# Leveraging Machine Learning to Improve Software Reliability

by

Song Wang

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2018

**Examining Committee Membership**

The following served on the Examining Committee for this thesis. The decision of the Examining Committee is by majority vote.

| | |
|---|---|
| External Examiner: | **Dr. Thomas Zimmermann**<br>Senior Researcher<br>Microsoft Research |
| Supervisor(s): | **Dr. Lin Tan**<br>Associate Professor, Canada Research Chair<br>Electrical and Computer Engineering<br>University of Waterloo |
| Internal Members: | **Dr. Sebastian Fischmeister**<br>Associate Professor<br>Electrical and Computer Engineering<br>University of Waterloo<br><br>**Dr. Mark Crowley**<br>Assistant Professor<br>Electrical and Computer Engineering<br>University of Waterloo |
| Internal-External Member: | **Dr. Joanne Atlee**<br>Professor, Director of Women in Computer Science<br>David R. Cheriton School of Computer Science<br>University of Waterloo |

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

**Abstract**

Finding software faults is a critical task during the lifecycle of a software system. While traditional software quality control practices such as statistical defect prediction, static bug detection, regression test, and code review are often inefficient and time-consuming, which cannot keep up with the increasing complexity of modern software systems. We argue that machine learning with its capability in knowledge representation, learning, natural language processing, classification, etc., can be used to extract invaluable information from software artifacts that may be difficult to obtain with other research methodologies to improve existing software reliability practices such as statistical defect prediction, static bug detection, regression test, and code review.

This thesis presents a suite of machine learning based novel techniques to improve existing software reliability practices for helping developers find software bugs more effective and efficient. First, it introduces a deep learning based defect prediction technique to improve existing statistical defect prediction models. To build accurate prediction models, previous studies focused on manually designing features that encode the statistical characteristics of programs. However, these features often fail to capture the semantic difference of programs, and such a capability is needed for building accurate prediction models. To bridge the gap between programs' semantics and defect prediction features, this thesis leverages deep learning techniques to learn a semantic representation of programs automatically from source code and further build and train defect prediction models by using these semantic features. We examine the effectiveness of the deep learning based prediction models on both the open-source and commercial projects. Results show that the learned semantic features can significantly outperform existing defect prediction models.

Second, it introduces an n-gram language based static bug detection technique, i.e., Bugram, to detect new types of bugs with less false positives. Most of existing static bug detection techniques are based on programming rules inferred from source code. It is known that if a pattern does not appear frequently enough, rules are not learned, thus missing many bugs. To solve this issue, this thesis proposes Bugram, which leverages n-gram language models instead of rules to detect bugs. Specifically, Bugram models program tokens sequentially, using the n-gram language model. Token sequences from the program are then assessed according to their probability in the learned model, and low probability sequences are marked as potential bugs. The assumption is that low probability token sequences in a program are unusual, which may indicate bugs, bad practices, or unusual/special uses of code of which developers may want to be aware. We examine the effectiveness of our approach on the latest versions of 16 open-source projects. Results

show that Bugram detected 25 new bugs, 23 of which cannot be detected by existing rule-based bug detection approaches, which suggests that Bugram is complementary to existing bug detection approaches to detect more bugs and generates less false positives.

Third, it introduces a machine learning based regression test prioritization technique, i.e., QTEP, to find and run test cases that could reveal bugs earlier. Existing test case prioritization techniques mainly focus on maximizing coverage information between source code and test cases to schedule test cases for finding bugs earlier. While they often do not consider the likely distribution of faults in the source code. However, software faults are not often equally distributed in source code, e.g., around 80% faults are located in about 20% source code. Intuitively, test cases that cover the faulty source code should have higher priorities, since they are more likely to find faults. To solve this issue, this thesis proposes QTEP, which leverages machine learning models to evaluate source code quality and then adapt existing test case prioritization algorithms by considering the weighted source code quality. Evaluation on seven open-source projects shows that QTEP can significantly outperform existing test case prioritization techniques to find failed test cases early.

Finally, it introduces a machine learning based approach to identifying risky code review requests. Code review has been widely adopted in the development process of both the proprietary and open-source software, which helps improve the maintenance and quality of software before the code changes being merged into the source code repository. Our observation on code review requests from four large-scale projects reveals that around 20% changes cannot pass the first round code review and require non-trivial revision effort (i.e., risky changes). In addition, resolving these risky changes requires 3X more time and 1.6X more reviewers than the regular changes (i.e., changes pass the first code review) on average. This thesis presents the first study to characterize these risky changes and automatically identify these risky changes with machine learning classifiers. Evaluation on one proprietary project and three large-scale open-source projects (i.e., Qt, Android, and OpenStack) shows that our approach is effective in identifying risky code review requests.

Taken together, the results of the four studies provide evidence that machine learning can help improve traditional software reliability such as statistical defect prediction, static bug detection, regression test, and code review.

**Acknowledgments**

First of all, I would like to express my deep gratitude to my supervisor, Dr. Lin Tan: thanks for accepting me as your student, believing in me, encouraging me to pursue the research topics that I love, being always there when I need help, and helping me grow with your continuous support and mentorship. Her guidance will continue to influence me beyond my Ph.D. education. I could not have wished for a better supervisor for my Ph.D. studies!

I would like to thank my examination committee members—Dr. Joanne M. Atlee, Dr. Thomas Zimmermann, Dr. Sebastian Fischmeister, and Dr. Mark Crowley, for providing useful comments that help me improve my thesis.

I would also like to thank Dr. Nachi Nagappan, Dr. Thomas Zimmermann, Dr. Harald Gall, Dr. Christian Bird, Zhen Gao, Chetan Bansal, Vincent J Hellendoorn, Justin Smith from whom I learned a lot, together with whom I had an amazing and wonderful internship at Microsoft Research in Summer 2018.

Thanks to all the people in our research group who made my life at Waterloo fantastically joyful: Dr. Jaechang Nam, Dr. Jinqiu Yang, Edmund Wong, Thibaud Lutellier, Michael Chong, Taiyue Liu, Ming Tan, Yuefei Liu, Xinye Tang, Alexey Zhikhartsev, Devin Chollak, Yuan Xi, Moshi Wei, Tian Jiang, Lei Zhang, Hung Pham, Weizhen Qi, Yitong Li, and Tej Toor—thank you very much for all the friendship, support, and joy.

I want to thank all my other collaborators during my Ph.D.'s study: Dr. Junjie Wang, Dr. Dana Movshovitz-Attias, Dr. Qiang Cui, Dr. Ke Mao, Dr. Tim Menzies, Mingyang Li, Adithya Abraham Philip, Dr. Mingshu Li, and Hu Yuanzhe. I really enjoyed working with them, and have learnt a lot from them.

I would like to acknowledge my Master's supervisors, Dr. Qing Wang, Dr. Ye Yang, and Dr. Wen Zhang, who introduced me to the software engineering area, taught me many research skills, helped me publish my first research papers, and gave me their strong support during my Ph.D. application process. Without them, I can hardly imagine that I can eventually become a Ph.D. in the software engineering area.

Finally, I would like to thank my family—my parents, parents-in-law, my younger brother, my two sisters and brothers-in-law, my four lovely nephews, for their support and unconditional love. And lastly, my beloved wife, Qi: thank you very much for your support, understanding, care, and love. I dedicate this thesis to them.

Dedicated to my family.

# Table of Contents

# List of Tables

# List of Figures

# Chapter 1

# Introduction

Today, software is integrated into every part of our society, which makes building reliable software an increasingly critical challenge for software developers. As a consequence, the impact and cost of software bugs increase dramatically, e.g., according to a report by Tricentis software bugs cost the worldwide economy around 1.7 trillion dollars in 2017 [1]. Thus, the techniques to help developers detect software bugs for improving software reliability are highly needed.

For building reliable software, many different software reliability practices have emerged over the last years to help developers improve software reliability by detecting bugs, i.e., statistical defect prediction that leverages software metrics to build classifiers and predicts unknown defects in the source code [82, 104, 138, 183, 189, 220, 326], static bug detection that uses static code analysis to infer heuristic patterns and detect violations as software faults [4, 5, 10, 30, 41, 70, 144, 207, 278], regression test that aims to find the broken functionalities earlier by scheduling and running test cases [102, 166, 233–235, 258, 266, 319], and code review that aims to reveal potential quality issues and improve the maintenance and quality of software before the code changes being merged into the source code repository [62, 105, 162, 182, 228, 229, 264, 267], etc.

Although these techniques have already been widely adopted in industry and proven can help detect bugs [2,3,33,34,127,167], most of the state-of-the-art techniques do not perform well in terms of effectiveness (i.e., low accuracy) or efficiency (i.e., time-consuming), which cannot keep up with the increasing complexity of modern software systems.

In this thesis, we argue that machine learning with its capability in knowledge representation, learning, natural language processing, classification, etc., can enable software

engineering researchers to extract invaluable information from software artifacts generated during software development, including code revision history, issue reports, history bugs and patches, software documentation, source code, etc., to improve existing software reliability practices.

This thesis presents four examples of how machine learning techniques can be used to mine invaluable information about the development of software systems, in aid of improved software reliability practices. To be precise, the thesis statement of this dissertation is as follows:

**Thesis Statement.** *Machine learning, with its capability in knowledge representation, learning, natural language processing, classification, etc., can help improve traditional software reliability practices such as statistical defect prediction, static bug detection, regression test, and code review.*

We provide evidence for this thesis by presenting four machine learning based novel techniques to improve existing software reliability practices for finding bugs more effective and efficient. First, it introduces a deep learning based defect prediction technique to improve existing statistical defect prediction models. To build accurate prediction models, previous studies focused on manually designing features that encode the characteristics of programs. However, these features often fail to capture the semantic difference of programs, and such a capability is needed for building accurate prediction models. To bridge the gap between programs' semantics and defect prediction features, this work leverages deep learning techniques to learn a semantic representation of programs automatically from source code and further build and train defect prediction models based on these semantic features. We examine the effectiveness of the deep learning based defect prediction approaches on both the open-source and commercial projects. Results show that the learned semantic features can significantly outperform the existing defect prediction models. Chapter 3 of the thesis describes this work in detail.

Second, it introduces an n-gram language based static bug detection technique, i.e., Bugram, to detect new types of bugs with less false positives. Most of existing static bug detection techniques are based on programming rules inferred from source code. It is known that if a pattern does not appear frequently enough, rules are not learned, thus missing many bugs. To solve this issue, this thesis proposes Bugram, which leverages n-gram language models instead of rules to detect bugs. Specifically, Bugram models program tokens sequentially, using the n-gram language model. Token sequences from the program are then assessed according to their probability in the learned model, and low probability sequences are marked as potential bugs. The assumption is that low probability token sequences in a program are unusual, which may indicate bugs, bad practices, or

unusual/special uses of code of which developers may want to be aware. We examine the effectiveness of our approach on the latest versions of 16 open-source projects. Results show that Bugram detected 25 new bugs, 23 of which cannot be detected by existing rule-based bug detection approaches, which suggests that Bugram is complementary to existing bug detection approaches to detect more bugs and generates less false positives. Chapter 4 of the thesis describes this work in detail.

Third, it introduces a machine learning based regression test prioritization technique, i.e., QTEP, to find and run test cases that could reveal bugs earlier. Existing test case prioritization techniques mainly focus on maximizing coverage information between the source code and test cases to schedule test cases for finding bugs earlier. While they often do not consider the likely distribution of faults in the source code. However, software faults are not often equally distributed in source code, e.g., around 80% faults are located in about 20% source code. Intuitively, test cases that cover the faulty source code should have higher priorities, since they are more likely to find faults. To solve this issue, this thesis proposes QTEP, which leverages machine learning models to evaluate source code quality and then adapt existing test case prioritization algorithms by considering the weighted source code quality. Evaluation on seven open-source projects shows that QTEP can significantly outperform existing test case prioritization techniques to find failed test cases early. Chapter 5 of the thesis describes this work in detail.

Finally, it introduces a machine learning based approach to identifying risky code review requests. Code review has been widely adopted in the development process of both the proprietary and open-source software, which helps reveal bugs, improves the maintenance and quality of software before the code changes being merged into the source code repository. Our observation on code review requests from four large-scale projects reveals that around 20% changes cannot pass the first round code review and require non-trivial revision effort (i.e., risky changes). In addition, resolving these risky changes requires 3X more time and 1.6X more reviewers than the regular changes (i.e., changes pass the first code review) on average. This thesis presents the first study to characterize these risky changes and automatically identify these risky changes with machine learning classifiers. Evaluation on one proprietary project and three large-scale open-source projects (i.e., Qt, Android, and OpenStack) shows that our approach is effective in identifying risky code review requests. Chapter 6 of the thesis describes this work in detail.

Taken together, the results of the four studies provide evidence that machine learning can help improve software reliability practices such as statistical defect prediction, static bug detection, regression test, and code review.

## 1.1 Organization

The rest of this thesis are organized as follows. Chapter 2 presents the related work in typical software reliability practices. Chapter 3 shows our work on leveraging deep learning techniques to generate semantic features to improve software defect prediction. Chapter 4 describes our work on leveraging n-gram language models to build the probability distribution of token sequences from source code and detect software bugs that are missed by rule-based bug detection tools. Chapter 5 describes our work on leveraging machine learning classifiers to improve test case prioritization in regression test. Chapter 6 describes our work on leveraging machine learning classifiers to improve the traditional code review practice. Chapter 7 summarizes this thesis and provides an outlook into future work.

## 1.2 Thesis Scope

In this thesis, the proposed techniques for improving software reliability are only examined on projects written by object-oriented programming languages, i.e., Java and C++, they might not work for other programming languages, e.g., assembly languages and script languages. In addition, the proposed techniques to predict bugs (in Chapter 3) and detect bugs (in Chapter 4) are only examined on general software bugs collected from software history, they might not work for certain types of bugs, e.g., real-time bugs from embedded systems and concurrency bugs.

## 1.3 Related Publications

Earlier versions of the work done in this thesis have been published in the following papers (listed in chronological order). My role in these studies includes raw data collection and process, conducting experiments, results analysis, and writing. My contributions take up more than 80% of these studies.

- **Song Wang**, Adithya Abraham Philip, Chetan Bansal and Nachi Nagappan. Leveraging Change Intents for Characterizing and Identifying Risky Changes. 12 pages. *Under Submission.*
- **Song Wang**, Taiyue Liu, Jaechang Nam and Lin Tan. Deep Semantic Feature Learning for Software Defect Prediction. 27 pages. *To appear in the IEEE Transactions on Software Engineering (TSE 2018).*

- **Song Wang**, Jaechang Nam and Lin Tan. QTEP: quality-aware test case prioritization. *In Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (FSE 2017)*, pp. 523-534. (acceptance rate=24% 72/295)

- **Song Wang**, Taiyue Liu and Lin Tan. Automatically learning semantic features for defect prediction. *In Proceedings of the 38th International Conference on Software Engineering (ICSE 2016)*, pp. 297-308. (acceptance rate=19% 101/530)

- **Song Wang**, Devin Chollak, Dana Movshovitz-Attias and Lin Tan. Bugram: Bug Detection with N-gram Language Models. *In Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE 2016)*, pp. 708-719. (acceptance rate=19% 57/298)

The following papers were published in parallel to the abovementioned publications. These studies are not directly related to this thesis, however they explored how to leverage machine learning to improve other software analytic tasks, i.e., bug triage, test selection, and crowdsourced testing, etc.

- Jaechang Nam, **Song Wang**, Xi Yuan and Lin Tan. Designing Bug Detection Rules for Fewer False Alarms. *In Proceedings of the 40th International Conference on Software Engineering (ICSE 2018 Poster).*

- Juejie Wang, **Song Wang** and Qing Wang. Is There A "Golden" Feature Set for Static Warning Identification? - An Experimental Evaluation. *In Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM 2018).*

- Junjie Wang, Qiang Cui, **Song Wang** and Qing Wang. Domain Adaptation for Test Report Classification in Crowdsourced Testing. *In Proceedings of the 39th International Conference on Software Engineering (ICSE-SEIP 2017).*

- Qiang Cui, **Song Wang**, Junjie Wang, Yuanzhe Hu, Qing Wang and Mingshu Li. Multi-Objective Crowd Worker Selection in Crowdsourced Testing. *In Proceedings of the 29th International Conference on Software Engineering and Knowledge Engineering (SEKE 2017).*

- Junjie Wang, **Song Wang**, Qiang Cui and Qing Wang. Local-based Active Classification of Test Report to Assist Crowdsourced Testing. *In Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE 2016).*

- Junjie Wang, Qiang Cui, Qing Wang and **Song Wang**. Towards Effectively Test Report Classification to Assist Crowdsourced Testing. *In Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM 2016).*

- Wen Zhang, **Song Wang** and Qing Wang. KSAP: An Approach to Bug Report Assignment Using KNN Search and Heterogeneous Proximity. *Information and Software Technology (IST 2016).*

- Edmund Wong, Lei Zhang, **Song Wang**, Taiyue Liu and Lin Tan. DASE: Document-Assisted Symbolic Execution for Improving Automated Software Testing. *In Proceedings of the 37th International Conference on Software Engineering (ICSE 2015).*

- Xinye Tang, **Song Wang** and Ke Mao. Will This Bug-fixing Change Break Regression Testing? *In Proceedings of the 9th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM 2015).*

# Chapter 2

# Background and Related Work

There is much previous work related to our goal of leveraging machine learning techniques to improve existing software reliability practices. The research problems of this thesis corresponds to four research areas: *software defect prediction*, *static bug detection*, *regression test*, and *code review*. In this chapter, we provide an overview of the existing research of leveraging machine learning techniques to improve software reliability.

## 2.1 Software Defect Prediction

Table 2.1: Defect prediction tasks.

| Prediction Level | Within-project | Cross-project |
|---|---|---|
| File | WPDP | CPDP |
| Change | WCDP | CCDP |

Table 2.1 shows the investigated defect prediction tasks and their corresponding abbreviations in this thesis.

### 2.1.1 File-level Defect Prediction

Figure 2.1 presents a typical file-level defect prediction process that is adopted by existing studies [109, 138, 170, 193, 194, 211]. The first step is to label the data as buggy or clean based on post-release defects for each file. One could collect these post-release defects

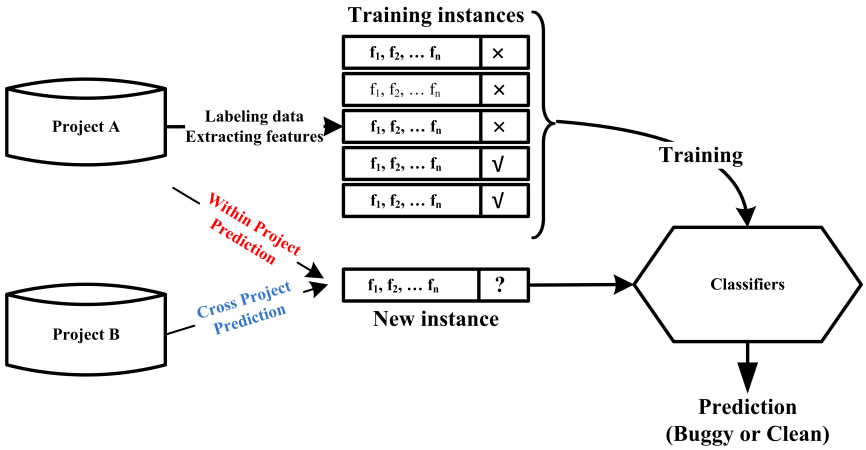



Figure 2.1: Defect Prediction Process

from a Bug Tracking System (BTS) via linking bug reports to its *bug-fixing* changes. Files related to these bug-fixing changes are considered as *buggy*. Otherwise, the files are labeled as *clean*. The second step is to collect the corresponding traditional features of these files. Instances with features and labels are used to train machine learning classifiers. Finally, trained models are used to predict new instances as buggy or clean.

We refer to the set of instances used for building models as a *training set*, whereas the set of instances used to evaluate the trained models is referred to as a *test set*. As shown in Figure 2.1, when performing within-project defect prediction (following existing work [193], we call this WPDP), the training and test sets are from the same project, i.e., project A. When performing cross-project defect prediction (following existing work [193], we call this CPDP), the prediction models are trained by a training set from project A (source), and a test set is from a different project, i.e., project B (target). In this thesis, for file-level defect prediction, we examine the performance of the learned DBN-based semantic features on both WPDP and CPDP.

There are many file-level software defect prediction techniques [82, 109, 125, 138, 168, 183, 189, 200, 220, 276, 326]. Most defect prediction techniques leverage features that are manually extracted from labeled historical defect data to train machine learning based classifiers [170]. Commonly used features can be divided into static code features and process features [169]. Code features include Halstead features [75], McCabe features [163], CK features [46], and MOOD features [81], which are widely examined and used for defect prediction. Recently, process features have been proposed and used for defect prediction.

Moser et al. [183] used the number of revisions, authors, past fixes, and ages of files as features to predict defects. Nagappan et al. [189] proposed code churn features, and showed that these features were effective for defect prediction. Hassan et al. [82] used entropy of change features to predict defects. Their evaluation on six projects showed that their proposed features can significantly improve results of defect prediction in comparison to other change features. Lee et al. [138] proposed 56 micro interaction metrics to improve defect prediction. Their evaluation results on three Java projects shown that their proposed features can improve defect prediction results compared to traditional features. Other process features, including developer individual characteristics [204] and collaboration between developers [138, 170, 211, 289], were also useful for defect prediction.

Many machine learning algorithms have been adopted for within-project defect prediction, including Support Vector Machine (SVM) [60], Bayesian Belief Network [11], Naive Bayes (NB) [263], Decision Tree (DT) [67, 118, 276], Neural Network (NN) [206, 218], and Dictionary Learning [109]. Elish et al. [60] evaluated the capability of SVM in predicting defect-prone software modules, and they compared SVM against eight statistical and machine learning models on four NASA datasets. Their results shown that SVM was promising for defect prediction. Amasaki et al. [11] proposed an approach to predicting the final quality of a software product by using the Bayesian Belief Network. Tao et al. [263] proposed a Naive Bayes based defect prediction model, they evaluated the proposed approach on 11 datasets from the PROMISE defect data repository. Wang et al. [276] and Khoshgoftaar et al. [118] examined the performance of Tree-based machine learning algorithms on defect prediction, their results suggested that Tree-based algorithms could help defect prediction. Jing et al. [109] introduced the dictionary learning technique to defect prediction. They proposed a cost-sensitive dictionary learning based approach to improving defect prediction.

Due to the lack of data, it is often difficult to build accurate models for new projects. Some work [126, 272, 325] has been done on evaluating cross-project defect prediction against within-project defect prediction and shows that cross-project defect prediction is still a challenging problem. The main issue that degrades the performance of cross-project defect prediction is that, the distribution of data metrics is not shared between projects. To address this issue, cross-project defect prediction models are trained by using data from other projects. He et al. [84] showed the feasibility to find the best cross-project models among all available models to predict testing projects. Watanabe et al. [286] proposed an approach for CPDP by transforming the target dataset to the source dataset by using the average feature values. Turhan et al. [272] proposed to use a nearest-neighbor filter to improve cross-project defect prediction. Nam et al. [193] and Jing et al. [108] used different approaches to address the heterogeneous data problem in cross-project defect prediction.

Xia et al. [299] proposed HYDRA, which leverages genetic algorithm phase and ensemble learning (EL) to improve cross-project defect prediction. Their approach requires massive training data and a portion (5%) of labelled data from test data to build and train their model, while in real-world practice, it's very expensive to obtain labelled data from test projects, which requires developers to manually inspection, and the ground truth might be unguaranteed. Nam et al. [194] proposed TCA+, which adopted a state-of-the-art technique called Transfer Component Analysis (TCA) and optimized TCA's normalization process to improve cross-project defect prediction. In this work, we select TCA+ as our baseline for cross-project defect prediction, since TCA+ is more practical.

### 2.1.2 Change-level Defect Prediction

Change-level defect prediction can predict whether a change is buggy at the time of the commit so that it allows developers to act on the prediction results as soon as a commit is made. In addition, since a change is typically smaller than a file, developers have much less code to examine in order to identify defects. However, for the same reason, it is more difficult to predict buggy changes accurately.

Similar to file-level defect prediction, change-level defect prediction also consists of the following processes:

- Labeling process: Labeling each change as buggy or clean to indicate whether the change contains bugs.
- Feature extracting process: Extracting the features to represent the changes.
- Model building and testing process: Building a prediction model with the features and labels and then using the model to predict testing data.

Different from labeling file-level defect data, labeling change-level defect data requires further linking of bug-fixing changes to bug-introducing changes. A line that is deleted or changed by a bug-fixing change is a faulty line, and the most recent change that introduced the faulty line is considered a bug-introducing change. We could identify the bug-introducing changes by a *blame* technique provided by a Version Control System (VCS), e.g., git or SZZ algorithm [124]. Such blame techniques are widely used in existing studies [104, 124, 178, 256, 311]. In this work, the bug-introducing changes are considered as buggy, and other changes are labeled clean. Note that, not all projects have a well maintained BTS, and we consider changes whose commit messages contain the keyword "fix"

as bug-fixing changes by following existing studies [104, 256]. In this thesis, similar to the file-level defect prediction, we also examine the performance of DBN-based features on both change-level within-project defect prediction (WCDP) and change-level cross-project defect prediction (CCDP).

