and fields of the calling API BufferedWriter are not prioritized based on relevance: they are simply listed in alphabetical order.

Since the developer still cannot decide which API to choose from the list recommended by IDE tools, she would like to ask for help from Gralan. As aforementioned, Gralan ranks candidate APIs by the probabilities of the corresponding child graphs given a parent graph and its subgraphs. Specifically, it starts by building a set of API usage graphs (such as the one shown in Figure 5.4-a) of the code snippet in
Figure 5.2: An API recommendation example from Eclipse

Figure 5.4: Parent-child graph example
For each of the API usage graphs, *Gralan* extracts the corresponding parent graph and its subgraphs. These context graphs are then used in calculating the probability of each candidate API using Equation (5.1). However, not all context graphs are relevant to the recommendation point. In other words, not all context graphs implement the same linguistic topics as that of the recommendation point. For example, the recommendation point N in Figure 5.4-a implements the linguistic topic “write to output text file” with its context graph in the green rectangle, while its context graph in the blue rectangle implements the linguistic topic “read from input text file”. However, based on Equation (5.1) this context graph is considered as important as other context graphs: like other generative models, the one employed by *Gralan* merely multiplies the probabilities associated with the parent graph and all of its subgraphs.

Table 5.1 and 5.2 show a few examples of the parent graphs (i.e., $g$) and their corresponding child graphs (i.e., $C(g)$) for the code snippet in Figure 5.2 as well as the probability (i.e., score) of each child graph. The scores of the child graphs over all of the parent graphs are calculated and sorted. As we can see in Table 5.1 and 5.2, even though API *BufferedWriter.write* is the correct API for the recommendation point N, *Gralan* recommended *BufferedReader.close* since it has the highest score (i.e., 0.33). The main reason behind this miss is that *BufferedReader.close* co-occurred more frequently with the irrelevant context graph in the blue rectangle in Figure 5.4 which implements the linguistic topic “read from input text file” rather than the topic that corresponds to the developer’s intent, “write to output text file”. Note that this is just one example of an irrelevant feature employed by *Gralan*: because of the way parent graphs are generated for a recommendation point, many of them (as well as the subgraphs generated from them) are irrelevant. Together with the overlapping features, these irrelevant features could harm *Gralan’s* performance.

Then the developer decides to try a state-of-the-art approach, *APIREC* [96]. The key idea behind *APIREC* is to leverage the regularity and repetitiveness of API usage patterns learned from software change history. It assumes that the changes that serve the same higher-level intent will co-occur more frequently than unrelated
Table 5.1: Probability scores of candidate APIs (Part I)

<table>
<thead>
<tr>
<th>$g$</th>
<th>$C(g)$</th>
<th>Candidate API</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>FileReader.&lt;init&gt;, BufferedReader.&lt;init&gt;, BufferedReader.readLine, CONTROL.WHILE</td>
<td>FileReader.&lt;init&gt;, BufferedReader.&lt;init&gt;, BufferedReader.readLine, BufferedReader.close</td>
<td>BufferedReader.&lt;init&gt;, BufferedReader.readLine, BufferedReader.close</td>
<td>0.33</td>
</tr>
<tr>
<td></td>
<td>FileReader.&lt;init&gt;, CONTROL.WHILE, BufferedWriter.write</td>
<td>BufferedWriter.&lt;init&gt;, BufferedWriter.write</td>
<td>0.15</td>
</tr>
<tr>
<td></td>
<td>BufferedWriter.&lt;init&gt;, BufferedWriter.&lt;init&gt;, CONTROL.WHILE, BufferedWriter.&lt;init&gt;, BufferedWriter.&lt;init&gt;, BufferedWriter.close</td>
<td>BufferedWriter.&lt;init&gt;, BufferedWriter.&lt;init&gt;, BufferedWriter.&lt;init&gt;, BufferedWriter.close</td>
<td>0.25</td>
</tr>
<tr>
<td></td>
<td>BufferedWriter.&lt;init&gt;, CONTROL.WHILE, BufferedReader.&lt;init&gt;, BufferedReader.&lt;init&gt;, BufferedReader.close</td>
<td>BufferedReader.&lt;init&gt;, BufferedReader.&lt;init&gt;, BufferedReader.close</td>
<td>0.02</td>
</tr>
</tbody>
</table>

...
changes [96]. In other words, those APIs in the context graphs that have a higher frequency of source code change co-occurrence (and hence are assumed to have a higher predictive power in API recommendation) will be given more importance in the API recommendation process. For each candidate API, APIREC first computes a score based on the change history, and then adds the resulting score to the one computed by Gralan to form the final score.

Not all changes are applicable, however, since some of them could be specific to a historical project and could therefore incur noise in the change patterns. In the example in Figure 5.4-a, after analyzing a large number of historical fine-grained changes, APIREC learned that BufferedWriter.write changed with BufferedReader.<init> with a probability of 0.3 and that it changed with FileReader.<init> with a probability of 0.05. Meanwhile, it also learned that BufferedReader.close changed with BufferedReader.<init> with a probability of 0.7 and that it changed with FileReader.<init> with a probability of 0.5. Hence, using only the code change history, APIREC will select the wrong API, BufferedReader.close, for the given recommendation point since its probability of change co-occurrence ($0.7 \times 0.5 = 0.35$) is larger than that of BufferedWriter.write ($0.3 \times 0.05 = 0.015$). In other words, using the code change history, APIREC cannot override Gralan’s erroneous recommendation for this recommendation point. In ad-

### Table 5.2: Probability scores of candidate APIs (Part II)

<table>
<thead>
<tr>
<th>CONTROL.WHILE, BufferedReader.close</th>
<th>BufferedReader.&lt;init&gt;</th>
<th>0.1</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONTROL.WHILE, BufferedWriter.write</td>
<td>BufferedWriter.&lt;init&gt;</td>
<td>0.05</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
| ...                                  | ...                    | ...

92
dition, APIREC requires a long source code change history of each subject project, which limits its applicability to scenarios where long code change history is unavailable or inaccessible.

To address the challenge of accurate API recommendation, we propose RecRank, which recommends APIs based on the API usage paths generated from API usage graphs. An API usage path (henceforth usage path) is generated to represent a data/control flow sequence of APIs that can arguably better encode the intention of the developer. Using discriminative learning in combination with usage paths as features, higher weights can be learned for usage paths that are more relevant and coherent to the given recommendation point, thereby reducing the noise possibly introduced by irrelevant or incoherent usage paths. For example, in Figure 5.4-a we extract one usage path \([FileWriter.<init> \rightarrow BufferedWriter.<init> \rightarrow \text{(recommendation point)}]\) from the code snippet in lines 7−12 in Figure 5.1. This usage path implements the linguistic topic “write to output text file”, which is weighted higher than other usage paths extracted in Figure 5.4-a. Since RecRank seeks to improve the accuracy of recommending the top-1 API, it could save the developer’s time and effort in manually selecting the correct API from multiple candidates. Note that RecRank seeks to achieve this goal without mining and using long fine-grained code change history.

5.2. Approach Description

Since RecRank is built upon the top-10 API candidates suggested by Gralan, in this section we first provide an overview of phase 1−Gralan. Second, as in phase 2 we present a novel approach to API recommendation, RecRank, which operates by re-ranking the top-10 candidate APIs recommended by Gralan for each recommendation point using a learned discriminative re-ranker in combination with our usage path-based features.

5.2.1. Phase 1 - Graph-based Generative API Recommendation (Gralan)
As mentioned before, given a recommendation point, Gralan recommends an API using its preceding context (i.e., the code that has been written so far).\(^1\) Gralan encodes the preceding context as a set of API usage graphs. In an API usage graph, each node is an API used in a method call, operator overloading, field access or branching (e.g., if, while, for, etc.). All API nodes are connected by directed edges. Each edge represents a data flow dependency (i.e., overloading operator, method calls, and field accesses) or a control flow dependency (i.e., condition and repetition) between two APIs.

An example of an API usage graph is shown in Figure 5.4-a, where node N is the recommendation point. The corresponding context graph (i.e., the graph that encodes the context in which N occurs) is shown in Figure 5.4-b. As can be seen, this context graph is created by removing node N as well as all of its incoming and outgoing edges. Throughout the chapter, if two graphs (e.g., the ones shown in Figure 5.4) have a parent-child relationship, we will refer to the one without the recommendation point as the parent graph and the one with the recommendation point as the corresponding child graph.

As mentioned before, Gralan uses the parent graph for predicting the API at the recommendation point. One way to make use of the parent graph is to estimate the probability that a candidate API co-occurs with the parent graph in the training data. The higher the co-occurrence probability is, the more likely that the candidate API is the correct API. However, a parent graph (such as the one shown in Figure 5.4-b) could be fairly complex. Complex parent graphs could yield a data sparsity problem: the more complex a parent graph is, the less likely it will be seen in the training data. To alleviate data sparsity, Gralan also makes use of all the (non-empty) subgraphs of the parent graph in the API prediction process. For instance, from the parent graph in Figure 5.4-b, we can extract subgraphs with one API (e.g., CONTROL.WHILE), subgraphs with two APIs (e.g., [FileWriter.<init>, CONTROL.WHILE]), subgraphs with three APIs (e.g., [FileWriter.<init>, BufferedReader.<init>, CONTROL.WHILE]),

\(^1\)The reason that only the preceding context is used is to mimic the realistic situation that when an API is to be recommended to a developer, only the code that has been written so far is available.
and subgraphs with four APIs (e.g., `[FileWriter.<init>, BufferedReader.<init>, BufferedReader.readLine, CONTROL.WHILE]`).

Specifically, given a parent graph \( g \) and subgraphs \( g_1, \ldots, g_n \) of \( g \), Gralan computes the probability of a child graph, \( C(g) \), which is created by filling the recommendation point with a candidate API, using Bayesian statistical inference as follows:

\[
\log(Pr(C(g)|g_1, g_2, \ldots, g_n, g)) \\
\propto \log(Pr(g_1|C(g)) \ldots Pr(g_n|C(g))Pr(C(g))) \\
= \sum_{j=1}^{n} \log(\#methods(g_j, C(g)) + \alpha) \\
+ \log(\#methods(g, C(g))) \\
- (n - 1) \log(\#methods(C(g)) + \alpha \#methods) \\
- \log(\#methods(g))
\]  

(5.1)

where the expression in the second line is obtained using Bayes rule, and the third line shows how the probabilities in the second line can be estimated. Specifically, \( \#methods(g, C(g)) \) is the number of times \( g \) appears as the parent of \( C(g) \) in the training data, \( \#methods(g) \) is the number of times \( g \) appears in the training data, and \( \#methods \) is the total number of methods in the training data.\(^2\) To avoid floating underflow, Logarithm (log) is applied to all the probabilities in the equation. To assign non-zero probabilities to events not seen in the training data, a smoothing factor (i.e., \( \alpha \)) is used. Note that each of the graphs being conditioned on in Equation 5.1 (i.e., \( g, g_1, \ldots, g_n \)) can be viewed as a feature used by Gralan in the recommendation process. Because the \( g_i \)'s are subgraphs of \( g \), these features are by design overlapping, which could harm the performance of a generative model like Gralan, as noted in the introduction.

There is another caveat. Recall that Figure 5.4-a only shows one of the many API usage graphs that Gralan generates for the recommendation point \( N \). The exact number of API usage graphs that Gralan generates for a recommendation point

\(^2\)Our training data is composed of the set of API usage graphs generated from all the methods in the source code collected from 1385 open source projects (see Section 5.3.1 for details).
depends on a parameter, \(d\), which specifies the maximum distance between the recommendation point and any of the nodes in an API usage graph. For instance, if \(d=3\), \textit{Gralan} will generate \textit{all} API usage graphs that can possibly be generated by including any subset of nodes whose distance is no larger than 3 from the recommendation point. For each of these API usage graphs, \textit{Gralan} generates the corresponding parent graph. Given each parent graph \(g\) (and its subgraphs \(g_1, \ldots, g_n\)), \textit{Gralan} computes the probability of each child graph \(C(g)\) using Equation (5.1). The candidate API that corresponds to the most probable child graph over all the parent graphs will be the API recommended by \textit{Gralan} for a given recommendation point.