For change-level defect prediction, Mockus and Weiss [178] and Kamei et al. [115] predicted the risk of a software change by using change measures, such as the number of subsystems touched, the number of files modified, the number of lines of added code, and the number of modification requests. Kim et al. [122] used the identifiers in added and deleted source code and the words in change logs to classify changes as being defect-prone or clean. Tan et al. [256] improved the change classification techniques to conduct online defects prediction for imbalanced data, which applied and adapted time sensitive change classification and online change classification to address the incorrect evaluation presented by cross-validation, and applied the resampling techniques and updatable classification to improve classification performance. Jiang et al. [104] and Xia et al. [300] built separate prediction models for each developer to predict software defects at change level. Nagappan et al. [189] leveraged the architectural dependency and churn measures to predict bugs. Change classification can also predict whether a commit is buggy or not [210,215]. Recently, Kamei et al. [114] empirically studied the feasibility of change-level defect prediction in a cross-project context.

The main differences between our approach and existing approaches for defect prediction are as follows. First, existing approaches to defect prediction are based on manually encoded traditional features which are not sensitive to programs' semantic information, while our approach automatically learns semantic features using DBN and uses these features to perform defect prediction tasks. Second, since our approach requires only the source code of the training and test projects, it is suitable for both within-project defect prediction and cross-project defect prediction.

### 2.1.3 Deep Learning and Semantic Feature Generation in Software Engineering

Recently, deep learning algorithms have been adopted to improve research tasks in software engineering. Yang et al. [311] proposed an approach that leveraged deep learning to generate features from existing features and then used these new features to build defect prediction models. This work was motivated by the weaknesses of logistic regression (LR), which is that LR cannot combine features to generate new features. They used a

DBN to generate features from 14 traditional change level features, including the following: the number of modified subsystems, modified directories, modified files, code added, code deleted, line of code before/after the change, files before/after the change, and several features related to developers' experience [311].

Our work differs from the above study mainly in three aspects. First, we use a DBN to learn semantic features directly from the source code, while features generated from their approach are relations among existing features. Since the existing features cannot distinguish between many semantic code differences, the combination of these features would still fail to capture semantic code differences. For example, if two changes add the same line at different locations in the same file, the traditional features cannot distinguish between the two changes. Thus, the generated new features, which are combinations of the traditional features, would also fail to distinguish between the two changes. Second, we evaluate the effectiveness of our generated features using different classifiers for both within-project and cross-project defect prediction, while they only use LR for within-project defect prediction. Third, we focus on both file and change-level defect prediction, while they only work on change-level defect prediction.

There also many existing studies that leverage deep learning techniques to address other problems in software engineering [71, 72, 107, 110, 133, 140, 156, 185, 209, 225, 290, 304, 317]. Mou et al. [185] used deep learning to model programs and showed that deep learning can capture the programs' structural information. Deep learning has also been used for malware classification [209,317], test report classification [110], link prediction in a developer online forum [304], software traceability [107], etc.

How to explain deep learning results is still a challenging question to the AI community. To interpret deep learning models, Andrej et al. [116] used character-level language models as an interpretable testbed to explain the representations and predictions of a Recurrent Neural Network (RNN). Their qualitative visualization experiments demonstrate that RNN models could learn powerful and often interpretable long-range interactions from real-world data. Radford et al. [219] focused on understanding the properties of representations learned by byte-level recurrent language models for sentiment analysis. Their work reveals that there exists a sentiment unit in the well-trained RNNs (for sentiment analysis) that has a direct influence on the generative process of the model. Specifically, simply fixing its value to be positive or negative can generate samples with the corresponding positive or negative sentiment. The above studies show that to some extent deep learning models are interpretable. However, these two studies focused on interpreting RNNs on text analysis. In this work we leverage a different deep learning model, i.e., the deep belief network (DBN), to analyze the ASTs of source code. DBN adopts different architectures and learning processes from RNNs. For example, an RNN (e.g., LSTM) can, in principle, use

its memory cells to remember long-range information that can be used to interpret data it is currently processing, while a DBN does not have such memory cells (details are provided in Section 3.2.1). Thus, it is unknown whether DBN models share the same property (i.e., interpretable) as RNNs.

Many studies used a topic model [32] to extract semantic features for different tasks in software engineering [45,150,197,200,254,302]. Nguyen et al. [200] leveraged a topic model to generate features from source code for within-project defect prediction. However, their topic model handled each source file as one unordered token sequence. Thus, the generated features cannot capture structural information in a source file.

This work shows that DBN is applicable for learning semantic features to improve software bug prediction, following this work, researches have explored the effectiveness of other deep learning algorithms on defect prediction [52,56,139,145,288]. For example, Li et al. [139] proposed to leverage Convolutional Neural Network (CNN) to generate semantic features for improving software defect prediction. Evaluation on seven open-source projects showed the CNN-based defect prediction approaches can outperform our DBN-based defect prediction model. Wen et al. [145] used Recurrent Neural Network (RNN) to learning features for change-level defect prediction tasks. Their evaluation showed the RNN-based features could outperform traditional change-level defect prediction features.

## 2.2 Static Bug Detection

### 2.2.1 Static Bug Detection Tools

Many static code analysis techniques have been developed to detect bugs based on bug patterns [4,5,10,30,38,41,70,96,134,143,144,179,207,230,243,255,269,278,285,287,291,301]. Existing static bug detection techniques could be divided into two categories according to how bug patterns are designed, i.e., manually identified bug patterns and automatically mined bug patterns from source code.

Two widely used open-source bug detection tools for Java language, FindBugs [4] and PMD [5], detect real bugs based on manually designed bug patterns by their contributors. Most manually designed patterns focus on language-specific common bugs in software projects. Such as buffer overflow, race conditions, and memory allocation errors, etc. Some other studies leverage particular types of bug patterns to detect special bugs. For example, Chen et al. [41] proposed six anti-patterns and leveraged these rules to detect log

related bugs. Palomba et al. [207] propose to detect five different code smells based on five well-summarized code patterns.

For detecting project-specific bugs, many approaches leveraged rules that are mined from specific projects. Li et al. [144] developed PR-Miner to mine programming rules from C code and detect violations using frequent itemset mining algorithm. Benjamin et al. [30] proposed DynaMine which used association rule mining to extract simple rules from software revision histories and detect defects related to rule violations. Wasylkowski et al. [285] and Gruska et al. [70] proposed to detect object usage anomalies by combining frequent itemset and object usage graph models.

### 2.2.2 Statistical Language Models

Statistical language models have been successfully used for tasks including code completion [24, 225, 290], fault localization and coding style consistency checking [10, 85, 242]. Hindle et al. [88] leveraged n-gram language models to show that source code has high repetitiveness. Han et al. [78] presented an algorithm to infer the next token by using a Hidden Markov Model. Pradel et al. [213] proposed an approach to generating object usage specifications based on a Markov Model. Yusuke et al. [202] leveraged n-gram models to generate pseudo-code from software source code. Ray et al. [224] used n-gram models to study language statistics of buggy code, which showed that software buggy lines are more unnatural than non-buggy lines. They also proposed a defect prediction model based on n-gram models. Specifically, they built n-gram models on an old version of a software project, and then used entropy from the n-gram models to estimate the naturalness of the source code lines in a later version, i.e., source code lines with higher entropy values are flagged as buggy lines. There are three main differences between their approach and Bugram. First, given a software project, Bugram directly builds n-gram models on it and detect bugs in this project, while their tool requires an old version of this project as training data. Second, they build n-gram models at the token level, while we build n-gram models at a higher level (e.g., statements, method calls, and control flows) (Section 4.3.1) aiming to detect semantic bugs more effectively. Third, their approach leverages entropy while Bugram uses probability to detect bugs. Using probability and entropy to rank token sequences are two different approaches [158]. The entropy used in [224] combines probability and sequence length. It may worth comparing using entropy versus probability for detecting bugs in the future.

White et al. [290] and Raychev et al. [225] investigated the effectiveness of language models, i.e., n-gram and recurrent neural network (RNN), for code completion. Bavishi et

al. [24] proposed a RNN-based framework to recover natural identifier names from minified JavaScript code. Movshovitz-Attias et al. [186] leveraged n-gram models to predict class comments for program source file documents. Campbell et al. [35] built n-gram models with historical correct source code to locate the cause of syntax errors. Some studies used n-gram token sequences instead of n-gram models to solve software engineering tasks. Nessa et al. [196] and Yu et al. [316] leveraged n-gram token sequences to help software fault localization. Lal et al. [132] combined n-grams with information retrieval techniques to improve fault localization. Sureka et al. [251] leveraged n-gram-based features to detect duplicated bug reports. Hsiao et al. [96] proposed to use n-gram token sequences and *tf-idf-style* measures to detect code clone and related bugs. In contrast, Bugram is not limited to detecting clone bugs.

After our study, applications of statistical language models on other topics have been examined, e.g., program synthesis [53, 305, 322] and code analysis [51, 106, 237, 275], and bug detection [214].

## 2.3 Regression Testing Techniques

A typical TCP technique reorders the execution sequence of test cases based on a certain objective, e.g., fault-detection rate [234]. Specifically, TCP can be formally defined as follows: given a test suite $T$ and the set of its all possible permutations of test $PT$, TCP techniques aim to find a permutation $P' \in PT$ that $(\forall P'')\ (P'' \in PT)\ (P'' \neq P'),\ f(P') \geq f(P'')$, where $f$ is the objective function.

Most existing TCPs leverage coverage information, e.g., dynamic code coverage (dynamic call graph) from the last run of test cases [102, 233], static code coverage (static call graph) from static code analysis [43, 101, 166, 244, 266]. The commonly used coverage criteria include statement, method, and branch coverages. In this work, we choose to examine statement and method coverages, since previous work has shown that statement and method coverages are more effective than other coverage criteria [153, 154, 235]. For coverage-based TCP techniques, there are two widely used prioritization strategies, i.e., total strategy and additional strategy [142,234,312]. The total coverage strategy schedules the execution order of test cases based on the total number of statements or methods covered by these test cases. Whereas, the additional coverage strategy reorders the execution sequence of test cases based on the number of statements or methods that are not covered by already ordered test cases but covered by the unordered test cases.

Many regression testing techniques have been proposed for improving test efficiency by test case prioritization [42, 43, 69, 101, 102, 121, 142, 166, 233–235, 258, 266, 312, 313, 319],

test case selection [21, 232, 241], and test case reduction [31, 240, 321]. In terms of TCP techniques, Rothermel et al. [234] first presented a family of prioritization techniques including both the total and additional test prioritization strategies using various coverage information. They also proposed the metric APFD for assessing TCPs. Along this line, TCP techniques with different prioritization strategies, coverage criteria and constraints are proposed.

A majority of existing TCPs leverage code coverage information to rank test cases. Dynamic code coverage from the last execution is widely used in existing TCP techniques [102, 142, 233, 312]. Another widely used code coverage is static code coverage, which is estimated from static analysis rather than gathered through instrumentation and execution. Mei et al. [166] are the first to prioritize test cases with static coverage information. Later, Thomas et al. [266] proposed a static test case prioritization method using topic models. Some other TCP techniques [50, 99, 201, 235] leveraged similarity between test cases and source code to prioritize test cases. Specifically, Saha et al. [235] proposed an information retrieval approach for regression testing prioritization based on program changes. Noor et al. [201] proposed a similarity-based approach for test case prioritization using historical failure data.

Instead of using the coverage or similarity information between source code and test cases, some approaches use other software process information as the proxy to rank test cases [15, 61, 130, 173–175, 270, 273, 315, 319]. Arafeen et al. [15] used software requirements to group and rank test cases. Mirarab et al. [175] and Zhang et al. [319] proposed to leverage the source code metric (i.e., McCabe) based coverage information of test cases to prioritize regression test cases. Engstrom et al. [61] selected a small set of test cases for regression testing selection based on previously fixed faults. Their work required previously revealed bugs within a given period. For projects that do not have well-maintained bugs or new projects (no past bugs are available), their approach cannot work. While our proposed `QTEP` does not have such limitation, since it focuses on potentially unrevealed faults. Laali et al. [130] proposed to utilize the locations of revealed faults of the executed test cases to rank the remaining test cases. Different from `QTEP`, they used injected faults, and the performance of their approach on real-world faults is unknown.

Yu et al. [315] proposed the fault-based prioritization of test cases that is designed by using the fault-based test case generation models. Different from `QTEP`, their approach assumed the fault-detecting ability of each test case is available. Miranda et al. [174] proposed scope-aided TCP for testing the reused code by using possible constraints delimiting the new input domain scope. On the contrary, `QTEP` is not limited to testing the reused code.

After this study, Liang et al. [147] have applied test case prioritization to continuous integration environments at Google. Their evaluation on a large data set from Google showed that test case prioritization techniques can lead to cost-effectiveness improvements in the continuous integration process.

## 2.4 Code Review

Code review is a manual inspection of source code by humans, which aims at identifying potential defects and quality problems in the source code before its deployment in a live environment [19,25–27,63,68,85,131,157,164,165,172,182,212,246,248,267,274]. In recent years, Modern Code Review (MCR) has been developed as a tool-based code review system and becomes popular and widely used in both proprietary software (e.g., Google, Cisco, and Microsoft) and open-source software (e.g., Android, Qt, and LibreOffice) [76]. Many studies have examined the practices of code review.

Stein et al. [248] explored the distributed, asynchronous code inspections. They studied a tool that allowed participants at separate locations to discuss faults. Porter et al. [212] reported on a review of studies on code inspection in 1995 that examined the effects of different factors on code inspections. Laitenburger [131] surveyed code inspection methods, and presented a taxonomy of code inspection techniques. Votta [274] found that 20% of the interval in a "traditional inspection" is wasted due to scheduling. Rigby et al. [227–229] have done extensive work examining code review practices in OSS development. Hellendoorn et al. [85] used language models to quantitatively evaluate the influence of stylistic properties of code contributions on the code review process and outcome. Sutherland and Venolia [252] conducted a study at Microsoft regarding using code review data for later information needs. Bacchelli & Bird find that understanding of the code and the reason for a change is the most important factor in the quality of code reviews [19]. Some other studies focus on accelerating code review process by recommending reviewers [267], decomposing code review changes with multiple changesets [23, 73, 128, 265].

In this thesis, we explore the feasibility of accelerating code review by identifying the risky code changes that require multiple rounds of code review or are reverted.

## 2.5 Other Applications of Machine Learning for Improving Software Reliability

Besides the four topics mentioned above, machine learning technologies have also been leveraged to improve other software reliability practices, such as symbolic execution [44, 141, 296], test generation [148, 149, 159, 176, 249], automatic bug repair [119, 137, 151, 152, 181, 303, 307, 309], comment generation [97, 98, 294, 295], bug report triage [14, 100, 187, 282, 283, 306, 324], false static analysis alerts filtering [79, 277], and fault localization [18, 94, 135, 226, 297, 298, 323]. For example, for accelerating symbolic execution, our previous work [296] leveraged NLP technologies to extract input constraints from program documents to provide a better guidance during symbolic execution. Along this line, Chen et al. [44] proposed to accelerate symbolic execution with code transformation and Li et al. [141] proposed to apply symbolic execution to complex program by using machine learning based constraint solving.

Note that this thesis is not ambitious to improve all existing software reliability practices, we demonstrate how to take the advantages of machine learning technologies in knowledge representation, learning, natural language processing, classification, etc., to improve software reliability practices.

# Chapter 3

# Leveraging Deep Learning to Improve Defect Prediction

This chapter presents our approach to improving software defect prediction by using semantic features learnt with deep learning algorithms, which was presented at the 38th International Conference on Software Engineering (ICSE'16 [280]) and IEEE Transactions on Software Engineering (TSE'18 [279]).

## 3.1 Motivation

Software defect prediction techniques [82, 104, 109, 125, 138, 168, 183, 189, 220, 276, 326] have been proposed to detect defects and reduce software development costs. Defect prediction techniques build models using software history data and use the developed models to predict whether new instances of code regions, e.g., files, changes, and methods, contain defects.

The efforts of previous studies toward building accurate prediction models can be categorized into the following two approaches: The first approach is manually designing new features or new combinations of features to represent defects more effectively, and the second approach involves the application of new and improved machine learning based classifiers. Researchers have manually designed many features to distinguish defective files from non-defective files, e.g., Halstead features [75] based on operator and operand counts; McCabe features [163] based on dependencies; CK features [46] based on function and inheritance counts, etc.; MOOD features [81] based on polymorphism factor, coupling factor,

```
1  public void copy(Directory to, String
       src, String dest) throws IOException
        {
2    IndexOutput os = to.createOutput(dest);
3    IndexInput is = openInput(src);
4    IOException priorException = null;
5
6    try {
7      is.copyBytes(os, is.length());
8    } catch (IOException ioe) {
9      priorException = ioe;
10   }
11   finally {
12     IOUtils.closeSafely(priorException
           , os, is);
13   }
14 }
```

(a) Original buggy code snippet.

```
1  public void copy(Directory to, String
       src, String dest) throws IOException
        {
2    IndexOutput os = null;
3    IndexInput is = null;
4    IOException priorException = null;
5    try {
6      os = to.createOutput(dest);
7      is = openInput(src);
8      is.copyBytes(os, is.length());
9    } catch (IOException ioe) {
10     priorException = ioe;
11   } finally {
12     IOUtils.closeSafely(priorException
           , os, is);
13   }
14 }
```

(b) Code snippet after fixing the bug.

Figure 3.1: A motivating example from Lucene.

etc.; process features [104, 220] (including number of lines of code added, removed, meta features, etc.); and object-oriented features [22, 57, 161].

Traditional features mainly focus on the statistical characteristics of programs and assume that buggy and clean programs have distinguishable statistical characteristics. However, our observations on real-world programs show that existing traditional features often cannot distinguish programs with different semantics. Specifically, program files with different semantics can have traditional features with similar or even the same values. For example, Figure 3.1 shows an original buggy version, i.e., Figure 3.1(a), and a fixed clean version, i.e., Figure 3.1(b), of a method from Lucene. In the buggy version, there is an IOException when initializing variables `os` and `is` before the try block. The buggy version can lead to a memory leak[1] and has already been fixed by moving the initializing statements into the try block in Figure 3.1(b). Using traditional features to represent these two code snippets, e.g., code complexity features, their feature vectors are identical. This is because these two code snippets have the same source code characteristics in terms of complexity, function calls, raw programming tokens, etc. However, the semantic information in these two code snippets is significantly different. Specifically, the contextual information of the two variables, i.e., `os` and `is`, in the two versions is different. Features that can distinguish such semantic differences are needed for building more accurate prediction models.

To bridge the gap between the programs' semantic information and defect prediction features, this thesis proposes leveraging a powerful representation-learning algorithm,

[1] `https://issues.apache.org/jira/browse/LUCENE-3251`

namely, deep learning [93], to *learn semantic representations of programs automatically*. Specifically, we use the deep belief network (DBN) [92] to automatically learn features from token vectors extracted from source code, and then we utilize these features to build and train defect prediction models.

To use a DBN to learn features from code snippets, we first convert the code snippets into vectors of tokens with the structural and contextual information preserved, and then we use these vectors as the input into DBN for generating features. For the two code snippets presented in Figure 3.1, the two input vectors are `[..., IndexOutput, createOutput(), IndexInput, openInput(), IOException, try, ...]` and `[..., IndexOutput, IndexInput, IOException, try, createOutput(), openInput()...]` respectively (details regarding the token extraction are provided in Section 3.3.1). As the vectors of these two code snippets are different, DBN will *automatically* learn features that can distinguish them.

We examine our DBN-based approach to generating semantic features on both file-level defect prediction tasks (i.e., predict which files in a release are buggy) and change-level defect prediction tasks (i.e., predict whether a code commit is buggy), because most of the existing approaches to defect prediction are on these two levels [17, 84, 104, 194, 222, 256, 272, 299, 300]. Focusing on these two different defect prediction tasks enables us to extensively compare our proposed technique with state-of-the-art defect prediction features and techniques. For file-level defect prediction, we generate DBN-based semantic features by using the complete Abstract Syntax Trees (AST) of the source files, while for change-level defect prediction, we generate the DBN-based features by using tokens extracted from code changes. In addition, most defect prediction studies have been conducted in one or two settings, i.e., within-project defect prediction [104, 178, 256, 300] and/or cross-project defect prediction [84, 194, 272, 299]. Thus, we evaluate our approach in these two settings as well.

## 3.2 Background

### 3.2.1 Deep Belief Network

A deep belief network is a generative graphical model that uses a multi-level neural network to learn a representation from the training data that could reconstruct the semantic and content of the training data with a high probability [28]. DBN contains one *input layer* and several *hidden layers*, and the top layer is the output layer that contains final features

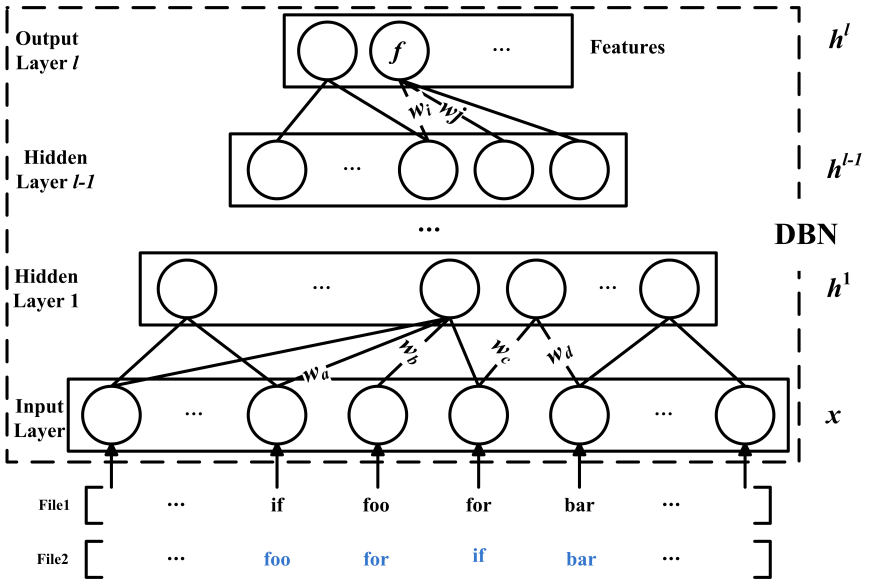



Figure 3.2: Deep belief network architecture and input instances of File1.java and File2.java. Although the token sets of these two files are identical, the different structural and contextual information between tokens enables DBN to generate different features to distinguish these two files.

to represent input data as shown in Figure 3.2. Each layer consists of several stochastic nodes. The number of hidden layers and the number of nodes in each layer vary depending on users' demand. In this study, the size of learned semantic features is the number of nodes in the top layer. The idea of DBN is to enable the network to reconstruct the input data using generated features by adjusting weights between nodes in different layers.

DBN models the joint distribution between input layer and the hidden layers as follows:

$$P(x, h^1, ..., h^l) = P(x|h^1)(\prod_{k=1}^{l} P(h^k|h^{k+1})) \tag{3.1}$$

where $x$ is the data vector from input layer, $l$ is the number of hidden layers, and $h^k$ is the data vector of $k^{th}$ layer ($1 \leq k \leq l$). $P(h^k|h^{k+1})$ is a conditional distribution for the adjacent $k$ and $k+1$ layers.

To calculate $P(h^k|h^{k+1})$, each pair of two adjacent layers in DBN are trained as a Restricted Boltzmann Machines (RBM) [28]. An RBM is a two-layer, undirected, bipartite graphical model where the first layer consists of observed data variables, referred to as *visible nodes*, and the second layer consists of latent variables, referred to as *hidden nodes*.

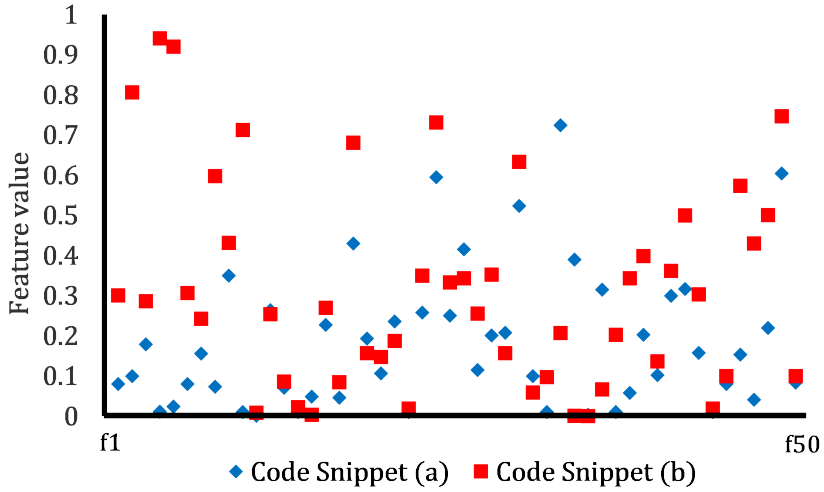



Figure 3.3: The distribution of DBN-based features of the example code snippets shown in Figure 3.1.

$P(h^k|h^{k+1})$ can be efficiently calculated as:

$$P(h^k|h^{k+1}) = \prod_{j=1}^{n_k} P(h_j^k|h^{k+1}) \tag{3.2}$$

$$P(h_j^k = 1|h^{k+1}) = sigm(b_j^k + \sum_{a=1}^{n_{k+1}} W_{aj}^k h_a^{k+1}) \tag{3.3}$$

where $n_k$ is the number of nodes in layer $k$, $sigm(c) = \frac{1}{1+e^{-c}}$, $b$ is a bias matrix, $b_j^k$ is the bias for node $j$ of layer $k$, and $W^k$ is the weight matrix between layer $k$ and layer $k+1$. $sigm$ is the sigmod function, which serves as the activation function to update the hidden units. We use the sigmod function because it outputs a more smooth range of nonlinear values with a relatively simple computation [77].