5.2.2. Phase 2 - Discriminative Re-ranking for API Recommendation (RecRank)

Before describing RecRank, we present two re-ranking systems that could help the reader better understand the power of discriminative re-ranking. The first re-ranking system is trained using the Naïve Bayes (NB) generative model on our usage path-based features. The second re-ranking system is a discriminative classifier trained using the support vector machine learner (henceforth SVC) on our usage path-based features. The performance difference between the NB system and the SVC system can shed lights on the relative effectiveness of generative models, which are sensitive to the presence of overlapping and irrelevant features, and discriminative models, which are robust to such features. Note that the SVC system is one step closer to RecRank than the NB system in the sense that both SVC and RecRank are discriminative in nature: the primary difference between them lies in the fact that SVC recasts the API recommendation task as a classification task whereas RecRank recasts the task as a ranking task. The performance difference between them can therefore shed lights on the relative effectiveness of classification and ranking. We will discuss the differences between classification and ranking later in this section.

5.2.2.1. Phase 2.1 - NB

In NB, we employ the Naïve Bayes learning algorithm implemented in scikit-learn Python library to train a binary classifier to classify whether a given recommended
API is the correct API at the recommendation point (i.e., a “hit”) or not (i.e., a “miss”). Recall that NB employs the following generative model:

\[ P(c|\text{candidate API}) = P(c) \prod_{i=1}^{n} P(f_i|c) \]

where \( c \) is the class (which in our case is either “hit” or “miss”), and each \( f_i \) corresponds to a usage path-based feature extracted for the candidate API under consideration. As can be seen, the NB generative model assumes that the values of the usage path-based features are conditionally independent of each other given the class. Each of the probabilities in the generative model can be estimated using maximum likelihood estimation from the training data. Specifically, \( P(c) \) is the fraction of instances in the training set that are labeled as \( c \). \( P(f_i|c) \) is the fraction of training instances labeled as \( c \) that contain feature \( f_i \).

We employ the trained NB model to re-rank the top-10 candidate APIs suggested by Gralan as follows. Since the model computes for each candidate API the probability that it is a “hit”, we rank the candidate APIs using their associated probabilities, where higher probabilities correspond to higher ranks.

Next, we discuss how the training instances are created and how the usage path-based features are extracted for each training instance.

*Creating training instances.* For each API recommendation point in the training set, we create one training instance for each of the 10 API candidates recommended by Gralan, labeling an instance as “hit” or “miss” depending on whether the corresponding candidate API is the correct API for the recommendation point under consideration. Each instance is represented using a set of usage path-based features, each of which corresponds to an usage path. This set of usage paths is the union of the usage paths extracted from each of the API usage graphs created for the given recommendation point. Below we define usage paths.

Each usage path is extracted from an API usage graph and is defined by three constraints. First, a usage path is formed by a sequence of APIs connected by directed data and/or control flow edges. Second, the APIs in a usage path are sequentially connected/listed in API usage order with one entry API and one exit API. Finally,
each usage path contains a candidate API (one of the 10 candidate APIs recommended by *Gralan*) that appears either at the end (where the directed flow ends) or at the beginning (where the directed flow starts) of the path. Usage paths of various lengths could be generated from an API usage graph. The length of a usage path is between 2 and the threshold parameter $d$, which determines the maximum distance between any node and the recommendation point in the graph (defined in Section 5.1). For example, 13 usage paths can be generated from the API usage graph in Figure 5.4-a, such as: $[\text{FileReader.} <\text{init}> \rightarrow \text{BufferedReader.} <\text{init}> \rightarrow \text{BufferedReader.} \text{readLine} \rightarrow \text{CONTROL. WHILE} \rightarrow \text{(candidate API)}]$.

To model different data/control flow in usage paths, we have designed different types of usage path features, as described below.

A **forward usage path feature** is created from a usage path in which the APIs in the path are connected by edges in the point-forward direction with the candidate API appearing at the end of the path. A forward data/control flow towards the recommendation point usually implies that the API at the recommendation point “consumes” the data passed by data/control flow. As an example, consider the API usage graph in Figure 5.4. From this graph, we can create a **forward usage path feature** from the path $[\text{FileReader.} <\text{init}> \rightarrow \text{BufferedReader.} <\text{init}> \rightarrow \text{BufferedReader.} \text{readLine} \rightarrow \text{CONTROL. WHILE} \rightarrow \text{(candidate API)}]$. We create forward usage path features from paths of different lengths, where the length of a path is defined as the number of APIs involved in the path. For instance, the path $[\text{CONTROL. WHILE} \rightarrow \text{(candidate API)}]$ is of length 2. We consider all paths of up to length $d$.

A **backward usage path feature** is created from a usage path that starts with the candidate API, and in which the APIs are connected by edges with a point-backward direction. A backward data/control flow from the recommendation point usually implies that the API at the recommendation point “produces” or “returns” the data to be delivered to the APIs in the back track. In Figure 5.4-a, we can create a **backward usage path feature** from the path $[\text{CONTROL. WHILE} \leftarrow \text{(candidate API)}]$. Similar to forward usage path features, backward usage path features are generated from paths of different lengths.
In addition, we derive **fuzzy usage path features** from the *forward usage path features* and the *backward usage path features*. To motivate fuzzy usage path features, we note the correspondence between these usage paths and the word $n$-grams used in natural language processing (NLP). Specifically, the sequence of APIs in a forward/backward usage path is reminiscent of the sequence of words in a word $n$-gram. NLP researchers have noted a weakness of using word $n$-grams as features in natural language learning: if $n$ is large, the resulting $n$-grams will suffer from data sparsity; and if $n$ is small, the $n$-grams will fail to capture longer-distance dependencies. To address this weakness, they have proposed the use of *skipgrams*, in which they allow all but the first word and the last word in an $n$-gram to match any words. For instance, given the word $n$-gram “I am a boy”, one can generate a skipgram “I * * boy”, where each wildcard * can match any word. This provides generalization of the original $n$-gram (and therefore addresses data sparsity) but at the same time captures the relationship between non-adjacent words (in this case “I” and “boy”).

**Fuzzy usage path features** are motivated by skipgrams. Specifically, a *fuzzy usage path feature* is created from a forward/backward usage path feature by replacing all but the entry API and the exit API in the corresponding path with wildcards. Returning to the example in Figure 5.4-a, `[FileReader.<init> → * → * → CONTROL.WHILE → (candidate API)]` is a fuzzy usage path feature with two “fuzzy” APIs in the path (represented as “*”). As in skipgrams, wildcards (i.e., fuzzy APIs) can only appear in the middle of a fuzzy usage path. As with skipgrams, the goal of these fuzzy path features is to provide generalizations of the forward/backward usage path features.

In comparison to the parent graphs and subgraphs that *Gralan* uses as features, our usage paths are arguably more relevant to API recommendation. First, since each usage path has to begin or end with a candidate API, it ensures that the path contains an API that is immediately adjacent to the candidate API, thereby increasing its relevance for API prediction. In contrast, a subgraph employed by *Gralan* may not contain any nodes that are adjacent to the recommendation point, thus possibly making it less relevant for API prediction. Second, from the example in Figure 5.4,
each context graph can potentially contain more than one linguistic topic (e.g., both read and write to a file). On the other hand, a usage path can typically allow us to focus on just one linguistic topic. This is especially important when it comes to discriminative learning: a discriminative learner can assign high weights to those features that encode the intended linguistic topic and low weights to those features that do not. If context graphs encoding multiple linguistic topics were used as features, the learner could find it difficult to decide whether high or low weights should be assigned to such features. Note that the computation of these usage path features can be done offline (i.e., during training) with the resulting values stored in a database. During testing, their values can simply be retrieved from the database.

A final issue that we have eluded so far concerns how we obtain *Gralan’s* top-10 candidate APIs on the training set. Recall that we create one training instance from each of *Gralan’s* top-10 candidate APIs. This means that before we can create training instances, we need to produce *Gralan’s* top-10 candidate APIs on the training set. We do so using 5-fold cross validation on the training set: we partition the training set randomly into five folds of roughly equal sizes. In each fold experiment, we train *Gralan* on four folds and applying the trained *Gralan* to generate the top-10 candidate APIs on the remaining fold. We repeat this five times, each time generating top-10 candidates on a different fold.

**Applying the NB classifier.** Test instances are created in the same way as the training instances. Specifically, we create one test instance from each of *Gralan’s* top-10 candidate APIs. This means that before we create test instances, we need to produce *Gralan’s* top-10 candidate APIs for each recommendation point in the test set. To do so, we train *Gralan* on the entire training set and apply the trained *Gralan* to generate top-10 candidate APIs on the test set.

As mentioned before, the resulting NB classifier can be used to compute the probability that each candidate API is a “hit” for a recommendation point. These probabilities are then used to re-rank the 10 candidate APIs.

**5.2.2.2. Phase 2.2 - SVC**
Our second re-ranking system, SVC, is a discriminative classifier trained using the SVM learning algorithm with a linear kernel, as implemented in the libSVM software package [38]. As in NB, we first use cross validation on the training set to produce Gralan’s top-10 candidate APIs on the training set, and then create one training instance from each of the 10 candidate APIs. Each training instance in SVC is represented using the same set of usage path-based features as in NB. The only difference lies in the value of each feature. As NB is generative, each feature is conditioned on the class. In contrast, SVC is discriminative, so we desire that the value of a feature provides some indication of how useful it is. Specifically, we desire that higher feature values imply more relevant features. To this end, we compute the value of a feature as follows. First, we count the number of times the corresponding usage path appears in the training set (call this number $a$). Second, we count the number of times the path appears in the training set after removing from it the candidate API (call this number $b$). Finally, we set the feature value to $\frac{b}{a}$. In other words, the more often the candidate API co-occurs with the rest of the path in the training set, the larger the feature value is. As in NB, the values of these usage path features can be computed and stored in a database during training, and they can simply be retrieved from the database during testing.

Training the SVC classifier. Given the training instances, the SVM learner learns a maximum margin hyperplane that minimizes the training error (i.e., the error of the hyperplane in classifying the training instances). A hyperplane is defined by a set of weights, each of which is associated with exactly one feature. In other words, the SVM learner learns a set of feature weights that minimizes training error, specifically by associating larger absolute weights with relevant features and lower absolute weights with irrelevant features. This distinguishes a discriminative learner from a generative model such as NB.

Applying the SVC classifier. After training, the resulting hyperplane can be used to classify the test instances, which are created in the same way as the training instances. As in NB, Gralan’s top-10 candidates on the test set are obtained by training Gralan on the entire training set and applying the trained Gralan on the
test set. We re-rank the top-10 candidate APIs based on their distances from the hyperplane. Specifically, the candidate API on the “hit” side of the hyperplane that is farthest away from the hyperplane receives the highest rank, whereas the one on the “miss” side of the hyperplane that is farthest away from the hyperplane receives the lowest rank.

5.2.2.3. Phase 2.3 - RecRank

Next, we describe RecRank, which differs from SVC in one respect: SVC classifies candidate APIs, whereas RecRank ranks candidate APIs. To understand the difference between classification and ranking, we first note that API recommendation is inherently a ranking task: its goal is to compare/rank candidate APIs and pick the best (i.e., highest-ranked) candidate API for a given recommendation point. When applying SVC, we essentially recast API recommendation as a classification task, where each candidate API is classified (as “hit” or “miss”) independently of other candidate APIs. In other words, SVC does not compare candidate APIs against each other, and without such comparisons, it fails to determine which candidate API is the best. In contrast, the goal of ranking is precisely to compare candidate APIs by imposing a ranking on them.

Training RecRank. RecRank trains an SVM ranker using the linear-kernel ranker learning algorithm implemented in SVM\textsuperscript{rank} [62]. The training instances (and the features that represent each training instance) are created in the same way as in SVC. The resulting training instances are then grouped into different ranking problems. Specifically, each ranking problem corresponds to exactly one recommendation point and is composed of the 10 training instances corresponding to the top-10 candidate APIs for this recommendation point. The goal of the ranker training procedure is to learn a hyperplane (by adjusting the feature weights) to minimize the number of violations of pairwise ranking in the training set. Specifically, a violation occurs if a training instance labeled as “hit” is ranked below a training instance labeled as “miss” by the ranker.
Applying RecRank. After training, the ranker can be used to directly rank the top-10 candidate APIs for each recommendation point in the test set. Specifically, the ranker assigns each candidate API a value, based on which a ranking can be imposed on the candidate APIs. RecRank then recommends the candidate API that has the highest rank.

5.3. Empirical Evaluation

This section presents the experiment results to answer our research questions of improving API recommendation.

5.3.1. Experiment Setup

Datasets. We collected a large dataset of 1385 Java projects from GitHub for training API recommendation systems and another eight for evaluation. Statistics of this dataset are shown in Table 5.3. In order to obtain high quality API usage graphs, we follow previous work [97]: we filter out the projects that are not parsable, experimental or toy programs. Also, we use only the latest snapshot of each project. For generalization purposes, we focus solely on Java Development Kit (JDK) APIs. To facilitate comparison with previous work, the eight projects in our evaluation set are the same as those used to evaluate APIREC, a state-of-the-art API recommendation system [96]. Training and test recommendation points are created from these projects in the same way as in previous work [96, 97]: except for the first two APIs in each method, we create one recommendation point for each API.

Evaluation Measures. We employ two evaluation measures, top-1 accuracy and mean reciprocal rank (MRR) [133]. Top-1 accuracy is a measure used in previous work on API recommendation [97]. For each API recommendation point in the test set, if the top-1 API candidate returned by a system is the correct API at the recommendation point, we count it as a “hit”. The top-1 accuracy is the ratio of the total number of hits to the total number of recommendation points. MRR is an evaluation measure commonly used in information retrieval to evaluate search results. Like top-1 accuracy, a score of 1 is given to a recommendation point for which the top-1 candidate
Table 5.3: Dataset statistics

<table>
<thead>
<tr>
<th></th>
<th>Training</th>
<th>Test</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total projects</td>
<td>1385</td>
<td>8</td>
</tr>
<tr>
<td>Total classes</td>
<td>138,791</td>
<td>8,621</td>
</tr>
<tr>
<td>Total methods</td>
<td>732,645</td>
<td>38,036</td>
</tr>
<tr>
<td>Total distinctive JDK API elements</td>
<td>18,166</td>
<td>7,272</td>
</tr>
<tr>
<td>Total recommendation points</td>
<td>70,377</td>
<td>11,872</td>
</tr>
<tr>
<td>Average features per API candidate</td>
<td>30</td>
<td>24</td>
</tr>
</tbody>
</table>

is the correct API. Unlike top-1 accuracy, where a system is not rewarded at all if the correct API is not the top-1 candidate API, MRR partially rewards a system as follows: a score of $\frac{1}{r}$ is given to a recommendation point if the correct API appears in rank $r$. In other words, the (partial) reward is inversely proportional to the rank of the correct API. MRR then averages the scores over the recommendation points in the test set. Thus, MRR can be viewed as a relaxed version of top-1 accuracy that partially rewards a system where the correct API is not the top-1 candidate. Since we are re-ranking Gralan’s top-10 candidate APIs, recommendation points where the correct API is not in Gralan’s top-10 will receive a score of 0.

**Baseline Systems.** We employ two baseline systems, neither of which is publicly available. As our first baseline, we employ APIREC. The APIREC results reported in this chapter are taken verbatim from the original APIREC paper [96].

As our second baseline, we employ Gralan. Since NB, SVC, and RecRank are all built upon Gralan’s top-10 candidate APIs, we re-implement Gralan, following the steps mentioned in Section 5.2.1. Specifically, we first build the API usage graphs from the collected 1385 open source projects in the training set. Then, following

---

[3] The reason we did not re-implement APIREC is that the significant large historical change repository dataset (i.e., 113,103 change commits and 471,730 changed source code files according to Nguyen et al. [96]) is hard to acquire.
Table 5.4: Re-implemented and original \textit{Gralan} results

<table>
<thead>
<tr>
<th>Project</th>
<th>Top-1 Accuracy</th>
<th></th>
<th></th>
<th>Top-10 Accuracy</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Dupli-</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>cated</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>\textit{Gralan}</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>\textit{Origin}</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>\textit{Gralan}</td>
<td></td>
<td></td>
<td>Error</td>
<td></td>
<td></td>
</tr>
<tr>
<td>antlr</td>
<td>38.3</td>
<td>26.0</td>
<td>+12.3</td>
<td>76.5</td>
<td>79.0</td>
<td>-2.5</td>
</tr>
<tr>
<td>Galaxy</td>
<td>22.4</td>
<td>22.0</td>
<td>+0.4</td>
<td>80.6</td>
<td>80.0</td>
<td>+0.6</td>
</tr>
<tr>
<td>Froyo-</td>
<td>25.5</td>
<td>46.0</td>
<td>-20.5</td>
<td>73.9</td>
<td>81.0</td>
<td>-7.1</td>
</tr>
<tr>
<td>Email</td>
<td>Grid-</td>
<td>31.2</td>
<td>+5.2</td>
<td>76.9</td>
<td>85.0</td>
<td>-8.1</td>
</tr>
<tr>
<td>Sphere</td>
<td>Itext</td>
<td>24.7</td>
<td>-8.3</td>
<td>80.5</td>
<td>76.0</td>
<td>+4.5</td>
</tr>
<tr>
<td>jGit</td>
<td>33.6</td>
<td>20.0</td>
<td>+11.6</td>
<td>77.1</td>
<td>74.0</td>
<td>+3.1</td>
</tr>
<tr>
<td>log4j</td>
<td>28.0</td>
<td>29.0</td>
<td>-1.0</td>
<td>75.2</td>
<td>74.0</td>
<td>+1.2</td>
</tr>
<tr>
<td>spring</td>
<td>30.2</td>
<td>28.0</td>
<td>+2.2</td>
<td>73.4</td>
<td>67.0</td>
<td>+6.4</td>
</tr>
</tbody>
</table>
Nguyen et al. [97], for each API usage graph we simulate the API recommendation process by predicting each API given its preceding context. We set the parameter $d$ to 3, meaning that only the context graphs involving the one, two or three APIs preceding a recommendation point are considered. The reason for setting $d$ to 3 is that according to Nguyen et al. [97], when $d=3$, the top-10 accuracy achieved by Gralan (86.0%) is close to the best accuracy (87.1%).

Table 5.4 compares the original Gralan results [96] with our duplicated/re-implemented Gralan results on the same eight subject projects. As can be seen, duplicated Gralan achieves better or comparable top-1 and top-10 accuracies than original Gralan results across all projects except Froyo-Email (top-1). Note that a strict comparison is not possible owing to the fact that original Gralan and duplicated Gralan were trained on different projects.\footnote{The list of projects used to train original Gralan is not revealed by the authors.}

Evaluation Settings. For NB, SVC and RecRank, we use the 1385 projects in the training set for model training. We conduct an 8-fold cross validation on the 8 projects in the test set as follows. In each fold experiment, we hold out exactly one project for testing and use the remaining 7 projects for development (i.e., parameter tuning). We repeat this experiment 8 times, each time choosing a different project as our held-out test set. For parameter tuning, we tune NB’s Laplace smoothing parameter $\alpha$ as well as libSVM and SVM$^{rank}$’s regularization parameter $C$ to maximize the top-1 accuracy on the development set. We limit the length an usage path based feature to no more than 4.

5.3.2. Experimental Results

This section empirically answers our research questions.

RQ1. How accurate do RecRank, NB, and SVC recommend APIs in comparison to the two baselines?

Results of NB, SVC, RecRank and the two baselines, APIREC and duplicated Gralan (a.k.a., D-Gralan), expressed in terms of per-project and overall top-1 accuracy and MRR, are shown in Table 5.5 and 5.6. As we can see, our proposed
Table 5.5: Evaluation results of API recommendation systems (Part I)

<table>
<thead>
<tr>
<th></th>
<th>System</th>
<th>Overall</th>
<th>antlr</th>
<th>Galaxy</th>
<th>Froyo-Email</th>
<th>Grid-Sphere</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>APIREC</td>
<td>59.5</td>
<td>57.0</td>
<td>56.0</td>
<td>60.0</td>
<td>58.0</td>
</tr>
<tr>
<td>2</td>
<td>D-Gralan</td>
<td>29.5***</td>
<td>38.3</td>
<td>22.4</td>
<td>25.5</td>
<td>31.2</td>
</tr>
<tr>
<td>3</td>
<td>NB</td>
<td>34.8***</td>
<td>45.1</td>
<td>37.3</td>
<td>35.3</td>
<td>38.3</td>
</tr>
<tr>
<td>4</td>
<td>SVC</td>
<td>59.6*</td>
<td>60.1</td>
<td>61.2</td>
<td>51.1</td>
<td>58.9</td>
</tr>
<tr>
<td>5</td>
<td>RecRank</td>
<td>64.8</td>
<td>69.4</td>
<td>72.4</td>
<td>63.5</td>
<td>67.6</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>System</th>
<th>MRR</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>APIREC</td>
<td>–</td>
</tr>
<tr>
<td>2</td>
<td>D-Gralan</td>
<td>0.27***</td>
</tr>
<tr>
<td>3</td>
<td>NB</td>
<td>0.60***</td>
</tr>
<tr>
<td>4</td>
<td>SVC</td>
<td>0.69*</td>
</tr>
<tr>
<td>5</td>
<td>RecRank</td>
<td>0.70</td>
</tr>
</tbody>
</table>
Table 5.6: Evaluation results of API recommendation systems (Part II)

<table>
<thead>
<tr>
<th>System</th>
<th>Top-1 Accuracy</th>
<th>MRR</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Itext</td>
<td>jGit</td>
</tr>
<tr>
<td>1 APIREC</td>
<td>44.0</td>
<td>54.0</td>
</tr>
<tr>
<td>2 D-Gralan</td>
<td>24.7</td>
<td>33.6</td>
</tr>
<tr>
<td>3 NB</td>
<td>29.4</td>
<td>38.2</td>
</tr>
<tr>
<td>4 SVC</td>
<td>57.3</td>
<td>56.1</td>
</tr>
<tr>
<td>5 RecRank</td>
<td>67.7</td>
<td>67.4</td>
</tr>
</tbody>
</table>
approaches (rows 3–5) outperform D-Gralan (row 2) in both top-1 accuracy and MRR across all subject projects. In particular, RecRank is the best performer in terms of both measures, achieving better top-1 accuracy than APIREC in all eight projects.\footnote{MRR results are missing for APIREC because they are not reported in the original paper.}

We further make several interesting observations. First, the proposed learning-based approaches (NB, SVC, and RecRank) achieve better top-1 accuracy and MRR than D-Gralan: top-1 accuracy improves by 0.7–50\% and MRR improves by 0.13–0.49. Compared to APIREC’s top-1 accuracy (59.5\%), SVC and RecRank achieve comparable or better results (i.e., 59.6 and 64.8\% respectively). Encouragingly, RecRank improves the state-of-the-art top-1 accuracies across all eight subject projects by 1.7–23.7\%.

To determine whether the improvements in overall top-1 accuracy and overall MRR between RecRank and other approaches are statistically significant or not, we conduct the Wilcoxon rank-sum test. Following Miller\cite{87}, the result of a significance test can be interpreted as follows. The performance difference between the two systems under comparison is (1) \textit{highly significant} if the null hypothesis (i.e., there is no performance difference between the two systems) can be rejected at the 0.01 level (represented as “***” in the table); (2) \textit{significant} if it can be rejected at the 0.05 level (represented as “**”); and (3) \textit{moderately significant} if it can be rejected at the 0.1 level (represented as “*”). Otherwise, the difference is \textit{statistically indistinguishable}. As can be seen in Table 5.5 and 5.6, RecRank is either highly or moderately significantly better than other systems.\footnote{Significance tests cannot be conducted on APIREC because we do not have its output.}

To evaluate the amount of performance difference between RecRank and each of the other approaches, we compute Cliff’s delta\cite{29}, a non-parametric effect size measure. Results show that in each case the delta value is greater than 0.474, which, according to Romano \textit{et al.}\cite{114}, implies a large effect size.

\textbf{RQ2. How effective are usage path features for API recommendation compared with context graphs?}
Table 5.7: Evaluation results for different model-feature combinations

<table>
<thead>
<tr>
<th>Combination</th>
<th>Overall Top-1 Accuracy</th>
<th>Overall MRR</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 RecRank+E</td>
<td>64.8</td>
<td>0.70</td>
</tr>
<tr>
<td>2 RecRank+C</td>
<td>36.6***</td>
<td>0.58***</td>
</tr>
<tr>
<td>3 SVC+E</td>
<td>59.6</td>
<td>0.69</td>
</tr>
<tr>
<td>4 SVC+C</td>
<td>25.4***</td>
<td>0.58***</td>
</tr>
<tr>
<td>5 NB+E</td>
<td>34.8</td>
<td>0.60</td>
</tr>
<tr>
<td>6 NB+C</td>
<td>16.1***</td>
<td>0.47***</td>
</tr>
<tr>
<td>7 D-Gralan+E</td>
<td>24.1***</td>
<td>0.25***</td>
</tr>
<tr>
<td>8 D-Gralan+C</td>
<td>29.5</td>
<td>0.27</td>
</tr>
</tbody>
</table>

To compare the effectiveness of these two types of features, we employ them to train four approaches: RecRank, NB, SVC and D-Gralan. This results in the eight combinations shown in Table 5.7. For instance, RecRank+E is the variant of RecRank trained using the usage path features, whereas RecRank+C is the variant of RecRank trained using context graphs. Note that the two variants within each of the four approaches differ only with respect to the feature set. In particular, the value of a feature is computed in the same way in the two variants of an approach. For instance, the value of a feature in RecRank+C is computed in the same way as that in RecRank+E, which was described in Section 5.2.2.1.