DBN automatically learns $W$ and $b$ matrices using an iteration process. $W$ and $b$ are updated via log-likelihood stochastic gradient descent:

$$W_{ij}(t+1) = W_{ij}(t) + \eta \frac{\partial log(P(v|h))}{\partial W_{ij}} \tag{3.4}$$

$$b_k^o(t+1) = b_k^o(t) + \eta \frac{\partial log(P(v|h))}{\partial b_k^o} \tag{3.5}$$

where $t$ is the $t^{th}$ iteration, $\eta$ is the learning rate, $P(v|h)$ is the probability of the visible layer of an RBM given the hidden layer, $i$ and $j$ are two nodes in different layers of the RBM, $W_{ij}$ is the weight between the two nodes, and $b_k^o$ is the bias on the node $o$ in layer $k$.

To train the network, one first initializes all $W$ matrices between two layers via RBM and sets the biases $b$ to $\mathbf{0}$. They can be well-tuned with respect to a specific criterion, e.g., the number of training iterations, error rate between reconstructed input data and original input data. In this study, we use the number of training iterations as the criterion for tuning $W$ and $b$. The well-tuned $W$ and $b$ are used to set up a DBN for generating semantic features for both training and test data. Also, we discuss how these parameters affect the performance of learned semantic features in Section 3.4.5.

DBN model generates features with more complex network connections. These network connections enable DBN models to generate features with multiple levels of abstraction and high-level semantics. DBN features are weighted combinations/vectors of input nodes, which may represent patterns of the usages of input nodes (e.g., methods, control-flow nodes, etc.). We believe such DBN-based features can help distinguish the semantics of different source code snippets, which traditional features cannot handle well. For example, Figure 3.3 shows the distribution of the DBN-based semantic features of the two code snippets shown in Figure 3.1. Specifically, we use the trained DBN model on project Lucene (details are in Section 3.4.8) to generate a feature set that contains 50 different features for each of the two code snippets. As we can see in the figure, the distributions of features of the two code snippets are different. Specifically, most of the features of code snippet shown in Figure 3.1(b) have larger values than those of the features of code snippet shown in Figure 3.1(a). Thus, the new features are capable of distinguishing these two code snippets with a proper classifier.

## 3.3 Approach

In this work, we use DBN to generate semantic features automatically from source files and code changes and further leverage these features to improve defect prediction. Figure 3.4 illustrates the workflow of our approach to generating features for both file-level defect prediction (inputs are source files) and change-level defect prediction (inputs are source code changes). Specifically, for file-level defect prediction, our approach takes AST node tokens from the source code of the training and test source files as the input, and generates semantic features from them. Then, the generated semantic features are used to build the models for predicting defects. Note that for change-level defect prediction, the input data

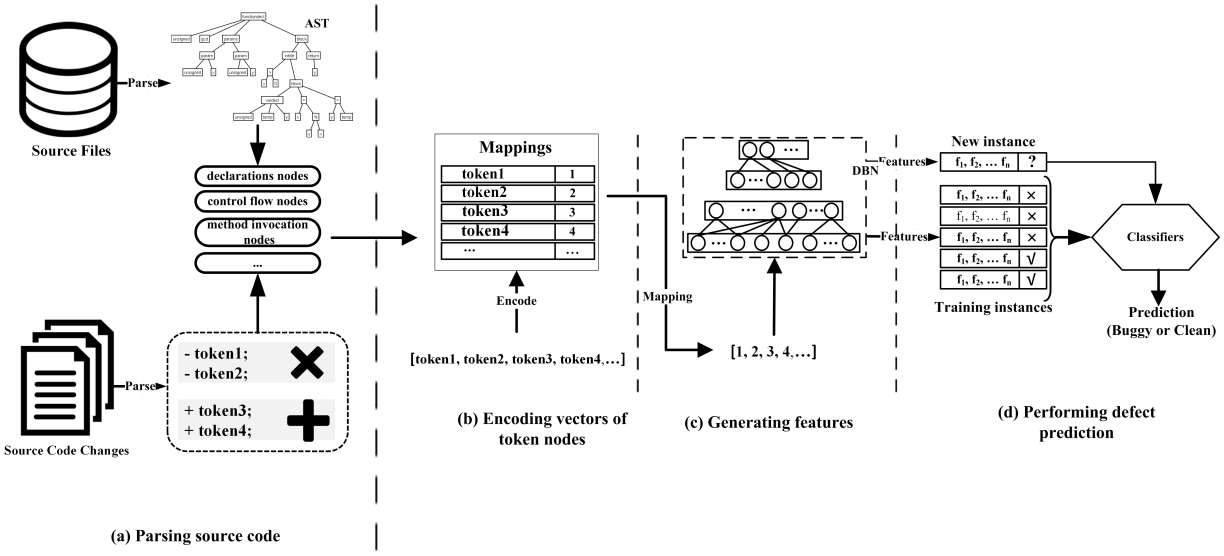



Figure 3.4: Overview of our DBN-based approach to generating semantic features for file-level and change-level defect prediction.

```
1  --- a/solr/src/java/org/apache/solr/handler/component/QueryComponent.java
2  +++ b/solr/src/java/org/apache/solr/handler/component/QueryComponent.java
3  @@ -217,14 +217,8 @@ public class QueryComponent extends SearchComponent
4     for (String groupByStr : funcs) {
5         QParser parser = QParser.getParser(groupByStr, "func", rb.req);
6         Query q = parser.getQuery();
7  -      SolrIndexSearcher.GroupCommandFunc gc;
8  -      if (groupSort != null) {
9  -         SolrIndexSearcher.GroupSortCommand gcSort = new SolrIndexSearcher.
       GroupSortCommand();
10 -         gcSort.sort = groupSort;
11 -         gc = gcSort;
12 -       } else {
13 -         gc =  new SolrIndexSearcher.GroupCommandFunc();
14 -       }
15 +       SolrIndexSearcher.GroupCommandFunc gc = new SolrIndexSearcher.GroupCommandFunc
       ();
16 +       gc.groupSort = groupSort;
17         if (q instanceof FunctionQuery) {
18         gc.groupBy = ((FunctionQuery)q).getValueSource();
```

Figure 3.5: A change example from Lucene.

to our DBN-based feature generation approach are changed code snippets. Since building AST for an incomplete code snippet is challenging, in this work we propose a heuristic approach to extracting important structural and context information from code change snippets (details are in Section 3.3.1). DBN requires input data in the form of integer

vectors, to satisfy this requirement, we first build a mapping between integers and tokens and then convert the token vectors to integer vectors, to generate semantic features, we first use the integer vectors of the training set to build and train a DBN. Then, we use the trained DBN to automatically generate semantic features from the integer vectors of the training and test sets. Finally, based on the generated semantic features, we build defect prediction models from the training set, and evaluate their performance on the test set.

Our approach consists of four major steps: 1) parsing source code (source files for file-level defect prediction and changed code snippets for change-level defect prediction) into tokens, 2) mapping tokens to integer identifiers, which are the expected inputs to the DBN, 3) leveraging DBN to automatically generate semantic features, and 4) building defect prediction models and predicting defects using the learned semantic features of the training and test data.

### 3.3.1 Parsing Source Code

#### Parsing Source Code for Files

For file-level defect prediction tasks, we utilize the Java Abstract Syntax Tree (AST) to extract syntactic information from source code files. Specifically, three types of AST node are extracted: 1) nodes of method invocations and class instance creations, e.g., in Figure 3.2, method `createOutput()` and `openInput()` are recorded as their method names, 2) declaration nodes, i.e., method declarations, type declarations, and enum declarations, and 3) control-flow nodes such as `while` statements, `catch` clauses, `if` statements, `throw` statements, etc. Control-flow nodes are recorded as their statement types, e.g., an `if` statement is simply recorded as `if`. In summary, for each file, we obtain a vector of tokens of the three categories. We exclude AST nodes that are not one of these three categories, such as assignment and intrinsic type declaration, because they are often method-specific or class-specific, which may not be generalizable to the whole project. Adding them may dilute the importance of other nodes.

Since the names of methods, classes, and types are typically project-specific, methods of an identical name in different projects are either rare or of different functionalities. Thus, for cross-project defect prediction, we extract all three categories of AST nodes, but for the AST nodes in categories 1) and 2), instead of using their names, we use their AST node types such as `method declarations` and `method invocations`. Take project `xerces` as an example. As an XML parser, it consists of many methods named `getXXX` and `setXXX`, where `XXX` refers to XML-specific keywords including `charset`, `type`, and `href`. Each of

these methods contains only one method invocation statement, which is in form of either `getAttribute(XXX)` or `setAttribute(XXX)`. Methods `getXXX` and `setXXX` do not exist in other projects, while `getAttribute(XXX)` and `setAttribute(XXX)` have different meanings in other projects, so using the names `getAttribute(XXX)` or `setAttribute(XXX)` is not helpful. However, it is useful to know that method declaration nodes exist, and only one method invocation node is under each of these method declaration nodes, since it might be unlikely for a method with only one method invocation inside to be buggy. In this case, compared with using the method names, using the AST node types `method declaration` and `method invocation` is more useful since they can still provide partial semantic information.

### Parsing Source Code for Changes

Different from file-level defect prediction data, i.e., program source files, for which we could build ASTs and extract AST token vectors for feature generation, change-level defect prediction data are changes that developers made to source files, whose syntax information is often incomplete. These changes could have different locations and include code additions and code deletions, which are syntactic incomplete. Thus, building ASTs for these changes is challenging. In this study, for tokenizing changes, instead of building ASTs, we tokenize a change by considering the `code addition`, the `code deletion`, and the `context code` in the change. Code additions are the added lines in a change, code deletions are the deleted lines in a change, and the code around these additions or deletions is considered the context code. For example, Figure 3.5 shows a real change example from project Lucene. In this change, the code addition contains lines 15 and 16, the code deletion contains lines 7 to 14, and the context contains lines 4 to 6, 17, and 18. Note that the contents of the source code lines in the additions, deletions, and context code are often overlapping, e.g., the deleted line 7 and the added line 15 contain the same line of code for class instance creation, i.e., `SolrIndexSearcher.GroupCommandFunc gc;`. Thus, to distinguish these lines, we add different prefixes to the raw tokens that are extracted from different types of changed code. Specifically, for the addition, we use prefix "added_", for the deletion, we use prefix "deleted_", and for the context code, we use prefix "context_". The details of the three types of tokens extracted from the example change (in Figure 3.5) are shown in Table 3.1.

From Table 3.1, we could observe that different types of tokens from the changed code snippets contain different information. For example, the context nodes show that the code is changed inside a `for` loop, an `if` statement is removed from the source code in the deletions, and an instantiation of class `GroupCommandFunc` was created in the additions.

Table 3.1: Three types of tokens extracted from the example change shown in Figure 3.5.

| | |
|---|---|
| added | added_SolrIndexSearcher.GroupCommandFunc<br>added_gc<br>added_SolrIndexSearcher.GroupCommandFunc<br>added_gc.groupSort<br>added_groupSort |
| deleted | deleted_SolrIndexSearcher.GroupCommandFunc<br>deleted_gc<br>deleted_if<br>deleted_groupSort<br>deleted_Notnull<br>deleted_SolrIndexSearcher.GroupSortCommand<br>deleted_gcSort<br>deleted_SolrIndexSearcher.GroupSortCommand<br>deleted_gcSort.sort<br>deleted_groupSort<br>deleted_gc<br>deleted_gcSort<br>deleted_delete else<br>deleted_gc<br>deleted_SolrIndexSearcher.GroupCommandFunc |
| context | context_for<br>context_QParser<br>context_parser<br>context_QParser.getParser<br>context_Query<br>context_q<br>context_parser.getQuery<br>context_if<br>context_q<br>context_FunctionQuery<br>context_gc.groupBy<br>context_FunctionQuery<br>context_q.getValueSource |

Intuitively, DBN-based features generated from different types of tokens may have different impacts on the performance of the change-level defect prediction. To extensively explore the performance of different types of tokens, we build and evaluate change-level defect prediction models with seven different combinations among the three different types of tokens, i.e., **added**: only considers the additions; **deleted**: only considers the deletions; **context**: only considers the context information; **added+deleted**: considers both the additions and the deletions; **added+context**: considers both the additions and the context tokens; **deleted+context**: considers both the deletions and the context tokens; and **added+deleted+context**: considers the additions, deletions, and context tokens together. We discuss the effectiveness of these different combinations in Section 3.5.

Note that some of the tokens extracted from the changed code snippets are project-specific, which means that they are rare or never appear in changes from a different project. Thus, for change-level cross-project defect prediction we first filter out variable names, and then use `method declaration`, `method invocation`, and `class instantiation` to represent a method declaration, a method call, and an instance of a class instantiation respectively.

### 3.3.2 Handling Noise and Mapping Tokens

#### Handling Noise

Defect data are often noisy and suffer from the mislabeling problem. Studies have shown that such noises could significantly erode the performance of defect prediction [87, 123, 261]. To prune noisy data, Kim et al. proposed an effective mislabeling data detection approach named *Closest List Noise Identification* (CLNI) [123]. It identifies the k-nearest neighbors for each instance and examines the labels of its neighbors. If a certain number of neighbors have opposite labels, the examined instance will be flagged as noise. However, such an approach cannot be directly applied to our data because their approach is based on the Euclidean Distance of traditional numerical features. Since our features are semantic tokens, the difference between the values of two features only indicates that these two features are of different tokens.

To detect and eliminate mislabeling data and to help DBN learn the common knowledge between the semantic information of buggy and clean instances, we adopt the edit distance similarity computation algorithm [195] to define the distances between instances. The edit distances are sensitive to both the tokens and the order among the tokens. Given two token sequences $A$ and $B$, the edit distance $d(A, B)$ is the minimum-weight series of edit operations that transform $A$ to $B$. The smaller $d(A, B)$ is, the more similar $A$ and $B$ are.

Based on edit distance similarity, we deploy CLNI to eliminate data with potential incorrect labels. In this study, since our purpose is not to find the best training or test set, we do not spend too much effort on well tuning the parameters of CLNI. We use the recommended parameters and find them to work well. In our benchmark experiments with traditional features, we also perform CLNI to remove the incorrectly labeled data.

In addition, we also filter out infrequent tokens extracted from the source code, which might be designed for a specific file and cannot be generalized to other files. Given a project, if the total number of occurrences of a token is less than three, we filter it out. We encode only the tokens that occur three or more times, which is a common practice in the NLP research field [158]. The same filtering process is also applied to change-level prediction tasks.

#### Mapping Tokens

DBN takes only numerical vectors as inputs, and the lengths of the input vectors must be the same. To use the DBN to generate semantic features, we first build a mapping between integers and tokens, and encode token vectors to integer vectors. Each token has a unique integer identifier. Since our integer vectors may have different lengths, we append 0 to the integer vectors to make all the lengths consistent and equal to the length of the longest vector. Adding zeros does not affect the results, and it is simply a representation transformation to make the vectors acceptable by the DBN. Taking the code snippets in Figure 3.2 as an example, if we only consider the two versions, the token vectors for the "Buggy" and "Clean" versions would be mapped to [1, 2, 3, 4, 5, 6, ...] and [1, 3, 5, 6, 2, 4, ...] respectively. Through this encoding process, the method invocation information and inter-class information are represented as integer vectors. In addition, some program structure information is preserved since the order of tokens remains unchanged. Note that, in this work we employ the same token mapping mechanism for both the file-level and change-level defect prediction tasks.

### 3.3.3 Training the DBN and Generating Features

#### Training the DBN

As we discussed in Section 3.2, to train an effective DBN for learning semantic features, we need to tune three parameters, which are: 1) *the number of hidden layers*, 2) *the number of nodes in each hidden layer*, and 3) *the number of training iterations*. Existing

studies that leveraged DBN models to generate features for NLP [238, 239] and image recognition [48, 129] reported that the performance of DBN-based features is sensitive to these parameters. A few hidden layers can be trained in a relatively short period of time, but result in poor performance as the system cannot fully capture the characteristics of the training datasets. Too many layers may result in overfitting and a slow learning time. Similar to the number of hidden layers, too few or too many hidden nodes or iterations result in either slow learning or poor performance [239]. We show how we tune these parameters in Section 3.4.5.

To simplify our model, we set the number of nodes to be the same in each layer. Through these hidden layers and nodes, DBN obtains characteristics that are difficult to observe but are capable of capturing semantic differences. For each node, the DBN learns the probabilities of traversing from this node to the nodes of its top level. Through back-propagation validation, the DBN reconstructs the input data using generated features by adjusting the weights between nodes in different hidden layers.

The DBN requires the values of the input data to range from 0 to 1, while the data in our input vectors can have any integer values due to our mapping approach. To satisfy the input range requirement, we normalize the values in the data vectors of the training and test sets by using min-max normalization [293]. In our mapping process, the integer values for different tokens are just identifiers. One token with a mapping value of 1 and one token with a mapping value of 2 only means that these two nodes are different and independent. Thus, the normalized values can still be used as token identifiers since the same identifiers pertain the same normalized values.

#### Generating Features

After we train a DBN, both the weights $w$ and the biases $b$ (details are in Section 3.2) are fixed. We input the normalized integer vectors of the training data and the test data into the DBN, and then obtain semantic features for the training and the test data from the output layer of the DBN. Note that the test data (including features and labels) are only used in the evaluation process not in the training process.

### 3.3.4 Building Models and Performing Defect Prediction

After we obtain the generated semantic features for each instance from both the training and the test datasets, we then build defect prediction models by following the standard

defect prediction process described in Section 3.2. The test data are used to evaluate the performance of the built defect prediction models.

Note that, as revealed in existing work [256, 260], the widely used validation technique, i.e., $k$-fold cross-validation often introduces nontrivial bias for evaluating defect prediction models, which makes the evaluation inaccurate. In addition, for change-level defect prediction, the $k$-fold cross-validation may make the evaluation incorrect. This is because the changes follow a certain order in time. Randomly partitioning the dataset into $k$ folds may cause a model to use future knowledge which should not be known at the time of prediction to predict changes in the past. Thus, cross-validation may use information regarding a change committed in 2017 to predict whether a change committed in 2015 is buggy or clean. This scenario would not be a real case in practice, because at the time of prediction, which is typically soon after the change is committed in 2015 for the earlier detection of bugs, the change committed in 2017 is not yet existent. A detailed discussion is provided in Section 3.6.1.

To avoid the above validation problem, we do not use the $k$-fold cross-validation in this work. Specifically, for file-level defect prediction, we evaluate the performance of our DBN-based features and traditional features by building prediction models with data from different releases. For change-level defect prediction, we collect the training and test datasets following the time order (details are in Section 3.4.3) to build and evaluate the prediction models without $k$-fold cross-validation.

## 3.4 Experimental Study

In this section, we describe the detailed settings for our evaluation experiments. All experiments are run on a 2.5GHz i5-3210M machine with 4GB RAM.

### 3.4.1 Research Questions

Table 3.2: Research questions investigated for defect prediction.

| | | Scope | |
|---|---|---|---|
| | | Within-project | Cross-project |
| Level | File | **RQ1** | **RQ2** |
| | Change | **RQ3** | **RQ4** |

Table 3.2 lists the scenarios for the investigated research questions. Specifically, we evaluate the performance of our DBN-based semantic features by comparing it with traditional defect prediction features under each of the four different prediction scenarios. These questions share the following format.

**RQi** ($1 \leq i \leq 4$): Do DBN-based semantic features outperform traditional features at the <**level**> <**scope**> under the non-effort-aware and effort-aware evaluation scenarios?

For example, in **RQ1**, we explore the effectiveness of the DBN-based semantic features for within-project defect prediction at the file-level under both the non-effort-aware and effort-aware evaluation scenarios.

### 3.4.2 Evaluation Metrics

#### Metrics for Non-effort-aware Evaluation

Under the non-effort-aware scenario, we use three metrics: *Precision*, *Recall*, and $F1$. These metrics have been widely adopted to evaluate defect prediction techniques [109, 169, 170, 194, 256, 326]. Here is a brief introduction:

$$Precision = \frac{true\ positive}{true\ positive + false\ positive} \tag{3.6}$$

$$Recall = \frac{true\ positive}{true\ positive + false\ negative} \tag{3.7}$$

$$F1 = \frac{2 * Precision * Recall}{Precision + Recall} \tag{3.8}$$

Precision and recall are composed of three numbers in terms of *true positive*, *false positive*, and *false negative*. True positive is the number of predicted defective files (or changes) that are truly defective, while false positive is the number of predicted defective ones that are actually not defective. A false negative records the number of predicted non-defective files (or changes) that are actually defective. Higher precision is demanded by developers who do not want to waste their debugging efforts on the non-defective code, while higher recall is often required for mission-critical systems, e.g., revealing additional defects [326]. However, comparing defect prediction models by using only these two metrics may be incomplete. For example, one could simply predict all instances as buggy instances to achieve a recall score of 1.0 (which will likely result in a low precision score) or only classify the instances with higher confidence values as buggy instances to achieve a higher precision score (which

could result in a low recall score). To overcome the above issues, we also use the F1 score (i.e., F1), which is the harmonic mean of precision and recall, to measure the performance of the defect prediction.

### Metrics for Effort-aware Evaluation

For effort-aware evaluation, we employ `PofB20` [104] to measure the percentage of bugs that a developer can identify by inspecting the top 20 percent lines of code.

To calculate `PofB20`, we first sort all the instances in the test dataset based on the confidence levels (i.e., probabilities of being predicted as buggy) that a defect prediction model generates for each instance. This is because an instance with a higher confidence level is more likely to be buggy. We then simulate a developer that inspects these potentially buggy instances. We accumulate the lines of code (LOC) that are inspected and the number of bugs identified. The process will be terminated when 20 percent of the LOC in the test data have been inspected and the percentage of bugs that are identified is referred to as the `PofB20` score. A higher `PofB20` score indicates that a developer can detect more bugs when inspecting a limited number of LOC.

### Statistical Tests

Statistical tests can help understand whether there is a statistically significant difference between two results. In this work, we used the Wilcoxon signed-rank test to check whether the performance difference between prediction models with DBN-based semantic features and prediction models with traditional features is significant. For example, in RQ3, we want to compare the performance of DBN-based features and traditional features for change-level within-project defect prediction for the projects listed in Table 3.5. To conduct the Wilcoxon signed-rank test, we first run experiments with these two sets of features and obtain prediction results for each test subject. We then apply the Wilcoxon signed-rank test on the results of the test subjects. The Wilcoxon signed-rank test does not require the underlying data to follow any distribution. In addition, it can be applied to pairs of data and is able to compare the difference against zero. At the 95% confidence level, $p$-values that are less than 0.05 indicate that the difference between subjects is statistically significant, while $p$-values that are 0.05 or larger indicate that the difference is not statistically significant.

Table 3.3: Cliff's Delta and the effectiveness level [49].

| Cliff's Delta ($\delta$) | Effectiveness Level |
|---|---|
| $\|\delta\| < 0.147$ | Negligible |
| $0.147 \leq \|\delta\| < 0.33$ | Small |
| $0.33 \leq \|\delta\| < 0.474$ | Medium |
| $\|\delta\| \geq 0.474$ | Large |

**Cliff's Delta Effect Size Analysis**

To further examine the effectiveness of our DBN-based features, following the existing work in [191, 299], we employ Cliff's *delta* ($\delta$) [49] to measure the effect size of our approach. Cliff's *delta* is a non-parametric effect size measure that quantifies the amount of difference between two approaches. In this work, we use Cliff's *delta* to compare the defect prediction models that are built with our DBN-based features to the defect prediction models that are built with traditional features. Cliff's *delta* is computed using the formula $delta = (2W/mn) - 1$, where $W$ is the $W$ statistic of the Wilcoxon rank-sum test, and $m$ and $n$ are the sizes of the result distributions of two compared approaches. The *delta* values range from -1 to 1, where $\delta = -1$ or 1 indicates the absence of an overlap between the performances of the two compared models (i.e., all F1 values from one prediction model are higher than the F1 values of the other prediction model, and vice versa), while $\delta = 0$ indicates that the two prediction models completely overlap. Table 3.3 describes the meanings of the different Cliff's delta values [49].

### 3.4.3 Evaluated Projects and Data Sets

In this work, we use different datasets for evaluating file-level and change-level defect prediction tasks. Specifically, for evaluating the performance of DBN-based features on file-level defect prediction, we use publicly available data from the PROMISE data repository, which are widely used for evaluating file-level defect prediction models [84,109,193,194,299]. For change-level defect prediction, we adopt the dataset from previous studies [104, 256, 300].

The main reason for adopting different datasets for file-level and change-level defect prediction tasks is that using existing widely used datasets enables us to directly compare our approach with existing defect prediction models on the same datasets, which makes the comparison more reliable.