As can be seen in Table 5.7, SVC, and RecRank, the E variant is highly significantly better than the C variant in terms of both top-1 accuracy and MRR with a large effect size. These results provide suggestive evidence that the usage path features are considerably more effective than the context graph-based features for both discriminative models (SVC and RecRank) and the NB generative model. The only exception is Gralan, where its C variant is highly significantly better than its E variant. We speculate that context graphs were specifically designed by their original authors so that they could work well when used in conjunction with Gralan’s
generative model, but additional experiments are needed to determine the reason.

**RQ3.** How effective are different classes of usage path features for API recommendation?

To answer this question, we divide our usage path features into 12 groups based on (1) whether the path is forward or backward; (2) whether the path contains fuzzy APIs or not; and (3) the length of the path, which could be 2, 3 or 4 (recall that we limit the length to no more than 4 in Section 5.3.1). To determine the contribution of each of these 12 groups of features to RecRank’s performance, we conduct ablation experiments, where in each ablation experiment, we re-train RecRank by leaving out one or more of the 12 feature types and measure the performance of the re-trained RecRank on the test projects. Intuitively, the larger the drop in performance is in an ablation experiment, the more important the missing feature group(s) are as far as performance is concerned.

Ablation results are shown in Table 5.8. For ease of comparison, we show in row 1 the results of RecRank when all usage path features are used. The remaining rows show the results when one or more of the feature groups are removed. In comparison to the RecRank that uses all of the usage path features, performance drops highly significantly with respect to both top-1 accuracy and MRR in three cases: (1) when the length 2 forward features are removed; (2) when all forward features are removed; and (3) when all length 2 features are removed. Interestingly, removal of other feature groups does not result in significant drops in performance. In particular, removal of any of the length 3 and 4 features causes little and sometimes no change in performance. However, it is important to note that this by no means implies that features of lengths 3 and 4 are not useful: these experiments only suggest that the feature group that is being removed is not useful in the presence of the remaining features. In other words, if two feature groups encode redundant information, then removal of one of them will not cause large drops in performance. In fact, the usefulness of features of length 3 and 4 can be seen when comparing the “No length 2 forward” results and then “No forward” results: the performance differences between these two ablated systems can be attributed to the length 3 and 4 features. Specifically, top-1 accuracy
Table 5.8: Feature ablation results of RecRank

<table>
<thead>
<tr>
<th>System</th>
<th>Overall Top-1 Acc</th>
<th>Overall MRR</th>
</tr>
</thead>
<tbody>
<tr>
<td>All features</td>
<td>64.8</td>
<td>0.70</td>
</tr>
<tr>
<td>No length2 forward</td>
<td>55.8***</td>
<td>0.64***</td>
</tr>
<tr>
<td>No length3 forward fuzzy</td>
<td>64.5</td>
<td>0.70</td>
</tr>
<tr>
<td>No length3 forward no-fuzzy</td>
<td>64.2</td>
<td>0.70</td>
</tr>
<tr>
<td>No length4 forward fuzzy</td>
<td>64.7</td>
<td>0.70</td>
</tr>
<tr>
<td>No length4 forward no-fuzzy</td>
<td>64.0</td>
<td>0.70</td>
</tr>
<tr>
<td>No length2 backward</td>
<td>64.5</td>
<td>0.68</td>
</tr>
<tr>
<td>No length3 backward fuzzy</td>
<td>64.0</td>
<td>0.70</td>
</tr>
<tr>
<td>No length3 backward no-fuzzy</td>
<td>64.7</td>
<td>0.70</td>
</tr>
<tr>
<td>No length4 backward fuzzy</td>
<td>64.1</td>
<td>0.70</td>
</tr>
<tr>
<td>No length4 backward no-fuzzy</td>
<td>64.8</td>
<td>0.70</td>
</tr>
<tr>
<td>No backward</td>
<td>60.1</td>
<td>0.65</td>
</tr>
<tr>
<td>No forward</td>
<td>39.9***</td>
<td>0.48***</td>
</tr>
<tr>
<td>No length3 or4 no-fuzzy</td>
<td>65.1</td>
<td>0.70</td>
</tr>
<tr>
<td>No fuzzy</td>
<td>64.4</td>
<td>0.70</td>
</tr>
<tr>
<td>No length2</td>
<td>47.2***</td>
<td>0.53***</td>
</tr>
</tbody>
</table>
Figure 5.5: Learning curves of API recommendation approaches on the entire test set drops by more than 15% points and MRR drops by 16% points when the length 3 and 4 features are removed. Similarly, the usefulness of the backward features can be seen by comparing the “No length 2 forward” results and the “No length 2” results: the performance differences between these two ablated systems can be attributed to the backward features. Specifically, top-1 accuracy drops by 8% points.

**RQ4. What is the learning curve of each system?**

To answer this question, Figure 5.5 presents the learning curve for each of these four systems when measured in terms of top-1 accuracy. Each curve is plotted using five data points that correspond to using 20%, 40%, 60%, 80%, and 100% of the available training projects collected in Section 5.3.1. As we can see, in none of the systems does top-1 accuracy plateau even when we use all of the available training dataset. This implies that the performance of each API recommendation system will likely to improve further as additional training projects are made available, which is encouraging as additional projects can be easily obtained. In addition, we observe that SVC achieves consistently better overall top-1 accuracy than *D-Gralan* regardless of the amount of available training data. NB achieves better overall top-1 accuracy than *D-Gralan* when more than 40% of training projects are available for training. The most effective learner, however, is RecRank.

5.4. Summary
In this chapter we proposed a novel discriminative re-ranking-based API recommendation system, RecRank, which uses usage path-based features to rank the top-10 API candidates generated by Gralan. In an evaluation on eight large scale open source projects, RecRank significantly improved top-1 accuracy by 28.5%–50.0% and MRR by 0.32–0.49 in comparison to Gralan. When compared to APIREC, RecRank improved top-1 accuracy by as much as 23.7%, yielding an overall improvement of 5.3% absolute. Perhaps even more encouragingly, we saw performance improvements in each of the eight projects. Importantly, RecRank does not require access to a large number of historical code changes for training and application.

5.5. Bibliographical Notes

The paper supporting the content described in this Chapter were written in collaboration with the members at Southern Methodist University, and University of Texas at Dallas.

Chapter 6

TOWARDS AUTOMATED SOFTWARE CHANGE – RECOMMENDING LICENSES FOR CHANGED SOURCE CODE

Although as we described in Chapter 3 that the proposed software actual change impact set recommendation system can effectively help developers to find the impact classes and methods, some restrictions declared in the license statements forbid us to make violated changes in the source code. Therefore, evolving license to reduce the possibility of restriction violation is urgently needed. However, manually selecting license is a costly approach. It requires significant human efforts and legal knowledge. Therefore, in this chapter we investigate deeper into automated software change by studying another software change propagated by source code change–software license change. In software development and evolution, software license changes to ensure license compatibility and prevent license restriction conflicts among source code during software changes. It is important to make sure software licenses change correctly because software licenses are designed to protect the intellectual property of Free and Open Source Software (FOSS) using licensing mechanisms and copyright notices that determine how an open source software can be (re)used [45].

Figure 6.1: Example illustrating licensing incompatibility
Vendome et al. [128] shed light on the software license change challenge by investigating the rationales behind license changes due to software changes from both quantitative and qualitative points of view. Their study reveals that updating the license in the presence of software changes is an even more time-consuming and labor-intensive process than determining the license for a newly released software project. Specifically, a developer has to review each changed source code file against the existing licenses to determine whether there is any license incompatibility, such as the violation of existing license terms/copyright or the presence of a license that is incompatible with the license of another piece of changed source code. Consider the example in Figure 6.1, which shows that a co-changed source code module, \( \text{LogEntry} \), is imported to the source code file \( \text{XMLPacker.java} \) that was originally licensed under MPL v1.1 according to its file header. To determine if a change of license is needed, the developer would begin by determining that \( \text{LogEntry} \) is distributed under license GPL v3+. Then, it requires a careful review and comparison of the license restrictions of MPL v1.1 and those of GPL v3+ to reveal a potential incompatibility between them. That is, a software system that imports or uses \( \text{LogEntry} \) is required to adopt GPL v3+, which imposes a stronger restriction than MPL v1.1. Detecting and resolving such incompatibilities thus places a lot of burden on developers.

Unfortunately, according to Vendome et al. [128], existing methods and tools for license prediction ( [33, 46, 127, 129]) are insufficient for the task of predicting licenses \textit{in the presence of software changes}. For instance, \textit{Ninka} [46], a state-of-the-art license detection method, uses regular expressions to predict licenses by detecting the presence of license copyright and terms in the header comments of the source code files (i.e., the file header). Hence, Ninka has no problem with (independently) detecting MPL v1.1 as the license adopted by \( \text{XMLPacker.java} \) before the code change and GPL v3+ as the license adopted by \( \text{LogEntry} \). However, \textit{Ninka cannot} detect file dependencies. So, in our example, it cannot take into account the license restriction imposed by the newly imported code module. Failure to do so deprives \textit{Ninka} of its

\[ \text{A source code file } f_1 \text{ is a co-changed file of another source code file } f_2 \text{ if the two files both changed in a single change commit.} \]
ability to address the license compatibility issues that arise from code changes.

Our goal in this chapter is to advance the state-of-the-art in license prediction for software changes. Specifically, we propose Automatic License Prediction (ALP), a novel method and supporting tool for automatically predicting source code file-level licenses for code changes. Leveraging the recent successes of machine learning methods in empirical SE research, we propose a learning-based ALP system. At the core of ALP are four key ideas: (1) exploiting a rich set of features extracted from the inline text of the changed file under consideration; (2) modeling the license of the previous version of the file; (3) exploiting features extracted from the associated software documents and co-changed files; and (4) identifying and resolving incompatibilities, such as those illustrated in Figure 6.1.

Our contributions in this chapter are three-fold. First, we manually annotate the licenses of 57450 changed source code files taken from 700 Java projects hosted on GitHub. To our knowledge, this is the first large-scale effort aiming to create an annotated corpus for license prediction. Second, we propose ALP, the first machine learning approach to license prediction, which considers the dependencies among the licenses of the source code modules. Note that the development of ALP is made possible by the availability of the large amount of annotated training data provided by our corpus. Finally, extensive experiments demonstrate the effectiveness of our approach. In an evaluation on 700 Java projects involving the prediction of 25 software licenses, ALP achieves a micro F1 score of 92.5%, highly significantly surpassing the performance of three baseline systems, including Ninka, which only achieves a micro F1 score of 73.5% on the same corpus.

We believe our results have another important ramification. Vendome et al.’s [128] study of the rationales behind license changes due to software changes were based on the automatic annotations provided by Ninka. However, Ninka’s rather mediocre performance on our corpus casts doubts on the degree to which the conclusions drawn by Vendome et al. are valid. We believe that it is worthwhile to re-examine their conclusions by re-conducting their study using ALP’s output, which is considerably more accurate than Ninka’s.
6.1. Background & Problem Definition

6.1.1. Software License Background

In recent decades, more and more Free and Open Source Software (FOSS) projects have been made available online by developers. The shift towards FOSS projects allows developers to not only contribute to the software community but also benefit themselves [128]. For example, by hosting their FOSS projects on web-based version control platforms (e.g., GitHub, SourceForge, etc.), developers can receive help from third-party testers and other developers to improve the quality of their software systems. Nevertheless, developers who are interested in releasing their projects to the open source community should be aware that the redistribution, reuse and modification of their projects must be regulated under software licenses. Software licenses are important because they are designed to protect the intellectual property of FOSS using licensing mechanisms and copyright notices that determine how an open source software can be (re)used [45].

To apply appropriate software licenses to their software projects, developers need to select from a variety of licenses: either the ones that allow redistributors to incorporate the reused software under different licenses (i.e., permissive licenses) or the ones that require developers to use the same license when distributing new software that incorporates the reused software (i.e., restrictive licenses) [128]. These licenses range from highly restrictive (e.g., the General Public License (GPL) family, which requires developers to use GPL to distribute new software that reuses GPL software) to less restrictive (e.g., the MIT license, which permits a third party to freely modify, reuse and redistribute the project by keeping term notices). Therefore, selecting the appropriate license for a given piece of software requires a great deal of experience and manual review effort on the part of developers.

6.1.2. Problem Statement

The problem we aim to solve with our proposed approach is the following: Given a changed source code file, automatically recommend a license that is compatible
among source code during software change. Specifically, we design this problem into two stages: Stage 1 centers around one question: how can we identify the “conflict” instances? In order to do so, we need to identify licenses of changed file, relevant software document(s) and co-changed files; Stage 2 centers around another question: how can we resolve the conflicts that arise in the conflict instances identified in Stage 1? In another word, the correct license must be chosen if there are different licenses used by documents and/or changed/co-changed files.

6.2. Data Preparation

We collect a large set of historical change repositories from 700 Java projects hosted on GitHub. These projects and their historical change repositories are previously used by Vendome et al. [128] in the aforementioned empirical study. We determine the ground truth license name (e.g., GPL, MIT, etc.) and version (e.g., v1, v2, etc.) of each changed file in each change commit via an open coding procedure. All changed files are coded by two coders, both of whom are senior software engineering Ph.D. students who have extensive experience in industry as developers. Initially, one of the coders conducted a pilot study on a subset of the changed files and their associated software documents. This subset was chosen in the following manner. First, 250 projects and their change commit histories were randomly chosen from the dataset. Then, one file was selected randomly from each of the change commit histories. The purpose was to obtain as many different types of licenses and relevant text statements as possible. The pilot study resulted in a list of preliminary coding criteria. Each criterion either describes the conditions under which a license is applicable and/or enumerates the license(s) for which a given term is a possible indicator. For example, one criterion says that if the term “AS IS” appears in a license’s text, then either LGPL v3+ (a highly restrictive license) or BSD (a fairly restrictive license) should be the license. Moreover, the choice depends on whether an incompatibility between the licenses exists: if there is no incompatibility, then BSD suffices. Then this coder trained the other coder on the coding criteria. After training, both coders simultaneously coded all the changed files in the dataset. As for
Table 6.1: Annotation examples

<table>
<thead>
<tr>
<th>Example 1</th>
<th>Changed file</th>
<th>Software doc</th>
<th>Co-changed file</th>
</tr>
</thead>
<tbody>
<tr>
<td>Example 1</td>
<td>N/A</td>
<td>...you can redistribute it and/or modify it under the terms of the GNU General Public License version 2...</td>
<td>can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation; either version 2.1 of the License or (at your option) any later version...</td>
</tr>
<tr>
<td>Example 2</td>
<td>...Licensed under the Apache License Version 2.0...</td>
<td>program is made available under the terms of the Eclipse Public License v1.0 which accompanies this distribution...</td>
<td>The MIT License Original work sponsored and donated by...</td>
</tr>
</tbody>
</table>
inter-coder agreement, the coders achieved an agreement ratio (i.e., the percentage of changed files that are assigned the same license by the two coders) of 73.7% and a Cohen’s Kappa [30] of 0.597, which indicates moderate agreement [131]. Disagreements in their annotations were resolved by open discussion. Disagreement primarily stems from the coders’ differing interpretations of the terms of the licenses. For example, one coder mistakenly assigned LGPL v3+ as the license after seeing the term “AS IS” because he determined that a license incompatibility exists due to his interpretation of the terms. Every case of disagreement was resolved when the coders reach a common interpretation of the terms.

To enable the reader to get a better idea of how the files are annotated, Table 6.1 shows two examples. As can be seen, each example is composed of the changed file under consideration, the associated software document, and its co-changed file. Owing to space limitations, only the snippet of each file/document that is relevant to license prediction is shown. In Example 1, the software document suggests that GPL v2 should be adopted while the co-changed file suggests that LGPL v2.1 should be adopted. There is a license conflict between GPL v2 and LGPL v2.1+. Since LGPL v2.1+ has a stronger copyleft than GPL v2, in order to accommodate the strong copyleft imposed by the co-changed file, this changed file should be labeled as LGPL v2.1+.

In Example 2, the three different resources suggest three different licenses: the changed file suggests Apache v2, the document suggests EPL v1, and the co-changed file suggests MIT. Since Apache v2, EPL v1 and MIT are all permissive licenses with no incompatible clauses declared, there is no need to alter the changed file’s license. In other words, the changed file should be labeled as Apache v2.

Statistics of the resulting dataset, which contains 57540 changed files annotated with their licenses, are shown in Table 5.2. From Table 6.2-a, we can see that there are totally 24 licenses\(^2\) that have appeared at least once in the 700 projects. Table 6.2-b shows the distribution of the licenses over the 57450 changed files (“Non-licensed” is used when a license is absent in a changed file, while “Other” shows the statistics

\(^2\)According to German et al. [46], all 24 licenses are frequent FOSS licenses and are detectable by Ninka.
Table 6.2: Dataset statistics

<table>
<thead>
<tr>
<th>Licenses</th>
<th># of changed files</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apache v2</td>
<td>18770 (32.7%)</td>
</tr>
<tr>
<td>GPL v2</td>
<td>9458 (16.5%)</td>
</tr>
<tr>
<td>GPL v3+</td>
<td>5943 (10.3%)</td>
</tr>
<tr>
<td>MIT</td>
<td>3125 (5.4%)</td>
</tr>
<tr>
<td>LGPL v3+</td>
<td>2609 (4.5%)</td>
</tr>
<tr>
<td>LGPL v2.1+</td>
<td>1542 (2.7%)</td>
</tr>
<tr>
<td>BSD</td>
<td>1404 (2.4%)</td>
</tr>
<tr>
<td>EPL v1</td>
<td>1276 (2.2%)</td>
</tr>
<tr>
<td>Other</td>
<td>4960 (8.6%)</td>
</tr>
<tr>
<td>Non-licensed</td>
<td>8363 (14.6%)</td>
</tr>
</tbody>
</table>

Overall statistics

<table>
<thead>
<tr>
<th># of systems</th>
<th>700</th>
</tr>
</thead>
<tbody>
<tr>
<td># of commits</td>
<td>8128</td>
</tr>
<tr>
<td># of changed files</td>
<td>57450</td>
</tr>
</tbody>
</table>

Per-license frequencies

<table>
<thead>
<tr>
<th>% of license changed files</th>
<th>6.7%</th>
</tr>
</thead>
<tbody>
<tr>
<td>% of license unchanged files</td>
<td>93.3%</td>
</tr>
</tbody>
</table>

License change statistics

aggregated over the remaining 14 (lowest-frequency) licenses.\(^3\) As we can see from Table 6.2-b, the most frequently used license is Apache v2 (32.7%). This is perhaps not surprising: Apache v2 extends software users enough freedom to use it for any purpose. Table 6.2-c shows the percentages of changed files that involved a license change in the collected projects. As we can see, 6.7% of them have their license changed.

Given this dataset, we create a multi-class prediction task, where we seek to predict each changed file as having either one of the 24 licenses or non-licensed (i.e.,

\(^3\)These 14 licenses (and their percentages) are: MPL v1.1 (2.1%), ECL v2 (0.7%), LGPL v2.1 (0.7%), LGPL v3 (0.5%), ShareAlike v3 (0.4%), OSL v3 (0.2%), ECL v1 (0.2%), LGPL v1 (0.06%), CPL v1 (0.05%), Apache v1.1 (0.02%), Microsoft (0.01%), CDDL v1 (0.007%), public-Domain (0.003%), GPL v1 (0.002%)
the associated file does not have a license). For the sake of brevity, we will refer to the class non-licensed simply as one of the “licenses” to be predicted in the rest of the chapter.

6.3. Baseline Approaches

This section introduces three baseline approaches that we implement for controlled experiments with our ALP system.

6.3.1. Ninka

As our first baseline, we employ a state-of-the-art license prediction system, Ninka [46]. Ninka is inspired by the observation that the information about a source code file license is typically found in the inline textual comment at the beginning of a source code file (i.e., the file header). In other words, Ninka detects the license of a changed source code file by relying on its file header. Specifically, given a changed source code file, Ninka first extracts the file header and segments it into a sequence of sentences, each of which is normalized by replacing each of its phrases with its equivalent common version without changing its meaning. Then it leverages a set of pre-defined regular expressions built upon these common terms to detect the presence of the license copyrights and terms. Finally, it outputs a list of licenses that are matched by their corresponding copyrights or terms in the file header. In our experiments, we use a publicly available implementation of Ninka,\(^4\) considering its prediction for a changed file correct as long as one of the licenses in the list of predictions it returns is correct. Note that since we allow Ninka to return more than one prediction for a given file, we are effectively giving it an unfair advantage over other systems that return only one prediction per file.

To give the reader a better sense of Ninka’s weakness, we apply Ninka to the example shown in Figure 6.1. Ninka starts by extracting and normalizing the file header of XMLPacker.java. Then all the pre-defined regular expressions are applied to the file header. Among them, the regular expression built for detecting the license MPL

\(^4\)http://ninka.turingmachine.org
v1.1 matches the term “MPL” in the file header sentence “If you do not delete the provisions above, a recipient may use your version of this file under the terms of any one of the MPL”. Hence, Ninka incorrectly predicts that XMLPacker.java adopts MPL v1.1. It fails to make the correct prediction (GPL v2) because it does not consider the potential conflict with the license restrictions imposed by the changed import code module LogEntry licensed under GPL v2, which specify that “You may not propagate or modify a covered work except as expressly provided under this license”.

6.3.2. Caller-Callee (CC)

As mentioned in the introduction, developers often reason about license changes based on code imports. German et al. [43] conduct an empirical study showing that software package dependency needs to be combined with license information to identify potential cases of redistribution with license incompatibilities. In other words, any change in the licenses of the imports (imports are often referred to as the callee) may affect the license of the changed file under consideration (changed files are often referred to as the caller) [43]. For example, if a caller A.java licensed under Apache v2 imports callee B.class, while B.class’s license is updated to GPL v3+, then both B.class and A.java should adopt GPL v3+ since GPL v3+ is more restrictive than Apache v2. Consequently, developers examine the imported code modules and their licenses, typically assigning to the changed file the license that is associated with the largest number of imported code modules. Our second baseline, Caller-Callee (CC), attempts to mimic this human decision process. Note that it is applicable to both imported third-party external libraries as well as project-internal classes.

We implement CC as follows. Given a changed file whose license is to be predicted, CC first extracts all imports using an off-the-shelf tool called QDox [2]. Next, it extracts the licenses associated with the imported code modules. We leverage two tools to do this: Ninka [46], which extracts the licenses of imported local classes and LicenseFinder [1], which extracts the licenses of imported third-party libraries. Using the extracted licenses, CC assigns a license to the changed file under consideration based on the majority rule. Specifically, the extracted licenses are ranked by the
number of imported code modules that adopt them (a.k.a. the import vote), and the license with the largest number of votes is returned. Our decision to employ the import vote is inspired by (1) German et al.’s [43] finding that software package dependency needs to be combined with license information to identify potential cases of redistribution with license incompatibilities; and (2) the intuition that developers simply license a piece of software under the one adopted by a majority of its imports. If none of the imports are licensed, \( CC \) will classify the changed file as non-licensed.

To give the reader a better sense of \( CC \)’s weakness, we apply it to the example shown in Figure 6.2. \( CC \) starts by extracting all imported classes and libraries of FluentList.java and then determines which license each of them adopts. All imports and their licenses are listed in Figure 6.2. To decide which license to choose for FluentList.java, \( CC \) determines that 14 imported code modules adopt Apache v2, one adopts GPL v2, and two are non-licensed. Using the import vote, it incorrectly predicts that the license of FluentList.java is Apache v2. In particular, it fails to predict the correct license (GPL v2) because it does not take into account the license compatibility of the different license restrictions from the imported code modules.

6.3.3. Previous Version (Prev)

Our third baseline, Previous Version, is motivated by the observation that only 6.7% of the license of a code file changes from one version to another (see Table 6.2-c). To exploit this observation, \( Prev \) first predicts the license of the first version of each
file, and for each subsequent change to the file, it simply predicts its license to be the same as the one that was used in its previous version (which essentially is the one predicted for the first version). In our experiments, Prev uses the Basic ALP system (see the next subsection) to predict the license of the first version of each file.