Table 3.4: Evaluated projects for file-level defect prediction. **BR** is the average buggy rate.

| Project | Description | Releases | Avg # Files | BR (%) |
|---|---|---|---|---|
| ant | Java based build tool | 1.5,1.6,1.7 | 463.7 | 21.0 |
| camel | Enterprise integration framework | 1.2,1.4,1.6 | 815 | 22.5 |
| jEdit | Text editor designed for programmers | 3.2,4.0,4.1 | 297 | 27.4 |
| log4j | Logging library for Java | 1.0,1.1 | 122 | 29.1 |
| lucene | Text search engine library | 2.0,2.2,2.4 | 260.7 | 56.0 |
| xalan | A library for transforming XML files | 2.4,2.5 | 763 | 32.6 |
| xerces | XML parser | 1.2,1.3 | 446.5 | 15.7 |
| ivy | Dependency management library | 1.4,2.0 | 296.5 | 9.4 |
| synapse | Data transport adapters | 1.0,1.1,1.2 | 211.7 | 25.5 |
| poi | Java library to access Microsoft format files | 1.5,2.5,3.0 | 354.7 | 62.9 |

Table 3.5: Evaluated projects for change-level defect prediction. **Lang** is the programming language used for the project. **LOC** is the number of the line of code. **First Date** is the date of the first commit of a project, while **Last Date** is the date of the latest commit. **Changes** is the number of changes. **TrSize** is the average size of training data on all runs. **TSize** is the average size of test data on all runs. **NR** is the number of runs for each subject. **Ch** is the number of changes. **BR** is the average buggy rate.

| Project | Lang | LOC | First Date | Last Date | # Ch | TrSize | TSize | BR (%) | # NR |
|---|---|---|---|---|---|---|---|---|---|
| Linux | C | 7.3M | 2005-04-16 | 2010-11-21 | 429K | 1,608 | 6,864 | 22.8 | 4 |
| PostgreSQL | C | 289K | 1996-07-09 | 2011-01-25 | 89K | 1,232 | 6,824 | 27.4 | 7 |
| Xorg | C | 1.1M | 1999-11-19 | 2012-06-28 | 46K | 1,756 | 6,710 | 14.7 | 6 |
| JDT | Java | 1.5M | 2001-06-05 | 2012-07-24 | 73K | 1,367 | 6,974 | 20.5 | 6 |
| Lucene | Java | 828K | 2010-03-17 | 2013-01-16 | 76K | 1,194 | 9,333 | 23.6 | 8 |
| Jackrabbit | Java | 589K | 2004-09-13 | 2013-01-14 | 61K | 1,118 | 8,887 | 37.4 | 10 |

### Evaluated Projects for File-level Defect Prediction

To facilitate the replication and verification of our experiments, we use publicly available data from the PROMISE data repository. Specifically, we select all the Java projects from PROMISE[2] whose version numbers are provided. We need the version numbers of each project because we need its source code archive to extract token vectors from the ASTs of the source code to feed our DBN-based feature generation approach. In total, 10 Java projects are collected. Table 3.4 lists the versions, the average number of source files

[2]http://openscience.us/repo/defect

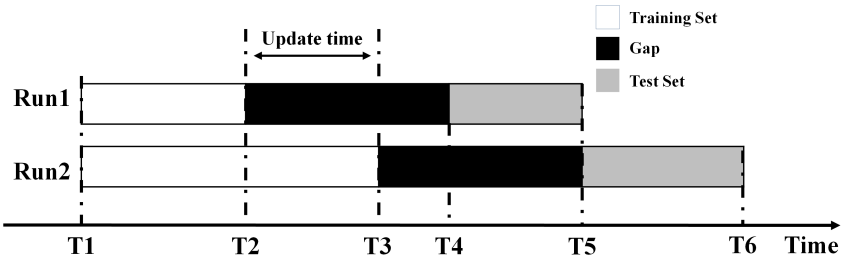



Figure 3.6: Change-level data collection process [256].

(excluding test files), and the average buggy rate of each project. The average number of files of the projects ranges from 122 to 815, and the buggy rates of the projects have a minimum value of 9.4% and a maximum value of 62.9%.

### Evaluated Projects for Change-level Defect Prediction

We choose six open-source projects: Linux kernel, PostgreSQL, Xorg, Jdt (from Eclipse), Lucene, and Jackrabbit. They are large and typical open-source projects covering operating systems, database management systems. These projects have sufficient change histories to build and evaluate change-level defect prediction models and are commonly used in the literature [104, 256, 300]. For Lucene and Jackrabbit, we use manually verified bug reports from Herzig et al. [87] to label the bug-fixing changes, and the keyword search approach [245] is used for the others.

Table 3.5 shows the evaluated projects for change-level defect prediction. The LOC and the number of changes in Table 3.5 include only source code (C and Java) files[3] and their changes because we want to focus on classifying source code changes only. Although these projects are written in C and Java, our DBN-based feature generation approach is not limited to any particular programming language. With the appropriate feature extraction approach, our DBN-based feature generation approach can easily be extended to projects in other languages.

Change-level defect data are often imbalanced [83, 104, 114, 115], i.e., there are fewer buggy instances than clean instances in the training dataset. For example, as shown in Table 3.5, the average ratio of the buggy and the clean changes is 1.0 to 3.1. The imbalanced data can lead to poor prediction performance [256]. For change-level data, we borrow the

[3]We include files with these extensions: .java, .c, .cpp, .cc, .cp, .cxx, .c++, .h, .hpp, .hh, .hp, .hxx and .h++.

data collection process introduced by Tan et al. [256]. Specifically, a gap between the training set and the test set (see Figure 3.6) is used because the gap allows more time for buggy changes in the training set to be discovered and fixed. For example, the time period between time T2 and time T4 is a gap. In this manner, the training set will be more balanced, i.e., the training set will have a higher buggy rate. A reasonable setup is to make the sum of the gap and the test set, e.g., the duration from time T2 to T5, close to the typical bug-fixing time (i.e., the time from when a bug is introduced until it is fixed). We use the recommended gap values in [256] to collect multiple runs of experimental data, e.g., Linux has four different runs during the given time period (between the `First Date` and `Last Date`) as shown in Table 3.5. Note that our previous study [256] tuned and evaluated the defect prediction models based on their precision values. In this work, we do not have a bias on either precision or recall, and we tune and evaluate the prediction models based on the harmonic of the precision and recall, i.e., F1 (details are in Section 3.4.2).

Imbalanced data issues occur in both the file-level and the change-level defect data, and as shown in Table 3.4 and Table 3.5, most of the examined projects have buggy rates less than 50%. To build optimal defect prediction models, we also perform the re-sampling technique used in existing work [256], i.e., SMOTE [40], on the imbalanced projects.

### 3.4.4 Baselines of Traditional Features

#### Baselines for Evaluating File-level Defect Prediction

To evaluate the performance of semantic features for file-level defect prediction tasks, we compare the semantic features with two different traditional features. Our **first baseline** of traditional features consists of 20 traditional features. Table 3.6 shows the details of the 20 features and their descriptions. These features and data have been widely used in previous work to build effective defect prediction models [84, 109, 169, 170, 194, 326].

We choose the widely used PROMISE data so that we can directly compare our approach with previous studies. For a fair comparison, we also perform the noise removal approach described in Section 3.3.2 on the PROMISE data.

The traditional features from PROMISE do not contain AST nodes, which were used as the input by our DBN models. For a fair comparison, our **second baseline** of traditional features is the AST nodes that were given to our DBN models, i.e., the AST nodes in all files after handling the noise (Section 3.3.2). Each instance is represented as a vector of term frequencies of the AST nodes.

Table 3.6: Benchmark metrics used for file-level defect prediction.

| Metric | Description |
|---|---|
| WMC | the number of methods used in a given class [46] |
| DIT | the maximum distance from a given class to the root of an inheritance tree [46] |
| NOC | the number of children of a given class in an inheritance tree [46] |
| CBO | the number of classes that are coupled to a given class [46] |
| RFC | the number of distinct methods invoked by code in a given class [46] |
| LCOM | the number of method pairs in a class that do not share access to any class attributes [46] |
| LCOM3 | another type of lcom metric proposed by Henderson-Sellers [57] |
| NPM | the number of public methods in a given class [22] |
| LOC | the number of lines of code in a given class [22] |
| DAM | the ratio of the number of private/protected attributes to the total number of attributes in a given class [22] |
| MOA | the number of attributes in a given class which are of user-defined types [22] |
| MFA | the number of methods inherited by a given class divided by the total number of methods that can be accessed by the member methods of the given class [22] |
| CAM | summation of number of different types of method parameters in every method divided by a multiplication of number of different method parameter types in whole class and number of methods [22] |
| IC | the number of parent classes that a given class is coupled to [111] |
| CBM | the total number of new or overwritten methods that all inherited methods in a given class are coupled to [111] |
| AMC | the average size of methods in a given class [111] |
| CA | afferent coupling, which measures the number of classes that depends upon a given class [161] |
| CE | efferent coupling, which measures the number of classes that a given class depends upon [161] |
| Max_CC | the maximum McCabe's cyclomatic complexity (CC) score [163] of methods in a given class |
| Avg_CC | the arithmetic mean of the McCabe's clomatic complexity (CC) scores [163] of methods in a given class |

### Baselines for Evaluating Change-level Defect Prediction

Our baseline features for change-level defect prediction include three types of change features, i.e., `bag-of-words features`, `characteristic features`, and `meta features`, which have been used in previous studies [104, 256].

- `Bag-of-words features`: The bag-of-words feature set is a vector representing the count of occurrences of each word in the text of changes. We employ the snowBall

stemmer to group words of the same root, then we use Weka [74] to obtain the bag-of-words features from both the commit messages and the source code changes.

- `Characteristic features:` Inspired by the Deckard tool [103], we use characteristic vectors as features. Characteristic vectors represent the syntactic structure by counting the numbers of each node type in the Abstract Syntax Tree (AST). Bag-of-words and characteristic vectors have different abstraction levels. Although bag-of-words can capture keywords, such as `if` and `while`, it cannot capture abstract syntactic structures, such as the number of statements. Suppose that we are using `if` and `else` node types for characteristic vectors, the characteristic vector of the code before the changes shown in Figure 3.5 is (1, 1). After obtaining the characteristic vectors for the file before the change and the file after the change, we subtract the two characteristic vectors to obtain the difference. For each change, we use Deckard [103] to automatically generate two characteristic vectors: one for the source code file before the change and one for the source code file after the change. We use the difference between the two characteristic vectors and the characteristic vector of the file after the change as two sets of features.

- `Meta features:` In addition to characteristic and bag-of-words vectors, we also use a set of metadata features, which includes the basic information of changes, e.g., commit time, filename, developers, etc. It also contains code change metrics, e.g., the added line count per change, the deleted line count per change, etc.

### 3.4.5 Parameter Settings for Training a DBN

Many DBN applications [48, 129, 180] report that an effective DBN requires well-tuned parameters, i.e., 1) *the number of hidden layers*, 2) *the number of nodes in each hidden layer*, and 3) *the number of iterations*. In this section, we study the impact of the three parameters on defect prediction models.

#### Setting Parameters for File-level Defect Prediction

For file-level defect prediction, we tune the three parameters by conducting experiments with different values of the parameters on `ant` (1.5, 1.6), `camel` (1.2, 1.4), `jEdit` (4.0, 4.1), `lucene` (2.0, 2.2), and `poi` (1.5, 2.5). Each experiment has specific values for the three parameters and runs on the five projects individually. Given an experiment, for each project, we use the older version of the project to train a DBN with respect to the

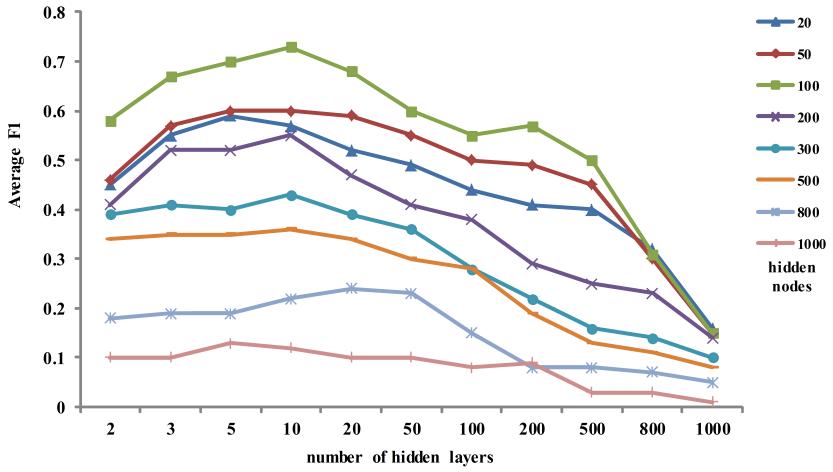



Figure 3.7: File-level defect prediction performance with different parameters.

specific values of the three parameters. Then, we use the trained DBN to generate semantic features for both the older and newer versions of the project. After this, we use the older version to build a defect prediction model and apply it to the newer version. Finally, we evaluate the specific values of the parameters by the average F1 score of the five projects for file-level defect prediction.

**Setting the number of hidden layers and the number of nodes in each layer.** Because the number of hidden layers and the number of nodes in each hidden layer interact with each other, we tune these two parameters together. For the number of hidden layers, we experiment with 11 discrete values that include 2, 3, 5, 10, 20, 50, 100, 200, 500, 800, and 1,000. For the number of nodes in each hidden layer, we experiment with eight discrete values i.e., 20, 50, 100, 200, 300, 500, 800, and 1,000. When we evaluate these two parameters, we set the number of iterations to 50 and keep it constant.

Figure 3.7 illustrates the average F1 scores obtained when tuning the number of hidden layers and the number of nodes in each hidden layer together for file-level defect prediction. When the number of nodes in each layer is fixed while increasing the number of hidden layers, all the average F1 scores are convex curves. Most curves peak at the point where the number of hidden layers is 10. If the number of hidden layers remains unchanged, the best F1 score occurs when the number of nodes in each layer is 100 (the top line in Figure 3.7). As a result, we choose the number of hidden layers as 10 and the number of nodes in each hidden layer as 100. Thus, the number of the DBN-based features for file-level defect prediction tasks is 100.

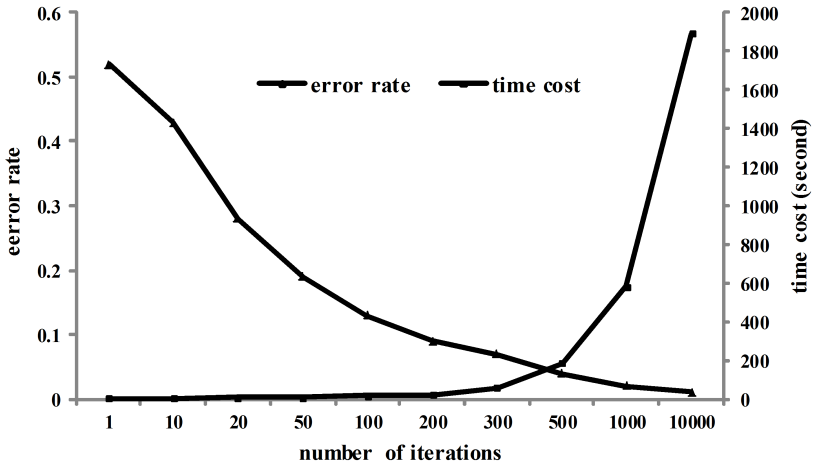



Figure 3.8: Average error rates and time costs for different numbers of iterations for tuning file-level defect prediction.

**Setting the number of iterations.** The number of iterations is another important parameter for building an effective DBN. During the training process, the DBN adjusts the weights to narrow down the *error rate* between the reconstructed input data and original input data in each iteration. In general, the higher the number of iterations, the lower the error rate. However, there is a trade-off between the number of iterations and the computational time cost. For tuning the parameters for file-level defect prediction, we choose the same five projects to conduct experiments with ten discrete values for the number of iterations. The values range from 1 to 10,000. We use the error rate to evaluate this parameter. Figure 3.8 demonstrates that, as the number of iterations increases, the error rate decreases slowly as the corresponding time cost increases exponentially. In this study, we set the number of iterations to 200, with which the average error rate is approximately 0.098 and the time cost is 15 seconds.

### Setting Parameters for Change-level Defect Prediction

For change-level defect prediction, we use the same parameter tuning process as the file-level defect prediction to explore the best parameter values with all the runs of each of the six projects listed in Table 3.5. For each run of a project, we use its training data to train a DBN with respect to the specific values of the DBN parameters. Then, we use the trained DBN to generate semantic features for both the training and test datasets. Afterward, we use the training dataset to build a defect prediction model and apply it to the test dataset.

Table 3.7: The comparison of F1 scores among change-level defect prediction with different DBN-based features generated by the seven different types of tokens. The F1 scores are measured as a percentage. The best F1 values are highlighted in bold.

| **Project** | **added** | **deleted** | **context** | **added+deleted** | **added+context** | **deleted+context** | **added+deleted+context** |
|---|---|---|---|---|---|---|---|
| Linux | 39.2 | 39.8 | 32.5 | 39.8 | 40.1 | 40.6 | **41.3** |
| PostgreSQL | 48.9 | 49.5 | 39.6 | 49.8 | 51.8 | 50.1 | **55.0** |
| Xorg | 41.1 | 38.4 | 30.2 | 40.7 | 41.3 | 41.2 | **41.4** |
| JDT | 39.5 | 30.5 | 18.7 | 40.1 | 39.6 | 33.3 | **41.4** |
| Lucene | 37.2 | 38.1 | 31.4 | 37.8 | 38.9 | 38.5 | **39.7** |
| Jackrabbit | 45.3 | 44.7 | 39.5 | 45.6 | 46.6 | 47.8 | **49.9** |
| **Average** | 41.9 | 40.2 | 32.0 | 42.3 | 43.1 | 41.9 | **44.8** |

Last, we evaluate the specific values of the parameters by using the average F1 score of the 41 runs from the six projects.

Note that, for change-level defect prediction, as we described in Section 3.3.1, we have seven different approaches available to extract the source code token vector for a source code change. Our tuning process considers these different types of tokens, the number of hidden layers, and the number of nodes in each layer together. Specifically, for each type of tokens we input them into our DBN model to generate features with different configurations. Similar to our tuning process of file-level defect prediction, for the number of hidden layers, we experiment with 11 discrete values, i.e., 2, 3, 5, 10, 20, 50, 100, 200, 500, 800, and 1,000. For the number of nodes in each hidden layer, we experiment with eight discrete values, i.e., 20, 50, 100, 200, 300, 500, 800, and 1,000. When we evaluate the seven different types of tokens and the two parameters, we set the number of iterations to 50 and keep it constant.

Table 3.7 shows the F1 scores of the change-level defect prediction with DBN-based semantic features generated by each of the seven types of tokens. Note that among the three basic token types (i.e., `added`, `deleted`, and `context`), the DBN-based features generated by `added` and `deleted` deliver better performance than `context` on all six projects. The improvement could be up to 20.8 percentage points (on project Jdt) and on average the improvement is larger than 8 percentage points. In addition, all the four different combinations, i.e., `added+deleted`, `added+context`, `deleted+context`, and `added+deleted+context`, can generate better performance than the corresponding three basic token types. This may be because the combinations provide more information to the DBN model for generating more effective features to capture buggy changes (a detailed discussion is provided in Section 3.6.3). Among the four combinations, `added+deleted+context` achieves the best performance.

In this work, we use the combination of `added`, `deleted`, and `context` tokens as input to DBN models to generate features. The corresponding best value of the number of hidden layers is 5 and the best value of the number of nodes in each hidden layers is 50. This means that the number of generated DBN-based features for change-level defect prediction is 50. Additionally, for change-level defect prediction, we also set the number of iterations to 200, with which the average error rate is less than 0.05 and the time cost for feature generation is less than 5 seconds.

### 3.4.6 File-level Within-Project Defect Prediction

To examine the performance of our semantic features on file-level within-project defect prediction, we build defect prediction models using three machine learning classifiers, i.e., ADTree, Naive Bayes, and Logistic Regression, which have been widely explored in previous work [109, 169, 170, 194, 326]. We use two consecutive versions of each project listed in Table 3.4 as the training and test data sets. We use the source code of an older version to train the DBN and generate the training feature set. Then we use the trained DBN to generate features for instances from a newer version. We compare our semantic features with the traditional features as described in Section 3.4.4. For a fair comparison, we use the same classifiers on these traditional features.

### 3.4.7 File-level Cross-Project Defect Prediction

Due to a lack of defect data, it is often difficult to build accurate prediction models for new projects. To overcome this problem, cross-project defect prediction techniques train prediction models using data from mature projects (called *source projects*), and use the trained models to predict defects for new projects (called *target projects*). However, because the features of source projects and target projects often have different distributions, making an accurate and precise cross-project defect prediction model is still challenging [193].

We believe that the semantic features can capture the common characteristics of defects, which implies that the semantic features trained from one project can be used to predict defects in a different project, and so is applicable in cross-project defect prediction. To measure the performance of the semantic features in cross-project defect prediction, we propose a technique called **DBN C**ross-Project Defect **P**rediction (`DBN-CP`). Given a source project and a target project, DBN-CP first trains a DBN by using the source project and generates semantic features for both projects. Then, DBN-CP trains an ADTree based

defect prediction model using data from the source project and uses the built model to perform defect prediction on the target project.

We choose TCA+ [194] as our baseline. To compare with TCA+, we design two different experiments. First, for each of the 16 test versions (which are the target versions in cross-project prediction) from the within-project experiments list in Table 3.8, we randomly select two source projects that are different from the target projects. Thus, 32 test pairs are collected. Our first experiment can help evaluate the performance of DBN-CP compared to TCA+ and the corresponding within-project defect prediction. Then, to extensively examine the performance of DBN-CP, we use each version from one project as a target project and each version from the other projects as a source project. In total, 606 test pairs are formed.

The reason why we use TCA+ for the comparison that TCA+ is one of the state-of-the-art techniques in cross-project defect prediction [194]. In our reproduction, we follow the processes described in [194]. We first implement all five of their proposed normalization methods and assign them the same conditions as given in the TCA+ paper. We then perform *Transfer Component Analysis* [208] on the source projects and the target projects together, and map them onto the same subspace while minimizing the data difference and maximizing the data variance. Finally, we use the source projects and target projects with the new features to build and evaluate the ADTree-based prediction models.

### 3.4.8 Change-level Within-Project Defect Prediction

To examine the effectiveness of the learned DBN-based features for change-level defect prediction tasks, we compare the performance of the DBN-based features to the three types of traditional features described in Section 3.4.4. By examining the combination of these traditional features, we should be able to generate the best performance for change-level defect prediction [104, 256]. In this work, we use the combination as the benchmark for change-level defect prediction.

To generate DBN-based semantic features, for each run of a project listed in Table 3.5, we use its training data to train a DBN (with the combination of all the tokens in a change as the input to the DBN). Then, we use the trained DBN to generate semantic features for both the training and test datasets. We then use the training data to build a defect prediction model and apply it to the test data. For the classification algorithm, we use ADTree in Weka [74] as the classifier, because it has delivered the best performance in previous work [104, 194, 256].

### 3.4.9 Change-level Cross-Project Defect Prediction

Similar to file-level defect models, change-level models also require a large amount of training data to train and build prediction models. However, sufficient training data are not often available when projects are in their initial development phases. To address this limitation, cross-project models for change-level prediction tasks are needed [114]. To explore the performance of the DBN-based semantic features in change-level cross-project defect prediction, we propose a technique called **DBN C**hange-level **C**ross-Project defect **P**rediction (`DBN-CCP`). Specifically, given a source project and a target project, DBN-CCP first trains a DBN by using the source project and generates semantic features for both the source project and the target project. Then, DBN-CCP trains a defect prediction model using data from the source project, and uses the built model to perform defect prediction on the target project.

For evaluating the performance of DBN-CCP, we also choose TCA+ [194] as our baseline. Note that TCA+ requires that the target and source projects have the same features for learning TCA+ based features. As described in Section 3.4.4, in this study we leverage three different types of features for change-level defect prediction, i.e., bag-of-words features, characteristic features, and meta features. Both the bag-of-words features and characteristic features are project-specific and vary for different projects. Thus, for TCA+ on change-level prediction, we only use the meta features.

To extensively evaluate the performance of DBN-CCP, we use each test dataset in all runs from one project as a target dataset and each training dataset in all runs from the other projects as a source dataset to form change-level cross-project test pairs. For example, one test pair could be a training set from Run 1 of Project A and a test set from Run 1 of Project B, a training set from Run 2 of Project A and a test set from Run 1 of Project B, etc. In total, 1,380 test pairs are formed.

## 3.5 Results and Analysis

### 3.5.1 RQ1: Performance of semantic features for file-level within-project defect prediction

#### Non-effort-aware evaluation scenario

We build file-level within-project defect prediction models to compare the impact of three sets of features: semantic features that are automatically learned by DBN, PROMISE

features, and AST features. The latter two are the baselines of traditional features. We conduct 16 sets of file-level within-project defect prediction experiments, each of which uses two versions from the same project (listed in Table 3.4). The older version is used to train the prediction models, and the newer version is used as the test set to evaluate the trained models.

Table 3.8 shows the performance of the file-level within-project defect prediction experiments. The highest F1 values of the three sets of features are shown in bold. For example, by using `ant 1.6` as the training set, and `ant 1.7` as the test set, the F1 of using semantic features is 94.2%, while the F1 is only 54.2% with the first baseline of traditional features (from PROMISE), and the F1 is 47.0% with the second baseline of traditional features (AST nodes). For this comparison, the only difference is the three sets of features, meaning that the same classification algorithm, namely ADTree and the same training and test sets are used.

The results demonstrate that by using the DBN-based semantic features instead of the PROMISE features, we can improve the F1 by 14.2 percentage points on the 16 experiment pairs on average. The average improvements in the precision and recall are 14.7 percentage points and 11.5 percentage points respectively.