6.4. Our Approach

In this section, we present ALP, our learning-based system for predicting licenses in changed source code files. ALP trains a classifier to classify the file as belonging to one of the 25 licenses in our corpus. For ease of exposition, we will decompose the description of ALP into four steps, starting with the basic system and then incrementally augmenting it in subsequent steps.

6.4.1. Step 1: Building the Basic ALP System

The basic ALP system trains a 25-class classifier for classifying a file as belonging to one of 25 licenses. Below we present the details on how this classifier is trained and applied.

Training the classifier To train the classifier, we create one training instance for each changed file in the training set. The label of an instance is the license of the corresponding file (or non-licensed if the file does not have a license). Each training instance is represented by two types of features.

Training the classifier To train the classifier, we create one training instance for each changed file in the training set. The label of an instance is the license of the corresponding file (or non-licensed if the file does not have a license). Each training instance is represented by two types of features.

The first type of features, code-inline text features, are extracted from each line of the source code inline text. The motivation behind these features should be fairly obvious: the inline text of a source code file may reflect the settings of this file, including its license adoption. We generate three kinds of code-inline-text features: unigrams (i.e., word tokens), bigrams (each of which is a pair of consecutive word tokens), and skipgrams (each of which is a pair of word tokens that are separated by exactly one other word token). We obtain word tokens from each line of the source code inline text using the tokenizer in the Stanford CoreNLP toolkit [82]. All code-inline-text features are binary features encoding the presence (value=1) or absence
(value=0) of the corresponding unigram/bigram/skipgram in any line of the code-inline text. For example, given a line in the header comment “OpenEMRConnect is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY”, the following skipgram features will have their values set to 1: OpenEMRConnect-distributed; is-in; distributed-the; in-hope; the-that; hope-it; that-will; it-be; will-useful; but-ANY; WITHOUT-WARRANTY.

Intuitively, any change in the source code inline text of a changed file from its previous version may reflect the intents behind source code changes, especially those that are directly relevant to the adoption and update of software licenses. We exploit changes in the source code inline text for license prediction by encoding them as **diff features**. Specifically, diff features are extracted from each line of the source code inline text that differs from its immediately previous version according to the Linux **diff** command. Like the code-inline text features, we generate three kinds of diff features: unigrams, bigrams, and skipgrams. We extract these features from each changed line of the source code and encode them as binary features, each of which indicates the presence (value=1) or absence (value=0) of the corresponding unigram/bigram/skipgram in any changed line of the code-inline text.

To train the 25-class classifier, we employ the linear-chain conditional random field (CRF) learning algorithm as implemented in the Wapiti software package [69]. The motivation behind our choice of CRF as the underlying learner will become obvious when we describe the next step.

**Applying the classifier** After training, the resulting classifier can be used to label each test instance. Test instances are created in the same way as the training instances. As described before, we use the trained CRF to classify a test instance as having one of the 25 licenses.

6.4.2. Step 2: Modeling the Previous License

License adoption depends not only on the current state of the changed file, but also on its past states. For example, if a changed file `A.java` was licensed with GPL
v3+, there is no need to update its license when a new import with MPL v1.1 is added because GPL v3+ has more “copyleft” than MPL v1.1. However, since Basic ALP predicts the license of each changed file independently of the other files, it does not exploit a changed file’s previous license(s). ALP2 is an extension of Basic ALP that attempts to implicitly exploit the license of a changed file’s immediately previous version. More specifically, rather than predicting each changed file’s license independently, we cast our license prediction task as a sequence prediction task.

Recall that given a sequence $x_1x_2\ldots x_n$ as input, the goal of sequence prediction is to output a sequence $y_1y_2\ldots y_n$ of the same length. In other words, output element $y_i$ is assumed to be the predicted class for input element $x_i$. In the context of license prediction, each input sequence $x_1x_2$ is a sequence of length 2, where $x_2$ corresponds to the changed file and $x_1$ corresponds to the previous version of the changed file.\(^5\) Hence, $y_1y_2$, the output sequence produced for $x_1x_2$, will also be of length 2, where $y_2$ is the predicted license for $x_2$ and $y_1$ is the predicted license for $x_1$.

ALP2 uses CRFs (as implemented in the Wapiti software package) to learn how to label sequences. Recall that CRFs are inherently sequence labelers. In fact, its sequence labeling capability distinguishes itself from the majority of the widely used machine learning algorithms. During training, a CRF is trained to maximize the probability of seeing the correct output sequence given an input training sequence. During testing, the Viterbi algorithm [132] is used to decode the most probable output sequence given an input test sequence. An important aspect of Viterbi is that it captures the relationship between consecutive elements in an output sequence. Since we only have sequences of length 2, CRF helps us capture the relationship between license $y_1$ and license $y_2$ in the prediction process. For instance, if license $y_1$ is rarely followed by license $y_2$ in the training data, the CRF learning algorithm will learn a model that assigns a low probability to this and other unlikely license sequences. In contrast, if $y_1$ is frequently followed by $y_2$ in the training data, the CRF learner will

\(^5\)We could employ longer sequences to capture a longer history of a given changed file’s previous licenses, but preliminary experiments indicate that employing sequences of length more than 2 does not provide additional gains, presumably because a file’s license is primarily dependent on that of its immediately preceding version.
learn a model that assigns a high probability to this and other likely license sequences. Hence, a CRF can potentially allow us to improve the prediction of $y_2$ (the license of the changed file) by exploiting $y_1$ (the license of the previous version of the changed file). We represent $x_1$ and $x_2$ using the same features that we used in Basic ALP for encoding a changed file and its previous version.

Two points deserve mention. First, for each output sequence $y_1y_2$, we use $y_2$ to be the predicted license for changed file $x_2$. However, we do not use $y_1$ to be the predicted license for $x_1$. The reason is that in practice, when predicting the license for $x_1$, $x_2$ may not even exist. Hence, it does not make sense to use information from $x_2$ when predicting $x_1$.

Second, we mentioned above that each input/output sequence for our license prediction task is of length 2. This is not true in a small number of cases, however. Recall that a changed file may not have a previous version (e.g., it is a newly created file). In that case, we will create a length one training/test sequence for each of these changed files that appears in the training/test set.

6.4.3. Step 3: Adding New Knowledge Sources

Next, we augment ALP2’s feature set with additional features extracted from two sources, the software documents associated with the changed file under consideration and its co-changed files.

6.4.3.1. Extracting features from software documents

The software documents associated with a changed file can sometimes provide important clues as to which license the file should adopt. As an example, consider the software document shown in Figure 6.3, which is a LICENSE file. The phrases highlighted in yellow, including “retain the above copyright notice”, “reproduce the above copyright notice”, and “name of the author may not be used to endorse or promote”, are relevant to determining that the license that the file should adopt is BSD, as they are consistent with the terms of the BSD license. While a software

---

6Only a snippet of the document is shown owing to space limitations.
document may contain useful information as far as license prediction is concerned, this example illustrates why automatically extracting such information is not always straightforward. First, only a small portion of the document may be relevant, so one challenge involves locating where the useful information is. Second, the useful information may not be expressed as explicitly as the name of the license that the file should adopt. Third, the license declared in the document may not be applicable to all the changed files while the source code changes. Nevertheless, being learning-based, our ALP system should be able to learn the association between phrases and licenses.

Motivated by these observations, we propose to extract document-text features from all the software documents (i.e., the readme, the POM file, and the license file) that are related to the changed file under consideration. Specifically, given a changed source code file, all the related software documents are first retrieved based on the relevance to the change. Then, a set of document-text features are extracted from each line of the retrieved software documents to represent their textual contents. Like code-inline text features, we also have three types of document-text features, namely unigrams, bigrams, and skipgrams. For each document-text feature, its value is 1 if the corresponding feature is present in the software document file. Otherwise, its value is 0.
6.4.3.2. Extracting features from co-changed files

Like software documents, co-changed files can similarly be useful for predicting the license of a given changed file. Recall that a file $f_1$ co-changed with another file $f_2$ if they both changed in a single change commit. Our example in the introduction, which was shown in Figure 6.1, illustrated why co-changed files are potentially useful for license prediction. In that example, the file XMLPacker.java was originally licensed under MPL v1.1 according to its file header, but when a co-changed source code module, LogEntry, was imported, it should adopt the stricter license that LogEntry adopts, GPL v3+. As mentioned before, without analyzing the dependencies among different files, it would not be possible to identify GPL v3+ as the correct license to use after the code change.

However, neither Basic ALP nor ALP2, the two systems we introduced in Steps 1 and 2, exploits the potentially useful information from co-changed files. In light of this weakness, we seek to extract co-change features from a source code file co-changed with the changed file whose license is to be determined. Motivated in part by the features used in Basic ALP and ALP2, we similarly extract from a co-changed file code-inline text features (i.e., unigrams, bigrams, and skipgrams) as well as diff features, which encode the difference in content between the co-changed file and its previous version.

A natural question is: how many co-changed files should we use to extract co-change features from? Using all of them could pose a computational efficiency problem, so we could use a subset of them. However, if only a subset of them were to be used, which ones should be chosen? Recall from the example in Figure 6.1 that a co-changed file will be most useful for license prediction if it suggests a license that is different from the one suggested by the changed file under consideration. Hence, it makes sense for us to use those co-changed files that suggest a different license. Of course, it is possible that none of the co-changed file suggests a different license than the one that the changed file suggests. If this happens, the co-changed files will be randomly chosen.

Since we prefer to choose those co-changed files that suggest a different license, the question, then, is: how many of them should be used? To answer this question,
we employ an empirical observation of our corpus: rarely do we see more than two
different licenses suggested by a set of co-changed files. Given this observation, it
is plausible that using just one co-changed file may suffice as long as it suggests a
different license. To empirically determine how the number of co-changed files used
to extract co-change features would impact overall performance, we will conduct
experiments where we extract co-change features from five co-changed files\(^7\) and from
just one co-changed file.

An important question that we have eluded so far is: how do we know whether a
co-changed file suggests a different license than the one that the changed file under
consideration suggests? Our idea is to use our ALP2 system. Specifically, we will
use ALP2 to predict the license of a changed file and all of its co-changed files, and
select from those co-changed files whose \textit{predicted} license is different from the original
changed file’s predicted license according to ALP2. Of course, ALP2 is not perfect,
but it provides a viable way of identifying such co-changed files.

We retrain ALP2 by augmenting its feature set with the document-text features
and the co-change features.\(^8\) Note that these two additional types of features can be
used in combination and in isolation. When they are used in combination, we name
the resulting system ALP2+Doc+Co, as ALP2 is trained with three types of features:
document-text features, co-change features, and the original features extracted from
the changed file under consideration. When they are used in isolation, we end up
with two systems, ALP2+Doc and ALP2+Co, depending on which of them is used to
augment ALP2. However, if software documents or co-changed files are not available
for the changed file under consideration, no document-text features and co-change
features can be extracted, in which case we will simply set the values of these features
to 0.

\(^7\)If fewer than five co-changed files are present, we will just use all of them.
\(^8\)What this means is that the document-text features and the co-change features need to be
computed for both the training instances and the test instances. In particular, to select which co-
changed files to compute co-change features from for a given changed file in the \textit{training set}, we
use ALP2 to predict the license of each changed and co-changed file by performing five-fold cross
validation on the \textit{training set}.
6.4.4. Step 4: Modeling Conflicts

As discussed before, software documents and co-changed files are most useful for license prediction if they suggest a different license than the one suggested by the changed file under consideration. In the previous subsection, the conflicts that resulted from the different licenses suggested by different sources of information are resolved *implicitly* by the CRF. More specifically, the CRF has access to one set of features extracted from the different sources and determines the license for the changed file under consideration.

We hypothesize that license prediction performance could be improved if we model the aforementioned conflicts *explicitly*. Specifically, we propose to first identify the set of instances with conflicts (i.e., the instances for which more than one license is suggested by different sources), and then learn to determine which source of information should be used to predict the license of a changed file when conflicts arise. Before we explain why we believe this “explicit” approach is potentially better than the “implicit” approach used in Step 3, we provide the details of our explicit approach, which is composed of two stages.