Since the DBN algorithm has randomness, the generated features vary between different runs. Therefore, we run our DBN-based feature generation approach five times for each experiment. Among the runs, the difference in the generated features is at the level of 1.0E-20, which is too small to propagate to precision, recall, and F1. In other words, the precision, recall, and F1 of all five runs are identical.

### Effort-aware evaluation scenario

For the effort-aware scenario, we rerun the 16 pairs of file-level within-project defect prediction experiments listed in Table 3.8, and calculate the `PofB20` of the test data in each experiment based on our setup description in Section 3.4.2.

Table 3.9 presents the `PofB20` of file-level within-project defect prediction models with DBN-based semantic features and the PROMISE features. As we can see, in all the experiments, DBN-based features could achieve better `PofB20` than the corresponding PROMISE features. Compared to the PROMISE features, the improvement could be as much as 26.7 percentage points (ant 1.6 $\Rightarrow$ ant 1.7) and is, on average, 13.3 percentage points.

Table 3.8: Comparison between semantic features and two baselines of traditional features (PROMISE features and AST features) using ADTree. Tr denotes the training set version and T denotes the test set version. P, R, and F1 denote the precision, recall, and F1 score respectively and are measured as a percentage. The better F1 values with statistical significance ($p$-value $< 0.05$) among the three sets of features are shown with an asterisk (*). The numbers in parentheses are the effect sizes comparative to the **Semantic**. A positive value indicates that the semantic features improve the baseline features in terms of the effect size.

| Project | Versions (Tr ⇒ T) | **Semantic*** | | | PROMISE (0.555) | | | AST (0.656) | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | P | R | F1 | P | R | F1 | P | R | F1 |
| ant | 1.5 ⇒ 1.6 | 88.0 | 95.1 | **91.4** | 44.8 | 51.1 | 47.7 | 40.5 | 51.4 | 45.3 |
| | 1.6 ⇒ 1.7 | 98.8 | 90.1 | **94.2** | 41.8 | 77.1 | 54.2 | 41.2 | 54.7 | 47.0 |
| camel | 1.2 ⇒ 1.4 | 96.0 | 66.4 | **78.5** | 24.8 | 75.2 | 37.3 | 32.3 | 55.6 | 40.2 |
| | 1.4 ⇒ 1.6 | 26.3 | 64.9 | 37.4 | 28.3 | 63.7 | **39.1** | 29.7 | 51.5 | 38.3 |
| jEdit | 3.2 ⇒ 4.0 | 46.7 | 74.7 | **57.4** | 44.7 | 73.3 | 55.6 | 45.8 | 47.4 | 46.6 |
| | 4.0 ⇒ 4.1 | 54.4 | 70.9 | **61.5** | 46.1 | 67.1 | 54.6 | 50.4 | 40.4 | 44.8 |
| log4j | 1.0 ⇒ 1.1 | 67.5 | 73.0 | **70.1** | 49.1 | 73.0 | 58.7 | 55.4 | 38.6 | 45.5 |
| lucene | 2.0 ⇒ 2.2 | 75.9 | 56.9 | **65.1** | 73.3 | 38.2 | 50.2 | 69.5 | 37.4 | 48.4 |
| | 2.2 ⇒ 2.4 | 66.5 | 92.1 | **77.3** | 70.9 | 52.7 | 60.5 | 65.9 | 53.1 | 58.8 |
| xalan | 2.4 ⇒ 2.5 | 65.0 | 54.8 | **59.5** | 64.7 | 43.2 | 51.8 | 60.1 | 43.5 | 50.5 |
| xerces | 1.2 ⇒ 1.3 | 40.3 | 42.0 | **41.1** | 16.0 | 46.4 | 23.8 | 25.5 | 22.0 | 23.6 |
| ivy | 1.4 ⇒ 2.0 | 21.7 | 90.0 | **35.0** | 22.6 | 60.0 | 32.9 | 31.6 | 28.6 | 30.0 |
| synapse | 1.0 ⇒ 1.1 | 46.0 | 66.7 | **54.4** | 45.5 | 50.0 | 47.6 | 51.5 | 45.7 | 48.4 |
| | 1.1 ⇒ 1.2 | 57.3 | 59.3 | **58.3** | 51.1 | 55.8 | 53.3 | 50.7 | 40.5 | 49.0 |
| poi | 1.5 ⇒ 2.5 | 76.1 | 55.2 | **64.0** | 73.7 | 44.8 | 55.8 | 70.0 | 31.6 | 43.5 |
| | 2.5 ⇒ 3.0 | 81.6 | 79.0 | **80.3** | 75.0 | 75.8 | 75.4 | 72.1 | 46.3 | 55.6 |
| **Average** | | 63.0 | 70.7 | **64.1** | 48.3 | 59.2 | 49.9 | 49.5 | 43.0 | 44.7 |

We further conduct the Wilcoxon signed-rank test ($p < 0.05$) to compare the performance of the DBN-based semantic features and PROMISE features for file-level within-project defect prediction with the 16 experiment pairs under both the non-effort-aware and effort-aware evaluation scenarios. The results suggest that the DBN-based semantic features are significantly better than the PROMISE features.

Our DBN-based approach is effective in automatically learning semantic features, which significantly improves the performance of file-level within-project defect prediction under both non-effort-aware and effort-aware evaluation scenarios with large effect sizes.

Table 3.9: `PofB20` scores of DBN-based features and traditional features for WPDP. The `PofB20` scores are measured as a percentage. The best values are in bold. The better `PofB20` values with statistical significance ($p$-value $< 0.05$) between the two sets of features are indicated with an asterisk (*). The numbers in parentheses are the effect sizes comparative to the **Semantic**.

| Project | Versions (Tr⇒T) | **Semantic*** | PROMISE (0.756) |
|---|---|---|---|
| ant | 1.5⇒1.6 | **44.3** | 16.3 |
| | 1.6⇒1.7 | **50.2** | 23.5 |
| camel | 1.2⇒1.4 | 33.2 | **33.8** |
| | 1.4⇒1.6 | **30.1** | 23.4 |
| jEdit | 3.2⇒4.0 | **40.1** | 29.3 |
| | 4.0⇒4.1 | **32.6** | 17.7 |
| log4j | 1.0⇒1.1 | **25.0** | 21.6 |
| lucene | 2.0⇒2.2 | **32.1** | 14.6 |
| | 2.2⇒2.4 | **37.9** | 23.2 |
| xalan | 2.4⇒2.5 | **24.5** | 8.3 |
| xerces | 1.2⇒1.3 | **9.1** | 7.2 |
| ivy | 1.4⇒2.0 | **28.3** | 15.1 |
| synapse | 1.0⇒1.1 | **29.6** | 13.3 |
| | 1.1⇒1.2 | **32.5** | 12.8 |
| poi | 1.5⇒2.5 | **38.7** | 26.2 |
| | 2.5⇒3.0 | **25.5** | 13.9 |
| **Average** | | **32.1** | 18.8 |

**RQ1a: Do semantic features outperform traditional features with other classification algorithms?**

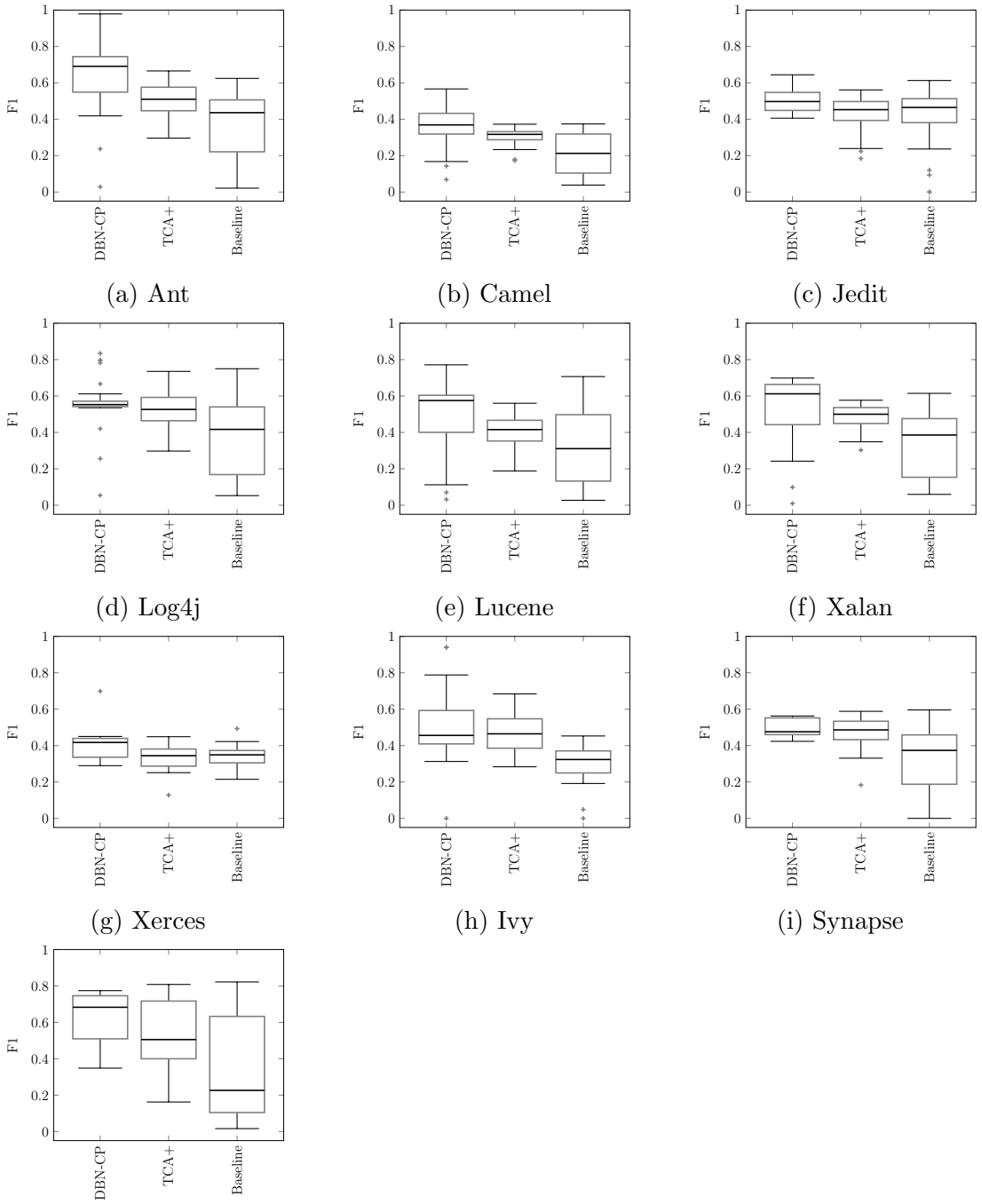
To answer this question, we build file-level within-project defect prediction models by using two alternative classification algorithms, i.e., Naive Bayes and Logistic Regression. We conduct 16 sets of file-level within-project defect prediction tests, where the training sets and the test sets are exactly the same as those in RQ1. Table 3.10 shows the F1 scores of running Naive Bayes and Logistic Regression on semantic features and PROMISE features. Take `ant` as an example, when the model is built on Naive Bayes, by choosing version 1.5 as the training set and 1.6 as the test set, the semantic features produce an F1 of 63.0%, which is 7.0 percentage points higher than using PROMISE features. For the same example with Logistic Regression as the classification algorithm, the semantic features achieve an F1 of 91.6%, while using PROMISE features produces an F1 of 50.6% only. Among the

Table 3.10: Comparison of F1 scores between semantic features and PROMISE features using Naive Bayes and Logistic Regression. Tr denotes the training set version and T denotes the test set version. The F1 scores are measured as a percentage.

| Project | Version (Tr⇒T) | Naive Bayes | | Logistic Regression | |
|---|---|---|---|---|---|
| | | Semantic | PROMISE | Semantic | PROMISE |
| ant | 1.5⇒1.6 | **63.0** | 56.0 | **91.6** | 50.6 |
| | 1.6⇒1.7 | **96.1** | 52.2 | **92.5** | 54.3 |
| camel | 1.2⇒1.4 | **45.9** | 30.7 | **59.8** | 36.3 |
| | 1.4⇒1.6 | **48.1** | 26.5 | 34.2 | **34.6** |
| jEdit | 3.2⇒4.0 | **58.3** | 48.6 | **55.2** | 54.5 |
| | 4.0⇒4.1 | **60.9** | 54.8 | **62.3** | 56.4 |
| log4j | 1.0⇒1.1 | **72.5** | 68.9 | **68.2** | 53.5 |
| lucene | 2.0⇒2.2 | **63.2** | 50.0 | **63.0** | 59.8 |
| | 2.2⇒2.4 | **73.8** | 37.8 | 62.9 | **69.4** |
| xalan | 2.4⇒2.5 | **45.2** | 39.8 | **56.5** | 54.0 |
| xerces | 1.2⇒1.3 | **38.0** | 33.3 | **47.5** | 26.6 |
| ivy | 1.4⇒2.0 | 34.4 | **38.9** | **34.8** | 24.0 |
| synapse | 1.0⇒1.1 | 47.9 | **50.8** | **42.3** | 31.6 |
| | 1.1⇒1.2 | **57.9** | 56.5 | **54.1** | 53.3 |
| poi | 1.5⇒2.5 | **77.0** | 32.3 | **66.4** | 50.3 |
| | 2.5⇒3.0 | **77.7** | 46.2 | **78.3** | 74.5 |
| **Average** | | **60.0** | 45.2 | **59.7** | 49.0 |

experiments with either Naive Bayes or Logistic Regression as the classification algorithm, the semantic features outperform the PROMISE features 14 out of the 16 times. On average, the Naive Bayes based defect prediction model with semantic features achieves an F1 of 60.0%, which is 14.8 percentage points higher than the Naive Bayes with PROMISE features. Similarly, the average F1 of using semantic features with Logistic Regression is 59.7%, which is 10.7 percentage points higher than Logistic Regression with PROMISE features.

The semantic features automatically learned from the DBN improve the file-level within-project defect prediction and the improvement is not tied to a particular classification algorithm.




(a) Ant (b) Camel (c) Jedit

(d) Log4j (e) Lucene (f) Xalan

(g) Xerces (h) Ivy (i) Synapse

(j) Xerces

Figure 3.9: Results of the DBN-CP, TCA+, and Baseline for CPDP.

Table 3.11: F1 scores of the file-level cross-project defect prediction for target projects explored in RQ1. The F1 scores are measured as a percentage. Better F1 values with statistical significance ($p$-value $< 0.05$) between DBN-CP and TCA+ are indicated with an asterisk (*). The numbers in parentheses are the effect sizes comparative to DBN-CP.

| Source | Target | Cross-Project | | Within-Project |
|---|---|---|---|---|
| | | **DBN-CP*** | **TCA+ (0.274)** | Semantic Features |
| camel1.4 | ant1.6 | **97.9** | 61.6 | 91.4 |
| poi3.0 | ant1.6 | 47.8 | **59.8** | |
| camel1.2 | ant1.7 | 31.2 | **35.4** | 94.2 |
| jEdit3.2 | ant1.7 | 41.7 | **45.5** | |
| ant1.6 | camel1.4 | **31.6** | 29.2 | 78.5 |
| jEdit4.1 | camel1.4 | **69.3** | 33.0 | |
| ant1.5 | camel1.6 | **49.0** | 21.3 | 37.4 |
| lucene2.0 | camel1.6 | **49.2** | 32.1 | |
| xerces1.2 | jEdit4.0 | **51.9** | 32.7 | 57.4 |
| ivy1.4 | jEdit4.0 | 35.0 | **50.2** | |
| camel1.4 | jEdit4.1 | **61.5** | 53.7 | 61.5 |
| log4j1.1 | jEdit4.1 | **50.3** | 41.9 | |
| jEdit4.1 | log4j1.1 | **64.5** | 57.4 | 70.1 |
| lucene2.2 | log4j1.1 | **61.8** | 57.1 | |
| xalan2.5 | lucene2.2 | **59.4** | 56.1 | 65.1 |
| log4j1.1 | lucene2.2 | **69.2** | 52.4 | |
| poi2.5 | lucene2.4 | **64.9** | 54.4 | 77.3 |
| xalan2.4 | lucene2.4 | **61.6** | 60.9 | |
| lucene2.2 | xalan2.5 | **55.0** | 53.0 | 59.5 |
| xerces1.3 | xalan2.5 | 57.2 | **58.1** | |
| xalan2.5 | xerces1.3 | 38.6 | **39.4** | 41.1 |
| ivy2.0 | xerces1.3 | **42.6** | 39.8 | |
| xerces1.3 | ivy2.0 | **45.3** | 40.9 | 35.0 |
| synapse1.2 | ivy2.0 | **82.4** | 38.3 | |
| ivy1.4 | synapse1.1 | **48.9** | 34.8 | 54.4 |
| poi2.5 | synapse1.1 | **42.5** | 37.6 | |
| ivy2.0 | synapse1.2 | 43.3 | **57.0** | 58.3 |
| poi3.0 | synapse1.2 | 51.4 | **54.2** | |
| synapse1.2 | poi3.0 | **66.1** | 65.1 | 80.3 |
| ant1.6 | poi3.0 | **61.9** | 34.3 | |
| synapse1.1 | poi2.5 | **44.6** | 40.6 | 64.0 |
| ant1.6 | poi2.5 | **47.5** | 44.7 | |
| **Average** | | **53.9** | 46.1 | 64.1 |

Table 3.12: F1 scores of file-level cross-project defect prediction for all projects listed in Table 3.4. The F1 scores are measured as a percentage. Better F1 values with statistical significance ($p$-value $< 0.05$) between DBN-CP, TCA+, and Baseline are shown with an asterisk (*). Numbers in parentheses are effect sizes comparative to DBN-CP.

| Source | Target | **DBN-CP** | **TCA+** | **Baseline** | **Within-Project** |
|---|---|---|---|---|---|
| **All Others** | ant | **57.3*** | 50.9 (0.638) | 38.9 (0.774) | 92.8 |
| | camel | **46.1*** | 32.7 (0.484) | 22.7 (0.685) | 57.9 |
| | jEdit | **49.7*** | 43.9 (0.405) | 44.5 (0.261) | 59.4 |
| | log4j | **56.2*** | 52.9 (0.231) | 38.7 (0.450) | 70.1 |
| | lucene | **43.9*** | 41.0 (0.522) | 31.7 (0.541) | 71.2 |
| | xalan | **46.2*** | 44.7 (0.363) | 35.2 (0.577) | 59.5 |
| | xerces | **39.7*** | 33.4 (0.570) | 34.6 (0.513) | 41.1 |
| | ivy | **41.4*** | 28.6 (0.148) | 31.7 (0.782) | 35.0 |
| | synapse | **50.2*** | 47.9 (0.336) | 35.3 (0.636) | 56.4 |
| | poi | **63.2*** | 58.1 (0.307) | 34.9 (0.592) | 72.2 |
| **Average** | | 49.4 | 43.4 (0.401) | 34.8 (0.628) | 61.6 |

### 3.5.2 RQ2: Performance of semantic features for file-level cross-project defect prediction

#### Non-effort-aware evaluation scenario

To answer this question, we compare our file-level cross-project defect prediction technique DBN-CP with TCA+ [194]. DBN-CP runs on the semantic features that are automatically generated by the DBN, while TCA+ uses the PROMISE features. For a fair comparison, we also provide a benchmark of within-project defect prediction. As described in Section 3.4.3, our preliminary experimental evaluation includes a set of 32 cross-project test pairs. Each experiment takes two versions separately from two different projects, with one used as the training set and the other used as the test set. The benchmark of the file-level within-project defect prediction uses the data from an older version of the target project as the training set.

Table 3.11 lists the F1 scores of the DBN-CP, TCA+, and the benchmark within-project defect prediction. The better F1 scores between the DBN-CP and TCA+ are in bold. Regarding the average F1, DBN-CP achieves 53.9%, which is 7.8 percentage points higher than the 46.1% of TCA+. The statistical test also suggests the DBN-CP is overall significantly better than TCA+.

As described in Section 3.4.3, to extensively evaluate the performance of DBN-CP, we use each version from one project as the target project and one version from the other

projects as the source project to form a file-level cross-project experiment test pair. This experiment includes 606 test pairs. Specifically, DBN-CP runs on the semantic features and TCA+ runs on generated features by using the PROMISE features. We also provide two benchmarks, i.e., Baseline and Within-Project. Baseline is the result of cross-project defect prediction with the original PROMISE features.

Table 3.12 shows the average F1 scores of the DBN-CP, TCA+, Baseline, and Within-Project defect prediction on each of the file-level projects. Overall, both DBN-CP and TCA+ deliver better performance than Baseline. Moreover, DBN-CP generates a better F1 than TCA+ on all 10 projects listed, and the improvement is as much as 12.8 percentage points (ivy) and is, on average, 6.0 percentage points higher. Compared with the within-project defect prediction, DBN-CP improves the cross-project defect prediction by reducing the gap to approximately 12 percentage points. The statistical tests also show that overall DBN-CP is significantly better than both TCA+ and Baseline.

Figure 3.9 shows the boxplots of the F1 scores for DBN-CP, TCA+, and Baseline for the 10 projects listed in Table 3.4. Specifically, each boxplot presents the F1 distribution (median and upper/lower quartiles) of each of the three approaches for cross-project file-level defect prediction. The boxplots indicate that overall, both DBN-CP and TCA+ perform better than Baseline, and that DBN-CP performs better than TCA+ and Baseline on almost all projects.

### Effort-aware evaluation scenario

For the effort-aware evaluation, we also calculate the `PofB20` for the DBN-CP, TCA+, and Baseline approaches on each of the target projects.

Table 3.13 shows the `PofB20` of the three file-level cross-project defect prediction models. The highest `PofB20` values among the three approaches are shown in bold. In all the experiments, DBN-CP achieves better `PofB20` than both TCA+ and Baseline. The `PofB20` scores of DBN-CP vary from 21.8 to 37.6 percentage points across the 606 experiments, and the average `PofB20` score of DBN-CP is 29.5 percentage points. Compared to TCA+, the improvement is as high as 21.8 percentage points (Poi) and is, on average, 10.3 percentage points. The results of the Wilcoxon signed-rank test ($p < 0.05$) also indicate that DBN-CP is overall significantly better than both TCA+ and Baseline.

Table 3.13: `PofB20` scores of DBN-based features and traditional features for CPDP. `PofB20` scores are measured in percentage. Better `PofB20` values with statistical significance ($p$-value $< 0.05$) among DBN-CP, TCA+, and Baseline are showed with an asterisk (*). Numbers in parentheses are effect sizes comparative to DBN-CP.

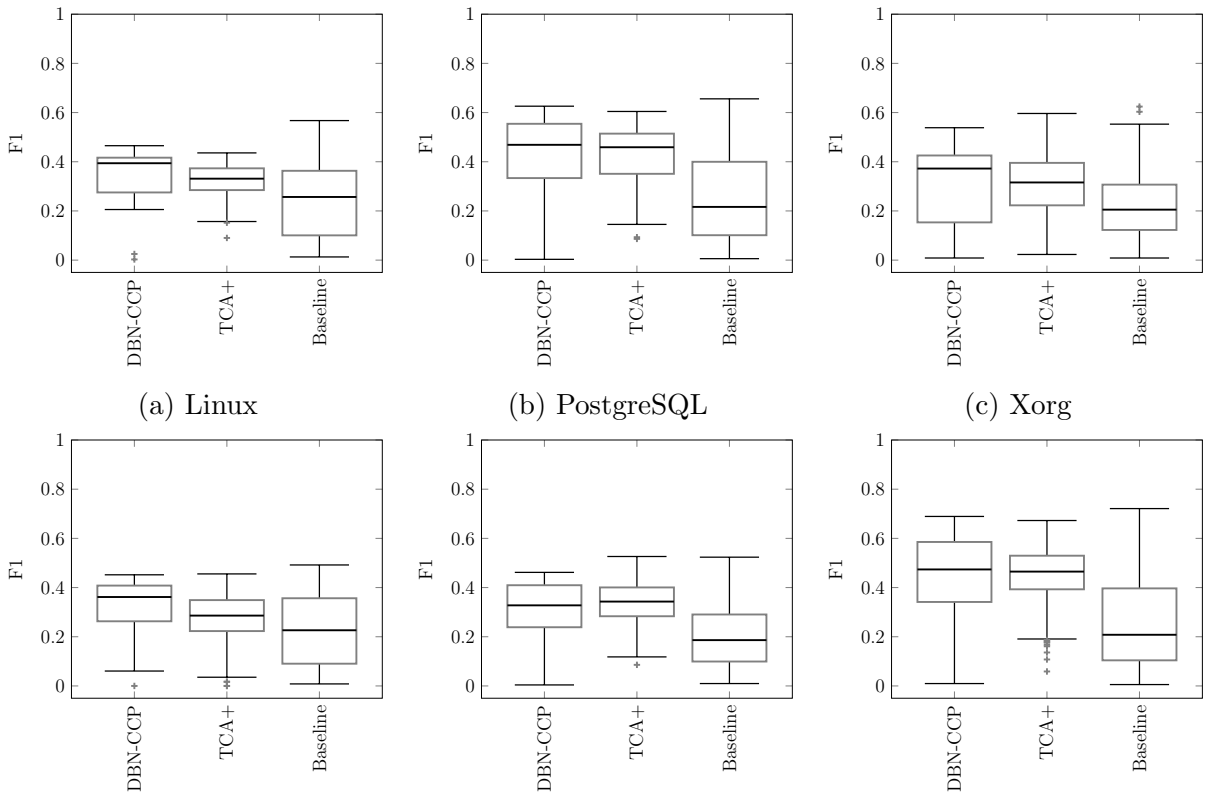
| Source | Target | **DBN-CP** | **TCA+** | **Baseline** |
|---|---|---|---|---|
| **All Others** | ant | **28.3*** | 28.1 (0.434) | 18.3 (0.888) |
| | camel | **32.7*** | 14.8 (0.982) | 10.7 (1) |
| | jEdit | **23.2*** | 21.8 (0.302) | 19.6 (0.462) |
| | log4j | **28.6*** | 19.1 (0.787) | 18.4 (0.855) |
| | lucene | **30.5*** | 15.6 (1) | 10.9 (1) |
| | xalan | **37.6*** | 15.5 (0.900) | 14.6 (0.910) |
| | xerces | **29.1*** | 22.5 (0.479) | 12.9 (0.855) |
| | ivy | **26.5*** | 20.1 (0.488) | 17.9 (0.789) |
| | synapse | **21.8*** | 19.2 (0.133) | 15.7 (0.450) |
| | poi | **36.7*** | 14.9 (0.994) | 11.2 (1) |
| **Average** | | **29.5** | 19.2 (0.650) | 15.0 (0.821) |

DBN-CP significantly improves the performance of file-level cross-project defect prediction under both non-effort-aware and effort-aware evaluation scenarios with a nontrivial effect. This implies that the semantic features learned by the DBN are effective and are able to capture the common characteristics of defects across projects.