Stage 1 centers around one question: how can we identify the “conflict” instances? Given a changed file, we first train three CRFs to predict its license. The first CRF is trained only on all and only the features extracted from each changed file (i.e., the code inline text features and the diff features) in the training set. The second CRF is trained only on all and only the document-text features extracted from a changed file’s associated software documents. The third CRF is trained on all and only the co-change features extracted from co-changed files. We identify an instance as a conflict instance if at least two of the three CRFs predict more than one license for the changed file under consideration. Note that if a changed file does not have any associated software documents or co-changed files, the corresponding instance will not be marked as a conflict instance.\(^9\)

\(^9\)Note that we need to identify conflict instances from both the training set and the test set. We use the three CRFs to predict the license(s) of a changed file in the *training set* by performing 5-fold cross-validation on the training set.
Stage 2 centers around another question: how can we resolve the conflicts that arise in the conflict instances identified in Stage 1? We answer this question by training a conflict resolver. Our idea is to cast the conflict resolution task as a ranking task, where we train a discriminative ranker to resolve conflicts using the ranker-learning algorithm implemented in the libSVM software package [28]. Specifically, we create one ranking problem for each conflict instance (i.e., each changed file determined to have a license conflict) identified in Stage 1. The instances to be ranked in a ranking problem are created as follows. The first instance is represented using all and only the features extracted from the changed file under consideration. The second instance is represented using all and only the document-text features extracted from all of the associated software documents. For each co-changed file, we will create one instance that is represented using all and only the co-change features extracted from the co-changed file. As mentioned in Step 3, we will experiment with using one co-changed file and using five co-changed file. This means that each ranking problem will be composed of 2–7 instances: we know that it will contain at least two instances because the changed file was determined to have a conflict in Stage 1; at the same time, we know the upper bound is 7 because besides the changed file with license to be predicted, we can have at most one instance representing the software document and at most five instances corresponding to the five co-changed files.

The goal of the ranker-learning algorithm is to rank the instances in each ranking problem so that the ones that predict the correct license are ranked higher than those that predict an incorrect license. With this goal in mind, we assign the rank value to each instance in each ranking problem as follows. If the license associated with an instance is correct, its rank value is HIGH; otherwise, its rank value is LOW. Note that the license associated with an instance is predicted by one of the three CRFs in Stage 1. For instance, the license associated with the instance corresponding to the software documents is the one predicted by the second CRF in Stage 1.

The resulting ranker can be applied to the conflict instances in the test set. For each conflict instance in the test set, a ranking problem will be created. This ranking problem is created in the same way as those in the training set. The ranker is then
used to rank the instances in the ranking problem. The license associated with the highest-ranked instance according to the ranker will be our system’s predicted license for the changed file under consideration. Note that the ranker will be applied to all and only those changed files that are determined to have a license conflict.

We believe our approach of explicitly modeling conflicts has at least two advantages over the implicit approach described in Step 3. First, (explicitly) identifying conflict instances enables us to learn a ranker to handle them specifically. This contrasts with the implicit approach, where all of the instances, regardless of whether they are conflict instances or not, are being classified by one model. In other words, the conflict instances, which are supposedly the difficult cases in license prediction, may be given less attention by the CRF model in the implicit approach because the CRF is being trained on both the easier (non-conflict) instances and the difficult (conflict) instances. Second, our ranker is not trained to directly predict licenses. Rather, it ranks instances corresponding to different sources of information. Consequently, compared to the CRFs trained in Steps 1–3, the ranker will have less bias towards classifying a conflict instance as belonging to one of the frequently occurring licenses in our corpus.

6.5. Empirical Evaluation

6.5.1. Experimental Setup

The goal of our empirical evaluation is to determine how accurately ALP can predict licenses in software changes. Our evaluation dataset is composed of changed source code files collected from 700 Java projects, where each file can be classified as belonging to one of 25 licenses.

Evaluation settings. We did not apply any text preprocessing to the relevant software documents we retrieved or the changed files. Hence, all systems, including the three baselines and all variants of our ALP system, are performed on the original (un-preprocessed) software documents and/or changed files. Given that ALP is learning-based, we evaluate it by adopting a five-fold cross validation strategy, in which the
subject projects are evenly distributed into five folds. In each fold experiment, we use three folds for training ALP, one fold for development (i.e., parameter tuning), and the remaining fold as our held-out test set.

Two points deserve mention. First, using a five-fold cross-validation strategy ensures that the entire dataset is used for training, parameter tuning and testing. For parameter tuning, we tune the regularization parameter $C$ associated with each CRF and each SVM ranker we train. Intuitively, the larger the $C$ value is, the higher the penalty on training error is. We choose the $C$ value that maximizes the overall micro F1 score (see below for details) on development data. Second, note that we divide the subject projects into five folds, meaning that all the files associated with a particular project will appear in the same fold. The reason for doing this is simple: in reality, a license prediction system will likely be used to predict licenses for the files in a totally new project. In other words, we cannot assume that there is any relationship between the training projects and the test projects. Hence, our dividing the projects into folds mimics this real-life application scenario. Not surprisingly, the learning task resulting from this particular way of creating the five folds is also harder, as it renders any project-specific knowledge that our system learns from the training data useless when the system is applied to the unseen projects in the test data.

**Evaluation metrics.** We use per-license precision, recall and F1 score to measure the performance of our systems. The precision (P) for license $l$ is the percentage of changed files predicted as $l$ that are correct with respect to the gold set (i.e., $\text{Precision} = \frac{TP}{TP+FP}$). The recall (R) for license $l$ is the percentage of changed files licensed under $l$ that are correctly predicted as $l$ (i.e., $\text{Recall} = \frac{TP}{TP+FN}$). The F1 score is the harmonic mean of precision and recall (i.e., $F1 = \frac{2 \times R \times P}{R + P}$)

To facilitate comparisons between different systems, we also compute the overall performance of each system by *aggregating* the per-license results. Specifically, we employ two commonly-used metrics, macro F1 and micro F1. Macro F1 is the unweighted average of the per-license F1 scores. Micro F1 is the fraction of instances that are correctly classified. Hence, macro F1 gives equal importance to each license,
whereas micro F1 puts more weights on more frequently occurring licenses.

**Statistical significance and effect size.** To determine whether the performance difference between two systems is statistically significant or not, we conduct the Wilcoxon rank-sum test. The type of distribution used for Wilcoxon rank-sum test is normal distribution. Following Miller [87], the result of a significance test can be interpreted as follows: The performance difference between the two systems under comparison is (1) *highly significant* if the null hypothesis (i.e., there is no performance difference between the two systems) can be rejected at the 0.01 level; (2) *significant* if it can be rejected at the 0.05 level; and (3) *moderately significant* if it can be rejected at the 0.1 level. Otherwise, the difference is statistically indistinguishable. Moreover, to evaluate the amount of performance difference between the two systems under comparison, we compute Cliffs delta [29], a non-parametric effect size measure. According to Romano et al. [114], the difference implies (1) a *large* effect size if the delta value is greater than 0.474; (2) a *medium* effect size if the delta value is greater than 0.33; and (3) a *small* effect size otherwise.

6.5.2. Results and Discussion

This section empirically answers our research questions.

**RQ1:** Which license prediction system performs the best?

Five-fold cross-validation results are shown in Table 5.3. Each row shows the macro and micro F1 scores of one system.

A few points deserve mention. First, *Ninka* is the best of the three baselines (rows 1 to 3), achieving a micro F1 of 73.5 and a macro F1 of 38.2. In particular, it highly significantly outperforms *Prev*, the second best baseline, with a large effect size in terms of macro F1 and significantly outperforms it with a large effect size in terms of micro F1. *Prev* in turn highly significantly outperforms *CC*, the weakest baseline, with a large effect size in terms of both macro and micro F1.

Second, Basic ALP (row 4), the most basic variant of ALP, performs as least as well as *Ninka* (row 1), the best baseline. Specifically, Basic ALP achieves micro and macro F1 scores of 82.2 and 38.9, which represents a highly significant improvement of 8.7
Table 6.4: Five-fold cross-validation results. The strongest result in each column is boldfaced.

<table>
<thead>
<tr>
<th>Systems</th>
<th>macro-F1</th>
<th>micro-F1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Ninka</td>
<td>38.2</td>
<td>73.5</td>
</tr>
<tr>
<td>2 CC</td>
<td>17.3</td>
<td>39.6</td>
</tr>
<tr>
<td>3 Prev</td>
<td>30.6</td>
<td>66.3</td>
</tr>
<tr>
<td>4 Basic ALP</td>
<td>38.9</td>
<td>82.2</td>
</tr>
<tr>
<td>5 ALP2</td>
<td>46.4</td>
<td>88.3</td>
</tr>
<tr>
<td>6 ALP2+Co1</td>
<td>45.7</td>
<td>87.9</td>
</tr>
<tr>
<td>7 ALP2+Co5</td>
<td>45.9</td>
<td>88.8</td>
</tr>
<tr>
<td>8 ALP2+Doc</td>
<td>47.9</td>
<td>90.3</td>
</tr>
<tr>
<td>9 ALP2+Doc+Co1</td>
<td>48.3</td>
<td>90.9</td>
</tr>
<tr>
<td>10 ALP2+Doc+Co5</td>
<td>47.9</td>
<td>90.3</td>
</tr>
<tr>
<td>11 ALP2-Ranker1</td>
<td>79.2</td>
<td>92.5</td>
</tr>
<tr>
<td>12 ALP2-Ranker5</td>
<td>77.4</td>
<td>92.5</td>
</tr>
</tbody>
</table>

points with a large effect size in micro F1 and a moderately significant improvement of 0.7 points with a large effect size in macro F1.

Third, ALP2 (row 5), which casts license prediction as a sequence prediction problem, highly significantly outperforms Basic ALP (row 4) with a large effect size in terms of micro F1 and moderately significantly outperforms it with a large effect size in terms of macro F1. These results provide suggestive evidence that modeling the immediately previous license is useful for predicting both frequent licenses (because of the improvement in micro F1) and infrequent licenses (because of the improvement in macro F1).

Fourth, incorporating additional knowledge derived from software documents and co-changed files as features for training ALP2 is generally, though not always, helpful for license prediction. As mentioned before, the document-text features (derived from
the software documents) and the co-change features (derived from the co-changed files) can be applied in isolation and in combination with the changed file’s features that are originally used to train ALP2. Results of adding only co-change files to ALP2 are shown in row 6 (ALP2+Co1, where co-change features were derived from just one co-changed file) and row 7 (ALP2+Co5, where co-change features were derived from five co-changed files). As we can see, adding co-change features may not always yield better performance. In contrast, adding only document-text features (ALP2+Doc, row 8) yields small, but moderately significant improvements with a medium effect size in terms of both macro and micro F1 scores. When the two types of features are applied in combination, we see small, insignificant gains in both micro and macro F1 scores when one co-changed file was used (ALP2+Doc+Co1, row 9). Overall, these results seem to suggest that document-text features are more useful than co-change features for license prediction when used in combination with the features derived from the changed file. In addition, deriving features from five co-changed files yields results that are statistically indistinguishable from those obtained using only one co-changed file.

Finally, comparing rows 11 and 12 with rows 6 to 9, we see that explicitly modeling and resolving conflicts using a ranker (ALP-Ranker1 and ALP-Ranker5) is much more effective in improving ALP2 than implicitly resolving conflicts (by incorporating features derived from different sources into just one feature set). Again, we have two sets of ranking results, one obtained by employing one co-changed file and the other five co-changed files. Note that the difference between these two sets of results is indistinguishable. Both sets of ranking results are highly significantly better than the best implicit results (row 9) with a large effect size in terms of macro F1 and significantly better than it with a large effect size in terms of micro F1. It is worth noting that in comparison to row 9 (the best implicit results), the macro F1 score improves by more than 30 points. This is very encouraging, since it is common for learning-based systems to sacrifice minority class performance for majority class

10Due to randomness involved in the selection of the one/five co-changed files, we repeat each of these experiments five times and report the average F1 scores in Table 6.3.
Table 6.5: Five-fold cross-validation results of systems on Ninka-detectable, Ninka-undetectable, and conflict instances

<table>
<thead>
<tr>
<th>Systems</th>
<th>Ninka-det</th>
<th>Ninka-undet</th>
<th>Conflict</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ma</td>
<td>mi</td>
<td>F1</td>
</tr>
<tr>
<td>1 Ninka</td>
<td>97.2</td>
<td>98.1</td>
<td>0.0</td>
</tr>
<tr>
<td>2 CC</td>
<td>38.1</td>
<td>41.7</td>
<td>18.7</td>
</tr>
<tr>
<td>3 Prev</td>
<td>88.9</td>
<td>84.3</td>
<td>45.0</td>
</tr>
<tr>
<td>4 Basic ALP</td>
<td>98.0</td>
<td>98.8</td>
<td>49.5</td>
</tr>
<tr>
<td>5 ALP2</td>
<td>97.8</td>
<td>98.8</td>
<td>67.5</td>
</tr>
<tr>
<td>6 ALP2+Co1</td>
<td>97.5</td>
<td>98.7</td>
<td>67.2</td>
</tr>
<tr>
<td>7 ALP2+Co5</td>
<td>97.8</td>
<td>98.7</td>
<td>68.6</td>
</tr>
<tr>
<td>8 ALP2+Doc</td>
<td>98.0</td>
<td>99.0</td>
<td>72.2</td>
</tr>
<tr>
<td>9 ALP2+Doc+Co1</td>
<td>97.8</td>
<td>98.8</td>
<td>73.3</td>
</tr>
<tr>
<td>10 ALP2+Doc+Co5</td>
<td>97.6</td>
<td>98.8</td>
<td>71.4</td>
</tr>
<tr>
<td>11 ALP2-Ranker1</td>
<td>99.1</td>
<td>98.7</td>
<td>90.0</td>
</tr>
<tr>
<td>12 ALP2-Ranker5</td>
<td>99.1</td>
<td>98.3</td>
<td>91.5</td>
</tr>
</tbody>
</table>

performance (because of their bias towards classifying an instance as belonging to a majority class). These results suggest that our idea of training a ranker to not predict licenses directly can effectively mitigate the problem that skewed class distributions typically bring about.