### 3.5.3 RQ3: Performance of semantic features for change-level within-project defect prediction

#### Non-effort-aware evaluation scenario

To answer this question, we use different features to build change-level within-project defect prediction models, e.g., DBN-based semantic features, and three change features described in Section 3.4.4 (i.e., the bag-of-words features, the characteristic features, and the meta features). As we described in Section 3.4.3, in the change-level dataset, each project has multiple runs. Thus, we use the training data from each run to build and train the ADTree based prediction model and evaluate its performance on the test data in this run. To show the overall performance, we use the weighted average precision, recall, and F1 following existing work [104, 256].




(a) Linux (b) PostgreSQL (c) Xorg

(d) Jdt (e) Lucene (f) Jackrabbit

Figure 3.10: Results of DBN-CCP, TCA+, and Baseline for CCDP.

Table 3.14: Overall results of the change-level within-project defect prediction. All values are measured as a percentage. The better F1 values with statistical significance ($p$-value $< 0.05$) between the two sets of features are indicated with an asterisk (*). The numbers in parentheses are the effect sizes comparative to the Semantic Features in terms of F1.

| Projects | Features | P | R | F1 |
|---|---|---|---|---|
| Linux | Change Features [104, 256] (0.821) | 28.7 | 49.5 | 36.3 |
| | Semantic Features | **32.5** | **56.9** | **41.3*** |
| PostgreSQL | Change Features (1) | 44.1 | 49.0 | 46.4 |
| | Semantic Features | **51.6** | **58.8** | **55.0*** |
| Xorg | Change Features (0.653) | 29.1 | 51.5 | 37.2 |
| | Semantic Features | **31.7** | **59.4** | **41.4*** |
| JDT | Change Features (0.421) | 32.5 | 41.5 | 36.4 |
| | Semantic Features | 30.2 | **65.9** | **41.4*** |
| Lucene | Change Features (0.136) | 33.0 | 46.8 | 38.7 |
| | Semantic Features | 31.4 | **54.0** | **39.7*** |
| Jackrabbit | Change Features (0.525) | 46.0 | 44.6 | 45.3 |
| | Semantic Features | **49.3** | **50.4** | **49.9*** |
| **Average** | Change Features (0.593) | 36.3 | 50.6 | 40.1 |
| | Semantic Features | **37.6** | **56.6** | **44.8** |

Table 3.14 shows the precision, recall, and F1 of the within-project change-level defect prediction experiments. Overall, the DBN-based features generate better results than the traditional change features in terms of F1. Specifically, for all the projects, the DBN-based features could improve the best existing change features up to 8.6 percentage points in F1, and the improvement is 4.7 percentage points, on average.

### Effort-aware evaluation scenario

We further evaluate the DBN-based semantic features and traditional change features for change-level within-project defect prediction with the `PofB20` metric.

Table 3.15 shows the `PofB20` of the change-level within-project defect prediction models with DBN-based semantic features and the traditional change features. The DBN-based features could achieve better `PofB20` scores than the corresponding change features in all the experiment pairs. The `PofB20` scores (measured as a percentage) of DBN-based features vary from 23.8 to 37.6 across the experiments, and the average `PofB20` score of the defect prediction models with DBN-based features is 29.2. Compared to the change features, the

Table 3.15: `PofB20` scores of the DBN-based features and the traditional features for WCDP. The `PofB20` scores are measured as a percentage. The best values are in bold. The better `PofB20` values with statistical significance ($p$-value $< 0.05$) among these two sets of features are indicated with an asterisk (*). The numbers in parentheses are the effect sizes comparative to the DBN-based semantic features.

| Project | **Semantic Features** | **Change Features** |
|---|---|---|
| Linux | **28.6*** | 25.0 (0.324) |
| PostgreSQL | **29.2*** | 8.1 (1) |
| Xorg | **37.6*** | 24.8 (0.901) |
| JDT | **23.8*** | 14.5 (1) |
| Lucene | **28.1*** | 21.9 (0.887) |
| Jackrabbit | **27.9*** | 21.3 (0.621) |
| **Average** | **29.2** | 19.3 (0.789) |

Table 3.16: F1 scores of change-level cross-project defect prediction for all projects. The F1 scores are measured as percentages. The better F1 values with statistical significance ($p$-value $< 0.05$) between DBN-CCP, TCA+, and Baseline are indicated with an asterisk (*). The numbers in parentheses are the effect sizes comparative to DBN-CCP.

| Source | Target | **DBN-CCP** | **TCA+** | **Baseline** | **Within** |
|---|---|---|---|---|---|
| **All Others** | Linux | **35.1*** | 32.4 (0.295) | 24.7 (0.439) | 41.3 |
| | PostgreSQL | **44.2*** | 43.6 (0.130) | 25.7 (0.370) | 55.0 |
| | Xorg | **31.8*** | 30.4 (0.128) | 22.8 (0.310) | 41.4 |
| | JDT | **33.3*** | 27.3 (0.360) | 22.6 (0.566) | 41.4 |
| | Lucene | **31.3*** | 30.2 (0.129) | 21.3 (0.520) | 39.7 |
| | Jackrabbit | **44.4*** | 43.3 (0.131) | 26.5 (0.463) | 49.9 |
| **Average** | | 36.7 | 34.5 (0.196) | 23.9 (0.445) | 44.7 |

improvement could be up to 21.1 percentage points (PostgreSQL) and is 9.9 percentage points, on average. In addition, the statistical test, i.e., the Wilcoxon signed-rank test ($p < 0.05$), also suggests that the DBN-based features are overall significantly better than the change features under both the non-effort-aware and effort-aware evaluation scenarios.

The semantic features automatically learned from the DBN could improve the change-level within-project defect prediction with statistical significance under both non-effort-aware and effort-aware evaluation scenarios with nontrivial effect sizes.

### 3.5.4 RQ4: Performance of semantic features for change-level cross-project defect prediction

#### Non-effort-aware evaluation scenario

To answer this question, we compare our cross-project change-level defect prediction technique DBN-CCP with TCA+. For a fair comparison, we also provide two benchmarks, i.e., Baseline and Within-Project. The Baseline is the result of change-level defect prediction with the original change features. As we described in Section 3.4.9, we use the test data of one run from one project as the target project and the training data of one run from a different project as the source project to form the change-level cross-project test pairs (in total 1,380 pairs). For each test pair, we build the ADTree based defect prediction model using the three different sets of features.

Table 3.16 shows the average F1 scores of the DBN-CCP, TCA+, Baseline, and Within-Project for each of the change-level projects. Overall, both DBN-CCP and TCA+ deliver better performance than Baseline. Moreover, DBN-CCP generates a better F1 than TCA+ on all the projects on average. The improvement is as high as 6.0 percentage points and is 2.2 percentage points higher, on average. Compared to the within-project defect prediction, DBN-CCP improves the cross-project defect prediction by reducing the gap to only 8.0 percentage points.

Figure 3.10 shows the boxplots of the F1 scores for DBN-CCP, TCA+, and Baseline for the six projects listed in Table 3.5. Specifically, each boxplot presents the F1 distribution (median and upper/lower quartiles) of each of the three approaches for the change-level cross-project defect prediction. The boxplots show that overall both DBN-CCP and TCA+ perform better than Baseline, moreover DBN-CCP performs better than TCA+ and Baseline on almost all projects.

#### Effort-aware evaluation scenario

We also calculate the `PofB20` score for the DBN-CCP, TCA+, and Baseline approaches on each of the target projects when conducting change-level cross-project defect prediction. Table 3.17 shows the `PofB20` values of the three change-level cross-project defect prediction models. The highest `PofB20` values among the three approaches are shown in bold. DBN-CCP achieves better `PofB20` scores than both TCA+ and Baseline. On average, the `PofB20` score (measured as a percentage) of DBN-CCP is 21.9. Compared to TCA+, the improvement can be up to 3.0 percentage points (Jdt) and is 1.3 percentage points, on

Table 3.17: PofB20 scores of the DBN-based features and traditional features for CCDP. The PofB20 scores are measured as a percentage. The best values are in bold. The better PofB20 values with statistical significance ($p$-value $< 0.05$) among the DBN-CCP, TCA+, and Baseline are indicated with an asterisk (*). The numbers in parentheses are the effect sizes comparative to DBN-CCP.

| Source | Target | **DBN-CCP** | **TCA+** | **Baseline** |
|---|---|---|---|---|
| **All Others** | Linux | **24.7*** | 24.1 (0.255) | 18.5 (0.500) |
| | PostgreSQL | **20.7** | 20.3 (0.019) | 15.9 (0.438) |
| | Xorg | **22.7*** | 22.0 (0.110) | 19.7 (0.511) |
| | JDT | **25.6*** | 22.6 (0.273) | 13.5 (0.360) |
| | Lucene | **18.1** | 18.0 (0.030) | 17.0 (0.371) |
| | Jackrabbit | **19.3*** | 16.4 (0.352) | 16.1 (0.343) |
| **Average** | | **21.9** | 20.6 (0.180) | 16.8 (0.421) |

Table 3.18: Time and space costs of generating semantic features for file-level defect prediction (s: second).

| Project | Generating Features | |
|---|---|---|
| | Time (s) | Memory (MB) |
| ant | 15.5 | 2.8 |
| camel | 32.0 | 5.5 |
| jEdit | 18.1 | 3.3 |
| log4j | 10.1 | 2.2 |
| lucene | 11.1 | 2.4 |
| xalan | 29.6 | 6.2 |
| xerces | 13.9 | 5.8 |
| ivy | 8.0 | 2.2 |
| synapse | 8.5 | 1.9 |
| poi | 11.9 | 4.4 |

average. The results of the Wilcoxon signed-rank test ($p < 0.05$) also indicate that the performance of DBN-CCP is, overall, significantly better than TCA+ and Baseline under both the non-effort-aware and effort-aware evaluation scenarios.

DBN-CCP significantly improves the performance of the change-level cross-project defect prediction compared to the traditional change features with a nontrivial effect.

### 3.5.5 Time and Memory Overhead

To understand the space cost of file-level defect prediction, during file-level defect prediction experiments, we keep track of the time cost and memory space cost for our DBN-based feature generation process (details are in Section 3.3.3). In addition, we also have recorded the time cost for tuning the DBN models in our experiments. The other processes, including parsing source code, handling noise, mapping tokens, building models, and predicting defects, are all common procedures, so we do not analyze their costs.

As described in Section 3.4.5, we tune the three parameters, i.e., the number of hidden layers, the number of nodes in each layer, and the number of iterations, for the randomly selected five projects. To find the best combination among the three parameters, we have $11 \times 8 \times 10$ experiments. In total, the tuning process costs approximately 5 hours.

Table 3.18 shows the time cost and the memory space cost of each project for generating semantic features. As shown in Table 3.8, `ant` has two sets of within-project defect prediction experiments, which are `ant 1.5` $\Rightarrow$ `1.6` and `ant 1.6` $\Rightarrow$ `1.7`. On average, it takes the two experiments 15.5 seconds and 2.8 MB memory for the DBN to generate the semantic features for both the training data and the test data. Among all the projects, the time cost of automatically generating the semantic features varies from 8.0 seconds (`ivy`) to 32.0 seconds (`camel`). For the memory space cost, it takes less than 6.5MB for all the examined projects.

In addition, we also keep track of the time and memory space cost for generating DBN-based features for the change-level defect prediction during our experiments. Different from the file-level defect prediction that predicts whether a file contains bugs or not, change-level defect prediction predicts whether a change is buggy or clean. Source files often contain hundreds of LOC, while changes often have fewer lines than files. Thus, both the time and memory costs of generating DBN-based features for changes are smaller than those for files. In our experiments, the average time cost and memory cost of generating DBN-based features for changes are 2.4 seconds and 0.6 MB.

Our DBN-based approach to automatically learning semantic features is applicable in practice.

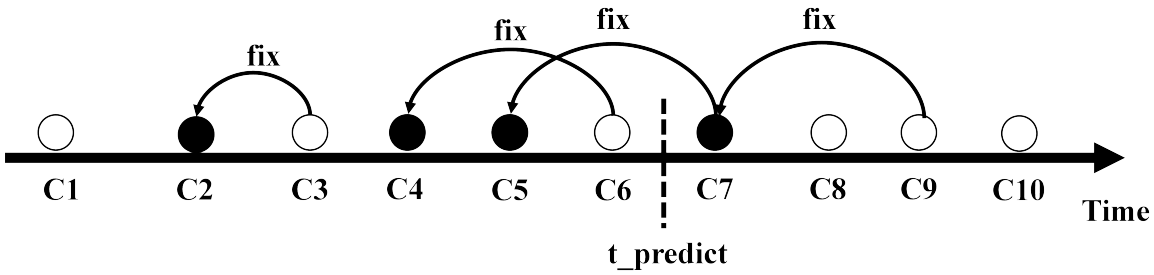



Figure 3.11: An example to illustrate the issues of using cross-validation to evaluate change-level defect prediction. Circles are clean changes and dots are buggy.

Table 3.19: Information gain of different types of tokens and their combinations that are used for generating DBN-based semantic features in change-level defect prediction. **Spearman** is the Spearman correlation value between the information gain and the prediction results of different types of tokens on a project.

| Project | added | deleted | context | added+deleted | added+context | deleted+context | all | Spearman |
|---|---|---|---|---|---|---|---|---|
| Jackrabbit | 8.2 | 7.4 | 7.4 | 8.2 | 8.6 | 8.4 | 8.9 | 0.96 |
| Lucene | 6.9 | 6.5 | 6.3 | 7.1 | 7.6 | 7.4 | 7.7 | 0.89 |
| Jdt | 8.1 | 7.0 | 7.1 | 8.0 | 9.8 | 8.4 | 8.9 | 0.64 |
| Linux | 4.2 | 3.8 | 3.0 | 4.3 | 5.0 | 4.9 | 5.1 | 0.90 |
| Postgresql | 5.9 | 5.5 | 5.3 | 6.0 | 6.7 | 6.6 | 7.6 | 0.96 |
| Xorg | 4.8 | 4.0 | 4.9 | 5.5 | 6.0 | 5.9 | 6.1 | 0.82 |
| Average | 6.4 | 5.7 | 5.6 | 6.5 | 7.3 | 6.9 | 7.4 | 0.86 |

## 3.6 Discussion

### 3.6.1 Issues of Using Cross-Validation to Evaluate Change-level Defect Prediction

As described in Section 3.3.4, this work did not use the widely used cross-validation to evaluate change-level defect prediction. The main reason is that change-level defect prediction is time sensitive, i.e., changes follow a certain order in time. For example, in Figure 3.11, changes C1–C10 are committed chronologically, where C1 is the earliest, and C10 is the latest. Dots denote the buggy changes and circles denote the clean changes. An arrow links a buggy change and the corresponding change that fixes the buggy change. For example, C6 fixes the bugs in C4; therefore, C4 is labeled as a buggy change.

Because of the characteristic of changes, using cross-validation to evaluate change classification may introduce the following two issues. First, cross-validation will use future data for prediction. In this example, 10-fold cross-validation will make each change the test set in each iteration. For example, it will use C2–C10 to predict whether C1 is buggy or not, which does not match a real-world usage scenario where we typically want to make the prediction at the time when C1 is committed and by then C2–C10 are not available yet. Second, cross-validation could mislabel changes. Using cross-validation, changes will be labeled as shown in Figure 3.11. For example, C5 will be labeled buggy. However, in practice, when we predict whether C6 is buggy, we would only have information at time *t_predict*. Therefore, C5 should be clean at time *t_predict* because C7 was nonexistent at that time. It is incorrect for cross-validation to consider C5 buggy when we predict the label of C6 at time *t_predict*, because we would not know that C5 is buggy at time *t_predict*. Due to the above two issues, cross-validation is inaccurate for evaluating defect prediction in practice. Thus, in this work, we did not adopt cross-validation to evaluate defect prediction tasks, we divided the data into different folds chronologically and evaluated the overall performance on all the folds for avoiding bias.

### 3.6.2 Why Do DBN-based Semantic Features Work?

Our experiments in Section 3.5 show that compared to traditional features, the DBN-based features that are directly learned from source code deliver significantly better performance for all the four defect prediction tasks investigated in this work. The probable reasons for the outstanding performance of DBN-based features are summarized as follows.

First, the DBN models generate features with more complex network connections. These network connections enable the DBN models to generate features with multiple levels of abstraction and high-level semantics. In this work, the generated DBN features are weighted combinations/vectors of original input source code, which could represent patterns of the usages of the input source code, e.g., method usages, control-flow usages, etc. While traditional features often focus on statistical information of the source code, e.g., LOC, the number of function calls, etc., which cannot capture the semantic information. Although the `Bag-of-words feature` or the `Characteristic feature` (details are in Section 3.4.4) are derived from the raw programming tokens, they consider each token as an independent feature element and cannot represent the contextual and structural information among the raw tokens. Thus, these features have underperformed.

Second, the DBN-based features are more capable of distinguishing between the semantic information of different code snippets, especially for code snippets that have similar

source code characteristics. For example, as shown in Figure 3.1, the traditional features (e.g., code complexity) of the two code snippets are identical. Training prediction models containing them will degrade the discrimination ability of classifiers and consequently hurt the prediction performance. While the DBN-based features can make a difference, as shown in Figure 3.3, the different structural and contextual information among tokens of these two code snippets enables the DBN model to generate different features to distinguish between these two code snippets.

### 3.6.3 Efficiency of Different Types of Tokens in Change-level Defect Prediction

As described in Section 3.4, to achieve better prediction performance for change-level defect prediction, we use the combination of the three types of tokens, i.e., `added`, `deleted`, and `context`, to generate DBN-based semantic features. In this section, we further examine the reason why the combination outperforms each of the three types of tokens extracted from changes. One possible reason is that, the combination contains more information than any of the three types of tokens. To explore this, we leverage the information gain [64], which is a widely used metric to measure how much information there is in a given event, to measure the information in each of the three types of tokens and their different combinations.

Specifically, given a document $s = \{a_1 \ ... \ a_n\}$ of length $n$, $a_1$ to $a_n$ are tokens in the document $s$. The information gain of this document $H(s)$ is measured as follows:

$$H(s) = \sum_{i=1}^{n} -p_i \log p_i \tag{3.9}$$

where $p_i$ is the probability of token $a_i$ in the document $s$. We use the TF (term frequency) of token $a_i$ to represent its probability in the document $s$.

To calculate the information gain of a specific type of token in changes, we first collect all seven types of tokens from all the changes in a project. Then, we calculate the information gain of a specific type of tokens extracted from all changes of a project. Table 3.19 shows the various information gains of the three types of tokens and their combinations. Overall, among the basic three types of tokens, `added` and `deleted` contain more information than `context`. The combination of either two of them could achieve better performance than either of the two types of tokens. In addition, the combination of all the three types of tokens contains more information than any other combinations. We further compute the Spearman correlation between the value of information gain and the prediction result

of different types of tokens in a project. The high correlation value (on average 0.86) indicates that the prediction result of DBN-based features generated from a specific type of token has a positive correlation with its information gain. This explains why DBN-based features generated from the combination of all the three types of tokens achieve the best performance.

### 3.6.4 Analysis of the Performance

In this section, we evaluate the performance of our proposed DBN-based semantic features on both file-level and change-level defect prediction tasks. We can observe that the improvement of DBN-based semantic features on file-level defect prediction is generally better than change-level defect prediction. The main reason for this phenomenon is that a file generally contains more information than a change. Thus, file-level defect prediction data often provide more context to a DBN model, allowing it to learn more accurate features.

We also note that our approach achieves better performance on some projects than others for file-level with-project defect prediction, e.g., it achieves an F1 of 94.2% on `ant` and an F1 of approximately 80% on `camel`, `lucene`, `poi`, and `jEdit`. This is because we use these projects, i.e., `ant`, `camel`, `lucene`, `poi`, and `jEdit`, as data to train a DBN model for generating features. During the training process, we tune the DBN parameters based on the performance of the defect prediction models with the generated features for the five projects (details are presented in Section 3.4.5). Because the training process is an optimization task to generate features that may produce the best performance for the training dataset, the features fit the training dataset better. Thus, our approach achieves relatively higher F1 values for the five projects (`ant`, `camel`, `lucene`, `poi`, and `jEdit`) than other projects. This may be a risk of overfitting. However, this may also suggest that training a DBN model by using a project's own history data is appropriate when applying our approach to the project.

### 3.6.5 Performance on Open-source Commercial Projects

In Section 3.4, we evaluated the DBN-based semantic features on 15 open-source projects (i.e., the projects listed in Table 3.4 and Table 3.5). To explore the performance of the

Table 3.20: The four open-source commercial projects evaluated. **Lang** is the programming language used for the project. **LOC** is the number of the line of code. **First Date** is the date of the first commit of a project, while **Last Date** is the date of the latest commit. **Changes** are the number of changes. **TrSize** is the average size of the training data on all runs. **TSize** is the average size of the test data on all runs. **Rate** is the average buggy rate for each project. **NR** is the number of runs for each subject.

| Project | Lang | LOC | First Date | Last Date | Changes | TrSize | TSize | Rate(%) | # NR |
|---|---|---|---|---|---|---|---|---|---|
| Buck (Facebook) | JAVA | 296K | 2013-04-18 | 2018-03-23 | 92K | 31k | 19k | 7.2 | 3 |
| Hhvm (Facebook) | C++ | 1M | 2010-02-03 | 2018-04-06 | 120K | 51K | 14K | 11.6 | 3 |
| Guava (Google) | JAVA | 380K | 2009-06-18 | 2018-03-21 | 29K | 8.6K | 8.8K | 4.0 | 2 |
| Skia (Google) | C++ | 765K | 2006-09-20 | 2018-04-07 | 147K | 48K | 22K | 30.6 | 5 |

DBN-based semantic features on commercial projects, we apply our approach to four additional open-source commercial projects, i.e., Buck[4], Hhvm[5], Guava[6], and Skia[7]. Buck is a build system developed and used by Facebook. Hhvm is a virtual machine, which was also developed and is currently used by Facebook. Guava is a set of Google's core libraries for Java. Skia is a complete 2D graphics library for drawing text, geometries, and images developed and used by Google. These four projects were originally developed and maintained by Facebook and Google and became open-source projects recently. We selected these four projects, because they are the largest Java/C++ projects (in terms of commit size). To collect the change-level data, we use the same approaches as we described in Section 2.1.2 and Section 3.4.4 to label the changes and collect the features for each change. The details of the four open-source commercial projects are listed in Table 3.20.

With these four additional projects, we conduct change-level within-project and change-level cross-project defect prediction tasks to compare the DBN-based semantic features to traditional features under both the non-effort-aware and effort-aware scenarios. Note that we adopt the same procedures to tune the DBN models and generate semantic features as described in Section 3.4.8 and Section 3.4.9.

Table 3.21 shows the results of the change-level within-project prediction on the four projects. Overall, the DBN-based features generate better results than traditional change features in terms of F1, which is consistent with our previous experiment results on pure open-source projects 3.14. Specifically, for all four projects, DBN-based features could improve the best existing change features up to 6.6 percentage points in F1, and on average

[4]`https://buckbuild.com/`
[5]`https://hhvm.com/`
[6]`https://github.com/google/guava`
[7]`https://skia.org/`

the improvement is 5.4 percentage points. These improvements are consistent with our previous experimental results on open-source projects (i.e., the best improvement is 8.6 percentage points and the average improvement is 4.7 percentage points).