**RQ2. How do the systems perform on the easy, difficult, and conflict instances?**

To gain additional insights into the different ALP variants, we report the performance of different systems, including the baselines, on three subsets of the instances in our dataset.

First, we compare system performance on only the Ninka-detectable instances (i.e., the set of instances whose license can be predicted by Ninka). They account
for 60.6% of the instances in our dataset. They are of interest because they are the easy-to-classify instances: their licenses can be simply extracted using one of *Ninka*’s high-precision regular expressions. Macro and micro F1 scores on these easy instances are shown under the “Ninka-det” column in Table 5.5. As we can see, *Ninka* achieves near-perfect performance on these instances, which is not surprising. Moreover, all of the ALP variants perform at least as well as *Ninka*, with the best results achieved by the rankers (rows 11 and 12).

Second, we compare system performance on only the Ninka-undetectable instances (i.e., the set of instances for which *Ninka* failed to predict any license). They account for 39.4% of the instances in our dataset. These instances are of interest for one important reason: since *Ninka* is a state-of-the-art system and these instances cannot be classified by *Ninka*, any success in predicting their licenses represents a solid advance over the current state-of-the-art. Macro and micro F1 scores on these “difficult” instances are shown under the column “Ninka-Undet” in Table 5.5. As we can see, more sophisticated ALP variants tend to yield better performance on these difficult instances than their simpler counterparts. In fact, comparing the results on the Ninka-detectable instances and the Ninka-undetectable instances, we can see that our extensions to the Basic ALP system have primarily helped to predict the licenses of the difficult instances. The best performance on the difficult instances is achieved by ALP2+Ranker1 (row 11): macro and micro F1 scores of 90 and 85, respectively.

Finally, we compare system performance on only the instances that are determined to be conflict instances according to ALP2-Ranker1. Results on the conflict instances will shed light on how well the ALP variants, particularly the rankers, are in resolving conflicts. Macro and micro F1 scores on these instances are shown under the “Conflict” column in Table 5.5. As we can see, *Ninka* achieves a micro F1 score of 56.9, meaning that not all conflict instances are difficult to classify. A closer examination of the conflict instances reveals the reason. Recall that these instances are *predicted* to be conflict instances. Specifically, some easy (Ninka-detectable) instances that do not have conflicts are mis-predicted to have conflicts. As an example, *Ninka* predicts an instance as having license A, and the associated LICENSE file simply
says a license is needed (without specifying which license should be used). This is an instance that does not have a conflict, but the CRF classifies the software document as non-licensed, thus erroneously creating a conflict instance for an easy instance. Nevertheless, the substantially higher macro and micro F1 scores achieved by the rankers show that they have successfully classified many difficult conflict instances.

**RQ3. What are some of the errors made by our best ALP variant, ALP2-Ranker1?**

To address this research question, we show in Table 5.6 two conflict instances that the ranker misclassified. As can be seen, each example is composed of the changed file under consideration, the associated software document, and the co-changed file. Owing to space limitations, only the snippet of each file/document that is relevant to license prediction is shown.

Example 1 shows that the changed file originally adopts GPL v2 while the co-changed file adopts GPL v3+. Both GPL v2 and GPL v3+ are weak copyleft licenses. While the correct license is GPL v2, the conflict resolver suggests a license change to GPL v3+. In cases like this, developers will typically keep GPL v2 as the changed file’s license since it is a major license (16.5% of the source code files across all subject projects in our dataset have this license). To improve the accuracy of ALP2-Ranker1, one can encode the license type (e.g., weak copyleft) and the popularity of a license as features.

Example 2 shows that the changed file originally adopts Apache v2 while the software document suggests GPL v2. Both licenses are loss copyleft (either weak or no copyleft) licenses. While the correct license is GPL v2, the conflict resolver mistakenly labels the changed file as Apache v2. To correctly classify this instance, however, the resolver may need to understand that GPL v2 does not permit incorporating one program into another proprietary program, such as linking proprietary applications with the library created by the author. In other words, given its stricter license clauses, GPL v2 should be used as the license when a conflict instance involves both Apache v2 and GPL v2.
Table 6.6: Examples of errors made by ALP2-Ranker1

<table>
<thead>
<tr>
<th>Changed file</th>
<th>Software doc</th>
<th>Co-changed file</th>
</tr>
</thead>
<tbody>
<tr>
<td>Example 1...you can redistribute it and/or modify it under the terms of the GNU General Public License version 2...You may not impose any further restriction on the recipients’ exercise of the rights granted herein...</td>
<td>N/A</td>
<td>...under the conditions of the GNU General Public License Version 3...you may add to a covered work material governed by the terms of that license document provided that the further restriction does not survive such relicensing or conveying...</td>
</tr>
<tr>
<td>Example 2...Licensed under the Apache License Version 2.0...</td>
<td>...under the terms of the GNU General Public License version 2.0...</td>
<td>N/A</td>
</tr>
</tbody>
</table>
6.6. Summary

Making appropriate selection of software licenses to adopt or update after software changes usually requires a great deal of experience and manual effort. To address this challenge, we annotated a large corpus of changed files with their licenses and developed ALP, a novel method and tool for automatic code-level license prediction for software changes. In an evaluation on 700 open source projects with a rich code change history, ALP2-Ranker1, the best variant of ALP, achieves an accuracy of 92.5% micro F1 score and 79.2% macro F1 score on licensed code changes, significantly surpassing the performance of three baselines, including a state-of-the-art license prediction system, *Ninka*.

6.7. Bibliographical Notes

The paper supporting the content described in this chapter were written in collaboration with the members at Southern Methodist University, Nanjing University, and University of Texas at Dallas.

Chapter 7

CONCLUSION & FUTURE WORK

As the complexity of software grows, software development is becoming more and more challenging and knowledge intensive. Consequently, it costs a significant amount of time and human efforts in software development, especially in software change process. In this dissertation, we proposed four recommendation systems to support intelligent software development and evolution. In addition, this dissertation establishes a future roadmap of research and development activities.

7.1. Concluding Remarks

We opened this dissertation with a thesis statement that asserted in essence, that automating the process of representing and relating software artifacts in source code and natural language allow for more efficient and effective software change. To summarize our contributions, in this dissertation we presented four novel recommendation systems that automated the software change processes. First, we introduced an automated system (CHIP) to predict software actual change impact set more effectively by using not only call dependency but also data sharing dependency. Second, to address the task of linking changed source code to untangled change intents, we presented two AutoCILink systems—a pattern-based system AutoCILink-P and a learning-based system AutoCILink-ML. Third, we introduced RecRank, a technique for automatically recommending the correct API by taking only top-10 API candidates. Finally, we introduced ALP, which is capable of automatically predicting code-level licenses for software changes.

To support our core thesis that recommendation systems can improve the processes of software development and evolution, we offer some quantitative empirical evidences. First, CHIP is empirically evaluated in both method-level and class-level on four open source projects. The evaluation results indicate that after applying
data sharing dependencies with extensions, CHIP is more effective in predicting actual change impact set in all four projects. Moreover, it shows that on predicting certain change impact scenarios specific type of data sharing dependencies are particular useful. Second, we evaluated AutoCILink on a newly annotated corpus from the repositories of 19 open source projects with links between changed source code files and untangled change intents and found that AutoCILink outperformed all baseline systems under comparison. Specifically, AutoCILink-P achieves 83.4% F1-score on “linked” code-intent data, 40.2% F1-score on “not linked” data and an average accuracy of 74.6%. AutoCILink-ML improves the performance and achieves 87.4% F1-score on “linked” data and 62.4% F1-score on “not linked” data, and an average accuracy of 81.2%. Third, our evaluation of RecRank on eight large scale open source projects illustrate that RecRank significantly improved top-1 accuracy in comparison to state-of-the-art approaches on all projects. While there is still much work to be done, the evidence mentioned above help support the notion that recommendation systems can dramatically improve the effectiveness and efficiency of developers in software development and evolution. Finally, the evaluation of ALP performed on 700 open source projects with a rich code change history shows that the best variant of ALP, ALP2-Ranker1, significantly outperformed the performance of the three baseline systems, including a state-of-the-art license prediction system, Ninka.

7.2. A Future Roadmap

Our proposed recommendation systems have helped to automate various aspects of software change. However, the presented work only touches the surface of various components of the change process that are ripe for automation. Thus, there are several promising directions of future work. Here we brief summarize some of those activities.

**Toward Automatically Propagating Software Change.** Software systems are composed of entities such as classes, methods, variables and software documents. These entities depend on one another. As we mentioned in Chapter 3, when systems evolve, developers must ensure that dependent entities’ changes are propagated
consistently. Otherwise, it may cause bugs and even significant system failures. To solve this challenge, CHIP is the first instance described in Chapter 3, which is aimed at predicting actual change impact set in code cases by relying on code structure, such as call and data sharing dependency. In order to understand the intent behind each changed code, in Chapter 4 we illustrated AutoCILink to establish links between changed source code and untangled change intents. In addition, in Chapter 5 we got even closer by presenting an API recommendation engine can effectively suggest the correct API to developers by achieving the state-of-the-art performance. Furthermore, in Chapter 6 we proposed ALP to predict compatible software licenses for software changes.

In summary, it is clear that automated support in software change propagation would greatly benefit developers. Other types of changes should be automated as well. The overarching goals of this future research thrust regarding automated software change propagation are as follows:

- **Research Goal 1:** *Understanding Developer’s and User’s Information Needs in Documenting Change Intent:* In order to create effective automated documentation for change intents, it is important to first understand what documentation information both developers and users find useful. Thus, the first goal of this research thrust is to conduct studies that will shed light on information needs for change intent documentation.

- **Research Goal 2:** *Designing Developer and User-Centric Approaches for Automated Change Intent Documentation:* Once we have established a set of guidelines for effective change intent documentation in eyes of developers and users, we will leverage this knowledge to create approaches that are capable of automatically documenting change intents as software evolves.

- **Research Goal 3:** *Explore the Representational Power of Source Code:* In order to move toward approaches capable of automatically suggesting source code for software change, the degree to which both the semantics and syntac-
tical information can be learned from software historical repositories must be explored.

- **Research Goal 4:** *Designing Approaches for Synthesizing Source Code:* According to the information gleaned from studying the representational power of source code, we will design approaches for synthesizing code related to various discrete functional requirements.
REFERENCES


[27] Chang, C.-C., and Lin, C.-J. Libsvm: a library for support vector machines. ACM transactions on intelligent systems and technology (TIST) 2, 3 (2011), 27.


[69] Lavergne, T., Cappé, O., and Yvon, F. Practical very large scale CRFs. In *Proceedings the 48th Annual Meeting of the Association for Computational Linguistics (ACL)* (July 2010), pp. 504–513.


[114] ROMANO, J., KROMREY, J. D., CORAGGIO, J., AND SKOWRONEK, J. Appropriate statistics for ordinal level data: Should we really be using t-test and Cohens d for evaluating group differences on the NSSE and other surveys. In