Table 3.21: Results of WCDP on the four projects. All values are measured in percentage. Better F1 values with statistical significance ($p$-value $< 0.05$) between the two sets of features are showed with an asterisk (*). Numbers in parentheses are effect sizes comparative to Semantic Features in terms of F1.

| Projects | Features | P | R | F1 |
|---|---|---|---|---|
| Buck | Change Features (0.542) | 10.9 | 39.9 | 17.2 |
| | Semantic Features | **14.0** | **46.6** | **21.6*** |
| Hhvm | Change Features(0.477) | 14.2 | **39.8** | 20.9 |
| | Semantic Features | **23.6** | 33.1 | **27.5*** |
| Guava | Change Features (0.685) | 7.4 | 58.0 | 13.1 |
| | Semantic Features | **10.5** | **63.2** | **18.1*** |
| Skia | Change Features (0.710) | 34.5 | 40.4 | 37.3 |
| | Semantic Features | **45.2** | **42.9** | **44.0*** |
| **Average** | Change Features (0.622) | 16.7 | 44.5 | 22.4 |
| | Semantic Features | **23.3** | **46.5** | **27.8** |

Table 3.22: F1 scores of change-level cross-project defect prediction for the four projects. Better F1 values with statistical significance ($p$-value $< 0.05$) between DBN-CCP, TCA+, and Baseline are showed with an asterisk (*). Numbers in parentheses are effect sizes comparative to DBN-CCP.

| Source | Target | **DBN-CCP** | **TCA+** | **Baseline** | **Within** |
|---|---|---|---|---|---|
| **All Others** | Buck | **14.3*** | 11.9 (0.459) | 10.8 (0.693) | 21.6 |
| | Hhvm | **22.6*** | 21.7 (0.017) | 20.2 (0.112) | 27.5 |
| | Guava | 7.2 | **7.6*** (-0.024) | 4.2 (0.407) | 18.1 |
| | Skia | **47.9*** | 38.0 (0.613) | 36.9 (0.765) | 44.0 |
| **Average** | | **23.0*** | 19.8 (0.358) | 18.0 (0.422) | 27.8 |

Table 3.22 shows the results of the change-level cross-project prediction for the four projects. Overall, DBN-CCP generates a better F1 than both TCA+ and Baseline for all four projects on average. The improvement is up to 9.9 percentage points and is 3.2 percentage points on average. The results are also consistent with our previous change-level cross-project defect prediction results listed in 3.16.

Table 3.23: PofB20 scores of DBN-based features and traditional features for WCDP on the four projects. PofB20 scores are measured in percentage. Better PofB20 values with statistical significance ($p$-value $< 0.05$) among these two sets of features are showed with an asterisk (*). Numbers in parentheses are effect sizes comparative to DBN-based semantic features.

| Project | **Semantic Features** | **Change Features** |
|---|---|---|
| Buck | **28.2*** | 14.7 (1) |
| Hhvm | **21.9*** | 13.3 (0.882) |
| Guava | **17.4*** | 15.5 (0.351) |
| Skia | **27.0*** | 21.3 (0.655) |
| **Average** | **23.6*** | 16.2 (0.520) |

Table 3.24: PofB20 scores of DBN-based features and traditional features for CCDP on the four projects. Better PofB20 values with statistical significance ($p$-value $< 0.05$) among DBN-CCP, TCA+, and Baseline are showed with an asterisk (*). Numbers in parentheses are effect sizes comparative to DBN-CCP.

| Source | Target | **DBN-CCP** | **TCA+** | **Baseline** |
|---|---|---|---|---|
| **All Others** | Buck | **25.0*** | 17.9 (0.801) | 14.2 (1) |
| | Hhvm | 13.4 | **17.4*** (-0.653) | 12.0 (0.455) |
| | Guava | **14.3*** | 13.0 (0.625) | 10.5 (0.746) |
| | Skia | **18.2*** | 17.1 (0.210) | 16.9 (0.437) |
| **Average** | | **18.7*** | 16.4 (0.623) | 13.4 (0.766) |

We also calculate the PofB20 values for both the change-level within-project and cross-project approaches. Table 3.23 shows the PofB20 values of the change-level within-project defect prediction models with DBN-based semantic features and the traditional change features. DBN-based features achieve better PofB20 values than the corresponding change features for all experiment pairs. The improvement is up to 13.5 percentage points (Buck) and is 7.4 percentage points on average. Table 3.24 shows the PofB20 values of the three change-level cross-project defect prediction approaches. Similar to our previous results, DBN-CCP achieves better PofB20 values than both TCA+ and Baseline on average. Compared to TCA+, the improvement is as high as 7.1 percentage points and is 2.3 percentage points, on average.

DBN-based semantic features outperform traditional features on four open-source commercial projects from Facebook and Google, which indicates that DBN-based semantic features are also applicable for improving defect prediction for open-source commercial projects.

Note that the results on commercial open-source projects are worse than that of open-source projects. One of the possible reasons is that the buggy rates of open-source projects used in this study are significantly higher than the buggy rates of commercial open-source projects, i.e., the average buggy rate of open-source projects is 24.4%, while the average buggy rate of commercial open-source projects is 13.0%. The low buggy rate can hurt the precision of prediction performance and result in a low F1 measure [256].

## 3.7 Threats to Validity

### Implementation of TCA+

For the comparative analysis, we compare our cross-project defect prediction models with TCA+ [194], which is the state-of-the-art cross-project defect prediction technique with traditional features. Since the original implementation is not released, we reimplemented our own version of TCA+. Although we strictly followed the procedures described in their work, our new implementation may not reflect all the implementation details of the original TCA+. We test our implementation with the data provided by their work. Since our implementation can generate the same results, we are confident that our implementation reflects the original TCA+.

In this work we did not evaluate our DBN-based feature generation approach on projects used for evaluating TCA+ [194]. This is because our DBN-based feature generation approach to within-project defect prediction works on data of two different versions from the same project. However, the datasets used in [194] only provided one version of defect data for each of their eight projects, which are unsuitable for evaluating our approach to within-project defect prediction. To reduce this threat, we evaluated TCA+ and our approach on the publicly available projects from PROMISE.

### Project Selection

The examined projects in this work have a large variance in average buggy rates. We have tried our best to make our dataset general and representative. However, it is still possible

that the 15 projects used in our experiments are not generalizable enough to represent all software projects. Our approach might generate better or worse results for other projects that are not used in the experiments. We mitigate this threat by selecting projects of different functionalities (operating systems, servers, and desktop applications) that are developed in different programming languages (C and Java).

Our approach to generating semantic features is only evaluated on open-source projects. While we believe that this approach should be generalizable to proprietary software, evaluating our approach on proprietary software is challenging, because the approach requires AST analysis of source code. We mitigate this threat by applying our approach to four open-source commercial projects that were originally developed and maintained by Google and Facebook and are open-source now. The performance of these projects suggests our proposed DBN-based semantic features could deliver better results than traditional features.

#### Labeling Data

Following previous work [124, 245], the labeling process is automatically completed with the annotating or blaming function in VCS. It is known that this process can introduce noise [104, 123]. The noise in the data can potentially harm the performance of defect prediction. Manual inspection of the process shows reasonable precision and recall on open source projects [104]. To mitigate this threat, we use the noise data filtering algorithm introduced in [123].

## 3.8 Summary

In this chapter we introduce an approach that leverages a representation-learning algorithm, i.e., deep learning, to learn semantic representation directly from source code for defect prediction. Specifically, we deploy a deep belief network to learn semantic features from programs' ASTs (for file-level defect prediction models) and source code changes (for change-level defect prediction models) automatically, and leverage the learned semantic features to build prediction models.

We examined the effectiveness of the learned DBN-based semantic features on two file-level defect prediction tasks, i.e., file-level within-project defect prediction (WPDP) and file-level cross-project defect prediction (CPDP), and two change-level defect prediction

tasks, i.e., change-level within-project defect prediction (WCDP) and change-level cross-project defect prediction (CCDP). To conduct comprehensive performance evaluations, we employed both non-effort-aware and effort-aware evaluation metrics.

For file-level defect prediction tasks, our evaluations were conducted on 26 versions of data from 10 open-source projects. Our results show that the DBN-based semantic features improve WPDP on average by 13.3 percentage points (in F1), and outperform the state-of-the-art CPDP with traditional features on average by 6.0 percentage points. For change-level defect prediction, our evaluations were conducted on more than 1M changes from six open-source projects and four open-source commercial projects. The experimental results indicate that the DBN-based semantic features can improve WCDP on average by 5.1 percentage points, and improve the state-of-the-art CCDP technique with traditional change-level features, on average, by 2.9 percentage points. In addition, under the effort-aware evaluation scenario, our DBN-based semantic features can outperform traditional features for both the file-level and the change-level defect prediction.

# Chapter 4

# Leveraging N-gram Language Models to Improve Rule-based Static Bug Detection

This chapter presents our approach (i.e., `Bugram`) that leverages n-gram language models to detect bugs that cannot be detected by rule-based bug detection tools. Specifically, the proposed approach leverages n-gram language models to detect bugs by calculating and ranking the probabilities of program tokens. `Bugram` was presented at the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE'16 [278]). This work is an collaboration with Devin Chollak, an early version of detection bugs with N-gram language models has been presented in Devin's Master thesis [47]. The differences between `Bugram` and Devin's Master thesis, i.e., NGDetection, include: 1) `Bugram` and NGDetection are two different approaches to detecting bugs, NGDetection is a rule-based bug detector, which leverages n-gram to mine rules and detects the violations to these rules as bugs. While `Bugram` is a non-rule-based bug detector, which directly detect bugs by calculating the probabilities of token sequences. 2) We further compare `Bugram` with five typical rule-based bug detectors, i.e., PR-Miner, improved PR-Miner, JADET, Tikanga, and GrouMiner.

## 4.1 Motivation

Software bug detection techniques have been shown to improve software reliability by finding previously unknown bugs in mature software projects [80, 95]. Rule-based bug

detection approaches infer likely programming rules from source code [8, 30, 38, 144, 146, 250, 268, 269, 301], version histories [30, 113, 292], and source code comments [255, 257]. These approaches detect violations of these rules as potential bugs.

Frequent itemset mining techniques were used to mine rules that capture the co-occurrence of methods and variables. Violations of these rules are reported as bugs [30,38, 144]. Along this line, more complex graph models are combined with frequent itemset mining techniques, which focus on mining programming rules that capture both method order and control flow information to detect violations of these complex rules [70, 199, 284, 285].

Let $ABC$ denote a sequence of calls to the methods $A$, $B$, and $C$. Imagine a contrived program that includes 98 occurrences of the sequence $ABC$, two occurrences of $ABD$, and a single occurrence of $EFG$. Existing rule-based bug detection approaches, such as PR-Miner [144], JADET [285], Tikanga [284], and GrouMiner [199] infer rules based on the *conditional probabilities* of method calls, for example, the conditional probability $P(C|AB)$ which denotes the likelihood of seeing a call to method $C$ after the sequence of calls $AB$. In our example, $P(C|AB) = \frac{98}{98+2} = 98\%$. This probability is higher than the threshold used by PR-Miner, which is 90%, and therefore, the potential rule that $C$ should appear after $AB$, denoted as $\{AB => C\}$, is selected as a high-probability rule. The *confidence* of this rule is its conditional probability, which is 98%. Given this rule, the sequence $ABD$ is flagged as a bug, because $C$ instead of $D$ is expected to follow $AB$.

It has recently been demonstrated that *n-gram language models* [39] can capture the regularities of software source code [88, 224]. To take advantage of the **n-gram language model**, which provides us with a Markov model for tokens, we propose an n-gram language model based bug detection technique, called *Bugram*. The assumption is that *low probability token sequences in a program are unusual, which may indicate bugs, bad practices, or unusual/special uses of code of which developers may want to be aware.*

While existing studies leverage n-grams for detecting clone bugs [96], localizing faults [196, 316], and code search [117], including some that use the term n-gram models [96, 117], these studies do not leverage n-gram models. Instead, they use *n-grams*, which are token sequences, while *n-gram models* are Markov models built on n-grams. On the other hand, n-gram models have been used for code completion and suggestion [78,198,225], fault localization [35], and coding style checking [10, 85]. The focus of this work is leveraging n-gram models for bug detection, which has its own challenges and requires a different design, as detailed in Section 4.3.

Instead of using *conditional probabilities*, Bugram highlights suspicious call sequences based on their *absolute probabilities* in the program. So in the example above, our approach evaluates the absolute probability $P(ABC)$ of the full sequence $ABC$, which is in contrast

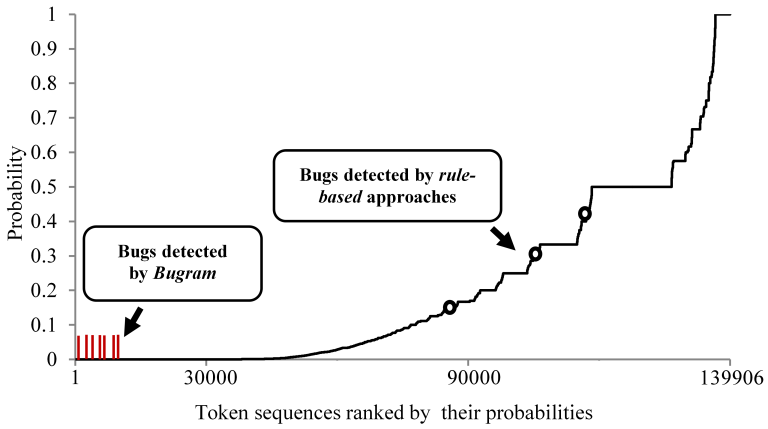



Figure 4.1: Bugs detected by Bugram versus bugs detected by rule-based techniques in the latest version of Hadoop

to PR-Miner which evaluates the conditional probability of $P(C|AB)$. The two sequences with the lowest probabilities in the program are $EFG$ and $ABD$, which have a probability that is markedly lower than that of $ABC$. By selecting sequences with low absolute probabilities, Bugram is able to recognize that $ABD$ and $EFG$ are both suspicious sequences. Notably, $EFG$ is not recognized as a suspicious sequence by PR-Miner, despite having only a single occurrence in the program, because there are no rules with high confidence related to $EFG$. In addition, it is known that even if a rule exists, but the rule pattern does not appear frequently enough, the rule cannot be learned, thus missing many bugs [144].

More broadly, rule-based approaches detect bugs from common program patterns, while Bugram detects sequences which are overall uncommon in the program. These two approaches target different types of program abnormalities, and will ultimately detect different types of bugs, as illustrated in Figure 4.1. The curve shows the probabilities of sequences in the Java project Hadoop sorted ascendingly. The bars depict examples of bugs that can be detected by our n-gram-based approaches, while circles represent examples of bugs that can be detected by rule-based approaches.

In this work, we study whether our n-gram-based approach can detect bugs in real-world software that rule-based approaches cannot find. In addition, we study whether Bugram is more *precise* than rule-based approaches, i.e., whether Bugram reports a smaller portion of false bugs than rule-based approaches.

(a) **Method call sequence from a buggy code snippet (appears once): `[isDebugEnabled(), debug(), indent(), stringify()]`**

```
if (LOG.isDebugEnabled()) {
     LOG.debug(indent(depth)+"converting␣from
␣␣␣␣␣␣␣␣␣␣␣Pig␣" + pigType + "␣" + value +
           "␣using␣" + stringify(schema));
}
```

(b) **A similar but correct method call sequence (appears three times): `[isDebugEnabled(), debug(), indent(), toString())]`**

```
if (LOG.isDebugEnabled()) {
     LOG.debug(indent(depth)+"converting␣from
␣␣␣␣␣␣␣␣␣␣␣␣Pig␣" + pigType + "␣" + toString(value) +
            "␣using␣" + stringify(schema));
}
```

Figure 4.2: A motivating example from the latest version 0.15.0 of the project Pig. Bugram automatically detected a real bug in (a), which has been *confirmed and fixed* by Pig developers after we reported it.

### 4.1.1 A Motivating Example

Existing rule-based techniques detect potential bugs by using mined rules with enough confidence and *support* (the number of occurrences) to avoid generating a large number of false bugs. For example, PR-Miner [144] requires the confidence of a method call sequence to be over 90% and the support larger than 15 to be identified as a rule, missing opportunities to detect many bugs. For example, Figure 4.2 shows a real bug detected by our tool in the latest version of Pig, which has already been confirmed by Pig developers. The code snippet in Figure 4.2(a) contains a bug: for the purpose of logging, the code snippet should convert the object `value` to a string by calling the `toString` method, but it does not. The method call sequence of the buggy code snippet is `[isDebugEnabled, debug, indent, stringify]`, which appears only once in the program. A similar but correct code snippet with a method call sequence `[isDebugEnabled, debug, indent, toString]` appears three times. One of the appearances is shown in Figure 4.2(b).

Using rule-based bug detection approaches such as PR-Miner, a potential rule that may detect this bug is `[isDebugEnabled, debug, indent => toString]`. Since PR-Miner groups method calls into a set, which means it ignores the order of method calls for example, all other rules that can potentially detect this bug are `[debug, indent => toString]`, `[isDebugEnabled, indent => toString]`, `[indent => toString]`, `[debug => toString]`, `[isDebugEnabled, debug => toString]`, and `[isDebugEnabled =>`

`toString]`. The confidence of each potential rule is only 75%, considering only these four code snippets. If we consider the entire Pig project, the confidences of these rules are even lower, ranging from 19.5% to 28.8%. Since PR-Miner requires rules to have confidences at least 90% (to avoid detecting too many false bugs), it filters out all these potential rules, thus missing this real bug. While it is possible to reduce the confidence requirement, the bug detection precision will likely be too low given that already 40–86% of bugs reported by PR-Miner are false bugs with the confidence 90% [144].

Different from these rule-based bug detection approaches, Bugram does not use programming rules. Instead, it detects potential bugs by reporting method call sequences of low probabilities in a project. We detect the real bug shown in Figure 4.2, because the method call sequence `[isDebugEnabled, debug, indent, stringify]` has a low probability of $2.855 \times 10^{-5}$, which is the 13th lowest probability of all 31,204 sequences of length five using a 3-gram model.

## 4.2 Background

The n-gram language model has been widely used in modelling natural language [39] and solving problems such as speech recognition [20], statistical machine translation, and other related language problems [231]. The n-gram language model typically has two components, words and sentences, where each sentence is an ordered sequence of words. A dictionary $D$ contains all possible words of a language, and each word is represented as $w$. The language model can build a probabilistic distribution over all possible sentences in a language using Markov chains. The probability of a sentence in a language is estimated by generating the sequence word by word. The probability of each word in a sentence is only determined by the conditional probabilities of the previous $n-1$ tokens. Given a sentence $s = w_1 w_2 w_3 \cdots w_m$, its probability is estimated as:

$$P(s) = \prod_{i=1}^{m} P(w_i | h_{i-1}) \tag{4.1}$$

where the sequence $h_i = w_{i-n} \cdots w_i$ is the history. In the n-gram model, the probability of the next word $w_i$ depends only on the previous $n-1$ words. For example, if the sequence length $m$ is four, the probability of the sequence $s = w_1 w_2 w_3 w_4$ using a 4-gram model is:

$$P(s) = P(w_1)P(w_2|w_1)P(w_3|w_1 w_2)P(w_4|w_1 w_2 w_3) \tag{4.2}$$

If we use a 3-gram model, the probability of token $w_4$ depends only on the previous two tokens, and the probability of $s$ is:

$$P(s) = P(w_1)P(w_2|w_1)P(w_3|w_1w_2)P(w_4|w_2w_3) \tag{4.3}$$

In this work, we build n-gram models to learn probabilities of using a method given different contexts. With the learned probability distribution, we further calculate the possibility of each token sequence and flag low probability token sequences as potential bugs.

## 4.3 Approach

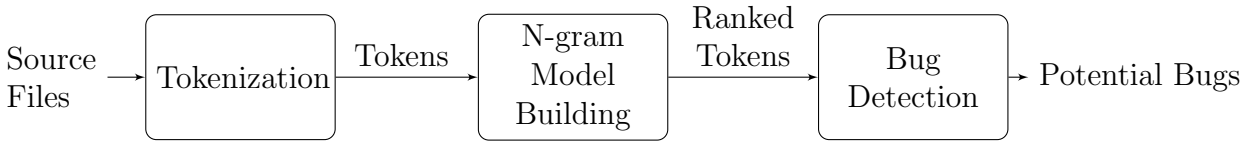



Figure 4.3: Overview of Bugram

Figure 4.3 shows the overview of Bugram. In this section, we first describe how to parse a project to convert it into tokens (Section 4.3.1), and then use the tokens to build n-gram models for the project (Section 4.3.2). Finally, we present how to leverage the n-gram models to detect bugs in the project (Section 4.3.3).

### 4.3.1 Tokenization

To build n-gram models, we need to tokenize the source code of a given project. A main challenge is selecting a suitable level of *granularity* for tokens when building the n-gram models. Existing work builds n-gram models at the syntactic level using low-level tokens to suggest the next tokens for code completion and suggestion [78, 88]. For example, after seeing "`for (int i=0; i<n;`", it suggests the tokens "`i++) {`". Building n-gram models at this level is likely to only detect syntactic errors, e.g., missing "`;`" or "`i++`" in a `for` loop, which will be caught by a compiler.

To detect bugs at the semantic level, we need to build n-gram models at a semantic level. Bugram selects high-level tokens that represent the structure and context of the code using a succinct semantic representation. Take the loop "`for (int i=0; i<n; i++)`

{ `foo(i);` }" as an example. Bugram will represent it with the following high-level tokens `[<FOR>, foo(), <END_FOR>]`.

As reported in existing work [284, 285], control flow information is important for the accuracy of bug detection. Inspired by the above work, during the tokenization process, Bugram also considers the control flow information of source code by adding the control flow elements into the token sequences. Therefore, we focus on method calls and control flow, i.e., method calls, constructors, and initializers; `if/else` branches; `for/do/while/foreach` loops; `break/continue` statements; `try/catch/finally` blocks; `return` statements; `synchronized` blocks; `switch` statements and `case/default` branches, and `assert` statements. A method call `methodA()` is resolved to its fully qualified name `org.example.Foo.methodA()` to prevent unrelated methods with an identical name from being grouped together. In addition, the type of exception in the `catch` clauses are considered as they provide important context information to help us infer more accurate contextual information of method sequences.

Bugram uses the Eclipse JDT Core[1] to tokenize the source files, construct the abstract syntax trees (ASTs), and resolve the type information for the tokens. In this work, we consider both *method* and *control flow* as tokens.

### 4.3.2 N-gram Model Building

In this work, we use n-gram models to learn a probability distribution over token sequences using all extracted sequences. For every sequence extracted from a method we add all its subsequences to the model. For example, given a token sequence $ABC$ extracted from a method, we add all of its ordered subsequences, i.e., $A$, $B$, $C$, $AB$, $BC$, and $ABC$, to the model. Note that we ignore incontinuous subsequences, such as $AC$ in this example.

Smoothing is a common process for n-gram models to help with handling unknown sequences. However, since the entire source code of a project is being used, we have the complete language of all possible sequences. This means smoothing is unnecessary since there are no unknown sequences.

When building n-gram models, one important parameter is `Gram Size` (details are in Section 4.3.3), which defines the length of considered token sequences. N-gram models assume that each token depends only on the previous $n-1$ tokens. To leverage n-gram models to generate probabilities of token sequences, given a specific gram size $n$, we build a set of *internal probabilities*. For example, the probability of a token sequence $ABC$

[1]https://eclipse.org/jdt/core/index.php

calculated by a 3-gram model is $P(ABC) = P(A) \cdot P(B|A) \cdot P(C|AB)$. We refer to $P(A)$, $P(B|A)$, and $P(C|AB)$ as internal probabilities. These internal probabilities can be reused to calculate the probabilities of sequences that shared common subsequences. Thus, we store internal probabilities to cut down the probability calculation time for building n-gram models of different gram sizes. After obtaining all the internal probabilities for an n-gram model, we use them to calculate the probabilities of token sequences.

Previous studies that leverage n-gram models for code completion [88, 225, 271] found that 3-gram, 4-gram, 5-gram, and 6-gram models generated reasonable results. However, the appropriate n-gram size for detecting bugs is unknown. To answer this question, we build n-gram models with gram size from two to ten to study the impact of gram size on the effectiveness of bug detection. The algorithm that we use to build the n-gram model is standard, which is described in Section 4.2.

### 4.3.3 Bug Detection

Bugram detects potential bugs by calculating and ranking the probabilities of all sequences. After obtaining the probabilities of all sequences, Bugram ranks them based on their probabilities in descending order, then reports sequences with the lowest probabilities as potential bugs.

#### Configurations

A few important factors affect the effectiveness of Bugram, i.e., the number of bugs Bugram can find. The four main factors are as follows. Section 4.4.2 describes the setup, tuning, and impact of these parameters.

- **Gram Size** - The size of an n-gram model.
- **Sequence Length** - The length of token sequences to be considered when building n-gram models and detecting bugs.
- **Reporting Size** - The number of sequences, in the bottom of the ranked list, which will be reported as bugs.
- **Minimum Token Occurrence** - The minimum number of times a token must occur in the software to be included in an n-gram model.

**Gram Size** $n$. As described in Section 4.2, the gram size is the size $n$ in an n-gram model. The probability of a sequence is estimated by generating the sequence token by token, and the probability of each token is determined by the conditional probabilities using a history of up to $n-1$ tokens. For example, given a token sequence $S = ABCD$, a 2-gram model considers the probabilities of each two sequential tokens, and will calculate its probability with $P(S) = P(A) \cdot P(B|A) \cdot P(C|B) \cdot P(D|C)$. While a 4-gram model considers the probabilities of each four sequential tokens, and calculates its probability as $P(S) = P(A) \cdot P(B|A) \cdot P(C|AB) \cdot P(D|ABC)$. In this work, we build n-gram models with gram size from two to ten to find an appropriate gram size for detecting bugs.

**Sequence Length** $l$. For building n-gram models, token sequences are extracted from all methods of a project. The length of token sequences extracted from different methods varies, which can be as small as one, and as large as 200. Breaking these long token sequences into many small sequences may help us obtain fine-grained method usage scenarios and detect more bugs. In this work, we evaluate the impact of different sequence lengths on the performance of Bugram.

**Reporting Size** $s$. Different from rule-based bug detection techniques, Bugram detects bugs by identifying token sequences of low absolute probabilities. Thus, an important question is how to set an appropriate threshold to separate sequences that indicate bugs from common sequences that are not bugs. We use this parameter to determine the bottom $s$ sequences in the ranked list and report them as bugs. In general, a larger $s$ allows Bugram to find more bugs at the cost of examining more sequences that potentially indicate bugs. We expect that as the probabilities increase in the ranked list, the percentage of true bugs decreases. An appropriate $s$ should help Bugram find as many bugs without losing much precision of bug detection.

The task of selecting the parameter $s$ in Bugram is the counterpart of selecting a rule probability threshold in rule-based bug detection approaches, such as PR-Miner [144]. As described in Section 1, rule-based approaches select a high-probability rule based on the conditional probability of tokens. For example, if the probability $P(C|AB)$ is higher than the threshold, $\{AB => C\}$ will be selected as a rule, and occurrences of the sequence $AB$ followed by a call other than $C$ is reported as a bug. According to the definition of conditional probability, $P(C|AB) = P(ABC)/P(AB)$, meaning that rule-based techniques consider the probability of the sequence $ABC$ and compare it to the background probability of $AB$. However, when the rule $\{EF => G\}$ is evaluated, the background probability is now that of the sequence $EF$ since the conditional probability is $P(G|EF)$. To achieve optimal performance, rule-based methods should ideally find the individual correct threshold for each background probability. This is of course not practically feasible, which is the reason a single threshold is used in practice. In Bugram, we avoid this problem by directly

```
String q[] = qqf.bestQueries("body",20)
    ;
  for (int i=0; i<q.length; i++) {
        System.out.println(newline+
        formatQueryAsTrecTopic(i,q[i],
            null,null));
}
```

Figure 4.4: The filtering based on `Minimum Token Occurrence` can help Bugram avoid reporting this false bug from the latest version of Lucene.

evaluating the probability of the entire sequence. In this case, it is more theoretically sound to select a single threshold.

**Minimum Token Occurrence** $y$. After performing the tokenization process described in Section 4.3.1, Bugram keeps only tokens with occurrences greater than $y$ in the project. Filtering out uncommon tokens is a standard technique for natural language processing (NLP) techniques [158]. In this work, the rationale is that some methods are generally not well used, or too unique, making them corner-cases that are harder to evaluate on a statistical basis. As such, their inclusion leads to generating false bugs. Take the code snippet in Figure 4.4 as an example. Using a 3-gram model, the sequence `[bestQueries, println, formatQueryAsTrecTopic]` is ranked at the bottom of all token sequences by their probabilities. However, this token sequence is not a bug. It has a low probability because it uses two infrequent private methods `bestQueries` and `formatQueryAsTrecTopic`, each of which is used only once in the whole project.

To avoid reporting the above false bug, Bugram performs a token level filtering. It filters out all tokens that appear fewer than a given $y$. Such process can help Bugram avoid reporting many token sequences with low probabilities that are not bugs.

### Pruning False Bugs

Bugram identifies token sequences with low probabilities as potential bugs. However, some low probability token sequences are unusual/special uses of code and are not bugs, they are false bugs for the purpose of bug detection. To filter out false bugs, we reduce the number of reported bugs (also called *candidate bug set*) by keeping only token sequences at the bottom of at least two ranked lists generated by different n-gram models with different sequence lengths. The rationale is that if a bug can be detected by multiple ranked lists, there is a higher chance that it is a true bug. Remember that given a specific gram size, we generate multiple n-gram models of different sequence lengths ranging from two to ten. Therefore, we only report token sequences that are at the bottom of at least two different n-gram models with the same gram size but different sequence lengths.

For example, if both sequences $ABCDE$ and $BCD$ are ranked at the bottom of the list of 5-token-sequences and 3-token-sequences respectively, Bugram identifies them as an **overlap (two sequences contain a common substring)** and reports $ABCDE$ and $BCD$ as one bug. More formally, we obtain a new candidate bug set using the following formula:

$$C(n,t) = \bigcup_{\forall i,j \in M, i \neq j} (Bottom(n,t,i) \cap Bottom(n,t,j)) \tag{4.4}$$

where $C(n,t)$ is the candidate bug set generated by an *n-gram* model with the reporting size of $t$. $M$ is the set of sequence length, and $i$ and $j$ are two different sequence lengths. $n$ is the gram size. $Bottom(n,t,i)$ is the bottom $t$ token sequences generated by an *n-gram* model with sequence length of $i$. Note that $\cap$ denotes the **overlaps** that are at the bottom of the two different n-gram models, and $\bigcup$ denotes the union of the **overlaps**.

## 4.4 Experimental Study

We evaluate Bugram in terms of the number of detected bugs and detection precision, and explore appropriate parameters for Bugram. All our experiments are conducted on a 4.0GHz i7-3930K desktop with 64GB of memory.

### 4.4.1 Evaluated Software

We evaluate Bugram on 16 widely-used open-source Java projects ranging from 36 thousand lines of code (KLOC) to almost one million lines of code (MLOC). These projects are selected since they are widely used, also we consider their sizes, both small and large projects are selected. Table 4.1 lists their versions, numbers of files, lines of code (LOC), and numbers of methods. We used the latest version of each project. These projects are selected since they are widely used, also we consider their sizes, both small and large projects are selected.

### 4.4.2 Parameter Setting and Sensitivity

To build n-gram models and detect bugs effectively, we need to tune these parameters proposed in Section 4.3.3. We use three widely-used and representative projects from Table 4.1, i.e., Pig, Hadoop, and Solr, to study the impact of different parameters on the

Table 4.1: Projects used to evaluate Bugram

| Project | Version | Files | LOC | Methods |
|---|---|---|---|---|
| Elasticsearch | 1.4 | 3,130 | 272,261 | 28,950 |
| GeoTools | 13-RCI | 9,666 | 996,800 | 89,505 |
| jEdit | 5.2.0 | 543 | 110,744 | 5,548 |
| Proguard | 5.2 | 675 | 69,376 | 5,919 |
| Vuze | 5500 | 3,514 | 586,510 | 37,939 |
| Xalan | 2.7.2 | 907 | 165,248 | 8,965 |
| Hadoop | 2.7.1 | 4,307 | 596,462 | 46,104 |
| Hbase | 1.1.1 | 1,392 | 465,456 | 42,948 |
| Pig | 0.15.0 | 948 | 121,457 | 9,323 |
| Solr-core | 5.2.1 | 1,061 | 146,749 | 9,938 |
| Lucene | 5.2.1 | 2,065 | 293,825 | 18,078 |
| Opennlp | 1.6.0 | 603 | 36,328 | 2,954 |
| Struts | 2.3.24 | 2,022 | 157,499 | 15,254 |
| Zookeeper | 3.5.0 | 492 | 61,708 | 5,034 |
| Nutch | 2.3.1 | 409 | 198,560 | 2,309 |
| Cassandra | 2.2.0 | 1,616 | 280,716 | 15,233 |

performance of Bugram. Specifically, we tune three of the four parameters, i.e., 9 different gram sizes, 9 different sequence lengths, and 5 different reporting sizes. For minimum token occurrence, we remove any token that appears fewer than three times [158]. In total, there are 9*9*5 = 405 possible combinations of the four parameters. Note that, for each combination, we need to manually examine the reported bugs for each project, which is prohibitively expensive. To save efforts, we only pick the three representative projects to tune the four parameters (in total we need to manually examine the reported bugs of 405*3 combinations) in this work. In practice, if users can afford more time, they can tune on more projects to obtain an optimal parameter combination of Bugram to detect bugs more effectively.

**Setting Gram Size**. Different gram sizes enable Bugram to use different internal probabilities to calculate the probabilities of token sequences. We build n-gram models for each project, and the gram size ranges from two to ten. To evaluate the performance of n-gram models of different gram sizes, we calculate the probabilities of all token sequences and rank them based on their probabilities in descending order, then we examine the bottom 10, 20, 30, 50, and 100 sequences from each n-gram model respectively, and manually verify whether a token sequence contains a bug or not.

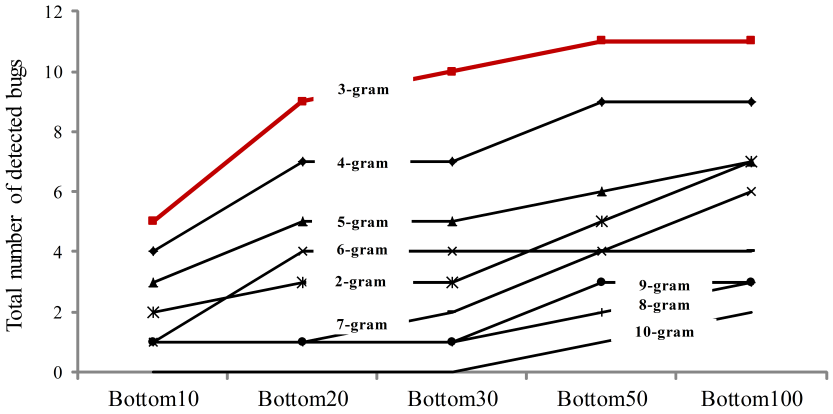


Figure 4.5: Impact of the gram size on the number of true bugs detected

For each n-gram model, we count the number of real bugs detected in the three projects. Figure 4.5 shows the results of the three projects combined. The results show that Bugram finds the most number of true bugs with a 3-gram model. Thus, in this work, we build 3-gram models for Bugram to detect bugs.

**Setting Sequence Length**. As described in Section 4.3.3, we break long sequences extracted from a function into small subsequences. Different sequence lengths enable Bugram to capture different program scenarios and further affect the performance of Bugram. To evaluate the impact of different sequence lengths, we perform Bugram with sequence length ranges from two to ten. For each sequence length, we build a 3-gram model, then calculate probabilities of all sequences. Based on generated probabilities, we rank all sequences. We examine the bottom 50 sequences with low probabilities to check how many true bugs are detected.

Table 4.2 shows the results of detected true bugs in the bottom 50 sequences with different sequence lengths. As we can see, sequence length can significantly affect the performance of Bugram. N-gram models, with sequence length ranges from three to eight, enable Bugram to detect bugs effectively. For example, when the sequence length is equal to five, we find seven true bugs on Hadoop, three on Pig, and one on Solr. When the sequence length is quite low (e.g., two) or quite big (e.g., nine, and ten), Bugram detects no bugs in two of the three examined projects. Thus, in this work, sequence length ranges from three to eight.

**Setting Reporting Size**. In this work, we use this parameter to limit the number of sequences in the bottom of ranked sequence list to be reported as bugs. An appropriate reporting size might help us identify many true bugs with a small number of false positives.

Table 4.2: Detected true bugs in the bottom 50 token sequences with different sequence lengths

| Project | Sequence Length | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** |
| Pig | 0 | 1 | 1 | 3 | 2 | 3 | 1 | 1 | 1 |
| Hadoop | 0 | 3 | 4 | 7 | 3 | 3 | 2 | 0 | 0 |
| Solr | 0 | 1 | 1 | 1 | 2 | 0 | 0 | 0 | 0 |

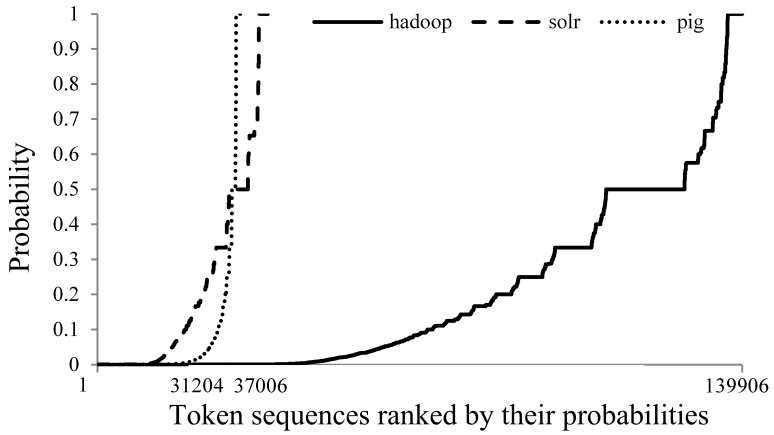
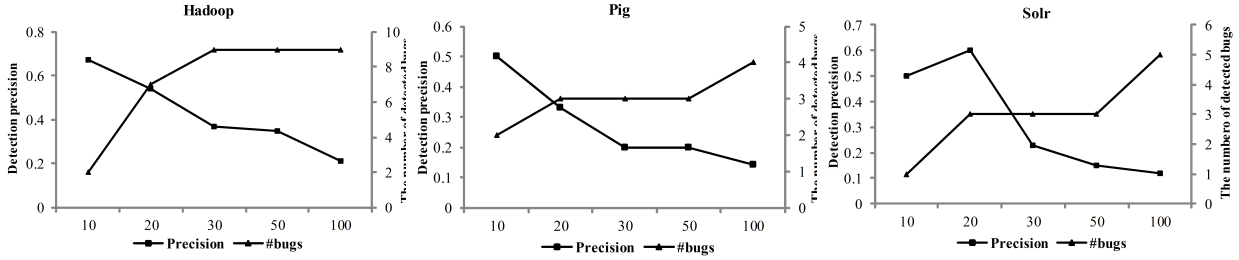



Figure 4.6: Probabilities distribution of all token sequences in Hadoop, Solr, and Pig

For each examined project, we build 3-gram models with sequence length ranges from three to eight.

We first examine the probabilities of all sequences, to explore whether there has a clear cutoff between low probability sequences and high probability sequences. We normalized the probabilities of all sequences in a project, since the range of sequence probability distribution varies in different projects, e.g., in Hadoop the sequence probability range is $[1.0 \times 10^{-11}\ 5.4 \times 10^{-4}]$, while this range for pig is $[2.84 \times 10^{-5}\ 2.7 \times 10^{-3}]$. Figure 4.6 shows the normalized probabilities of all sequences (ranked by probability), the X-axis is the number of sequences in different projects. As we can see, the probability curves are quite smooth at the bottom ten thousand sequences. However, it is prohibitively expensive to examine all these sequences.

Next, we narrow down the reporting size by only looking at the bottom 100 sequences. Specifically, for each project, we examine how many true bugs are detected when the reporting size is equal to 10, 20, 30, 50, and 100, which means we only examine the




(a) **Results of Hadoop** (b) **Results of Pig** (c) **Results of Solr**

Figure 4.7: Detection precision and number of detected bugs in the overlaps of bottom $s$ token sequences with low probability

bottom 10, 20, 30, 50, and 100 sequences in the ranked list. In practice, if developers can afford more time, they can examine more sequences to find more bugs. The number of detected true bugs and detection precisions of the three projects are shown in Figure 4.7a, Figure 4.7b, and Figure 4.7c. As we can see with the increasing of reporting size, the number of detected bugs increases, while corresponding detection precision declines sharply. When the reporting size is equal to 100, the corresponding detection precision is smaller than 20%. In this study, we set reporting size equal to 20, which could enable us to detect 13 true bugs with an average detection precision of 49% on the three examined projects.

**Setting Minimum Token Occurrence**. This parameter is the minimum number of times a token is required to appear in a program to be included in sequences. An appropriate value of this parameter helps filter out token sequences that use unusual/special methods, thus have low probabilities, but are not bugs.

In this study, we remove any token that appears fewer than three times. This is a common practice in NLP research, aimed at improving system performance [158].

### 4.4.3 Comparison with Existing Techniques

We compare Bugram with five existing graph- and rule-based approaches. First, we choose the most closely related work, PR-Miner [144]. While comparing with PR-Miner allows us to compare Bugram with an existing approach as is, we also want to study the sole impact of using n-gram models. In addition to using n-gram models, the differences between Bugram and PR-Miner include (1) Bugram uses control flow information, but PR-Miner does not, and (2) Bugram preserves the token order, while PR-Miner ignores the token order. As discussed in Section 4.3, we use control flow information because it has been shown to be beneficial for bug detection techniques [38, 285]. Therefore, our second approach for comparison is identical to PR-Miner except that it considers both the order of tokens and

control flow information. Since PR-Miner is not publicly available, we have reimplemented our own version of it. We refer to our implementation of PR-Miner as *FIM*, which stands for **F**requent **I**temset **M**ining. We call our implementation of the second approach described above *FSM*, which stands for **F**requent **S**equence **M**ining, because *FSM* mines rules with order preserved, which is what frequent sequence mining does.

To implement PR-Miner, we follow each step described in [144], we first parse the source code and extract variables, method calls, classes in a function. After that, we hash selected elements into numbers. Next, each function is mapped to an itemset. Then, using these itemsets, we perform frequent itemset mining, as provided by Weka [74] to mine frequent itemsets with a specific support and confidence. Finally, these frequent itemsets are treated as rules, and violations of these rules are identified as bugs.

Note that, FIM does not consider the order of tokens. Therefore, given a frequent itemset $ABC$, FIM may generate many rules, e.g., $\{A => BC\}$, $\{B => AC\}$, $\{AB => C\}$, $\{AC => B\}$, $\{C => AB\}$, and $\{BC => A\}$. Any violations of these rules will be reported as potential bugs. While FSM considers the order, thus given the same frequent itemset $ABC$, FSM generates at most two rules, i.e., $\{A => BC\}$ and $\{AB => C\}$. The more rules are inferred, the more potential bugs are likely to be reported. Thus, in practice, both the number of generated rules and the number of reported bugs of FIM are significantly larger than those of FSM.

To reduce false positives, PR-Miner set the support threshold to 15 and the confidence threshold to 90%. With these thresholds, PR-Miner reports many potential bugs, e.g., PR-Miner reported 1,447 potential bugs in the Linux kernel [144]. To save effort, they examined only the top 60 potential bugs ranked by confidence.

These thresholds are only evaluated on three C projects. We find that these thresholds produce poor results on the Java projects used in this work, e.g., we find 0 true bugs in the top 60 ranked bugs in the three Java projects, i.e., Hadoop, Solr, and Pig. Therefore, to set appropriate support and confidence thresholds for FIM and FSM, we have explored FIM and FSM with different combinations of support and confidence on the three projects. For FIM, we find that when the support is equal to seven and the confidence is larger than 75%, it performs the best on the three projects when examining the top 80 sequences ranked by confidence (we have examined up to the top 100, and found the top 80 gives the highest precision and recall). For FSM, when the support is equal to five and the confidence is larger than 85%, it performs the best on the three projects. For a fair comparison, we tune all four parameters of Bugram on the same three projects and apply the best parameters on the rest of the evaluated projects.

Second, we compare Bugram with three graph- and rule-based bug detection approaches, i.e., JADET [285], Tikanga [284], and GrouMiner [199]. These three approaches leverage graph models and frequent itemset mining techniques to mine rules that capture both method order and control flow information to detect bugs. Different from PR-Miner that was evaluated on C projects, all these three approaches were evaluated on open-source Java projects. Thus, Bugram can be applied to these projects directly. Since, two of these three tools, i.e., Tikanga and GrouMiner, are not publicly available, to compare with these three approaches, instead of implementing our own versions, we perform Bugram on the 14 projects evaluated by these three approaches, and compare our detection results with the results from these three studies. In addition, since JADET is an open-source tool, we also apply JADET on projects listed in Table 4.1, and compare its detection results with Bugram.

### 4.4.4 Evaluation Measures

We manually examine the reported potential bugs and categorize the bugs into three types: *True Bugs*, *Refactoring Opportunities*, and *False Positives*. True bugs are faults and can be fixed by altering the code and correcting its behaviour. Refactoring opportunities are bad practices and can be fixed by refactoring the infrequent code snippets to make them more regular. Any reported bugs that do not fit into the above two groups are considered to be false positives. We refer to the number of true bugs and refactoring opportunities as *True Positives*.

To evaluate the performance of a bug detection approach, we use three measures: standard *precision*, *relative recall* [236], and *F1*. Note that we use *relative recall* not standard *recall*, because it is not practical to know all bugs in a project. The precision is: (*True Positive*)/(*Reported Potential Bugs*); To calculate the relative recall, we first define the *Relative Ground Truth* [236] as all the unique true positives reported by Bugram, FIM, and FSM. For each of the three approaches, we calculate its relative recall as: (*True Positive*)/(*Relative Ground Truth*); F1 is the harmonic mean of the precision and relative recall.

### 4.4.5 Manual Examination of Reported Results

Following prior work [30,38,70,144,223,269,284,285], we manually check whether the bugs reported by Bugram are true positives. For manual evaluation, a token sequence was only

considered buggy if both the first author and a non-author graduate student agreed. Given a reported buggy sequence, we consider it a bug if it meets one of the following conditions:

- It has obviously incorrect project specific function calls.
- It violates common API usages, e.g., exception-handling API usages[2] and log-related API usages[3]. For exception handling APIs, we detect several true bugs that violate their usages, e.g., in the true bug in Figure 4.8(b), developers did not handle all potential exceptions that might be thrown by method `waitForCompletion`, which may crash the system if any of these exceptions are thrown. For log-related APIs, one common usage is that the checked log level and the used log level should be the same. Several of our detected true bugs violate this usage, e.g., in the true bug in Figure 4.8(c): before calling the method `info()` to log messages, instead of checking whether the log level `info` is available, developers checked whether `debug` is available, which is incorrect.

We consider a reported buggy sequence a refactoring issue based on principles of code smells proposed in [160], e.g., *duplicated code*—when two code fragments look almost identical. Duplicated code could hinder maintenance because developers need to track and modify each repeating fragment. Developers could refactor the duplicated code by extracting it into a function.

## 4.5 Results and Analysis

This section presents the results of detected bugs (Section 4.5.1 and Section 4.5.2) including the comparison with existing graph- and rule-based techniques, detected bug examples (Section 4.5.3), and the execution time of Bugram (Section 4.5.4).

### 4.5.1 Comparison with FSM and FIM

Table 4.3 shows the number of bugs detected by Bugram, FSM, and FIM on each evaluated project. In total, Bugram reported 59 potential bugs, 42 of which are correct and useful—25 true bugs, and 17 refactoring opportunities. We have reported these true bugs to

[2]https://docs.oracle.com/javase/tutorial/essential/exceptions
[3]https://logging.apache.org/log4j/1.2/manual.html

Table 4.3: Bug detection results. Reported is the number of reported bugs, TBbugs is the number of true bugs, and Refs is the number of refactoring opportunities. We manually inspect all reported bugs except for FIM whose 'Inspected' column shows the number of bugs inspected. Numbers in brackets are the numbers of true bugs detected by Bugram that are detected by neither FIM nor FSM.

| | **Bugram** | | | **FSM** | | | **FIM** | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| **Project** | **Reported** | **TBugs** | **Refs** | **Reported** | **TBugs** | **Refs** | **Reported** | **Inspected** | **TBugs** | **Refs** |
| Elasticsearch | 3 | 1(1) | 0 | 0 | 0 | 0 | 987 | 80 | 0 | 2 |
| GeoTools | 4 | 2(2) | 1 | 4 | 0 | 2 | 1,203 | 80 | 1 | 2 |
| JEdit | 3 | 0 | 1 | 0 | 0 | 0 | 451 | 80 | 0 | 2 |
| Proguard | 1 | 1(1) | 0 | 1 | 1 | 0 | 665 | 80 | 1 | 0 |
| Vuze | 2 | 0 | 2 | 0 | 0 | 0 | 435 | 80 | 0 | 4 |
| Xalan | 3 | 2(2) | 0 | 0 | 0 | 0 | 378 | 80 | 0 | 0 |
| Hadoop | 13 | 7(6) | 4 | 13 | 3 | 0 | 869 | 80 | 2 | 3 |
| Hbase | 1 | 1(1) | 0 | 10 | 2 | 3 | 774 | 80 | 1 | 0 |
| Pig | 9 | 3(2) | 4 | 6 | 0 | 2 | 605 | 80 | 1 | 3 |
| Solr-core | 5 | 3(3) | 1 | 0 | 0 | 0 | 787 | 80 | 0 | 1 |
| Lucene | 2 | 0 | 0 | 10 | 2 | 2 | 676 | 80 | 1 | 0 |
| Opennlp | 6 | 2(2) | 2 | 3 | 0 | 0 | 806 | 80 | 0 | 2 |
| Struts | 5 | 1(1) | 2 | 9 | 1 | 0 | 232 | 80 | 0 | 0 |
| Zookeeper | 0 | 0 | 0 | 3 | 0 | 0 | 442 | 80 | 0 | 1 |
| Nutch | 2 | 2(2) | 0 | 1 | 0 | 1 | 253 | 80 | 1 | 1 |
| Cassandra | 0 | 0 | 0 | 7 | 0 | 1 | 324 | 80 | 0 | 2 |
| **Total** | **59** | **25(23)** | **17** | **67** | **9** | **11** | **9,887** | **1,280** | **8** | **23** |
| **Relative Recall** | **54.5%** | | | **26.0%** | | | **40.3%** | | | |
| **Precision** | **71.2%** | | | **29.9%** | | | **2.4%** | | | |
| **F1** | **61.7%** | | | **27.8%** | | | **4.5%** | | | |

developers, 7 of which have already been confirmed by developers, while the rest await confirmation. The results suggest that Bugram is effective in finding real bugs in widely-used mature software projects to improve software reliability.

As described in Section 4.4.4, the relative recall shows the ability of a technique in finding new bugs, while the precision indicates the ability of a technique in avoiding reporting false bugs. F1 is the harmonic mean of the precision and recall. The relative recall of Bugram is 54.5%, which is higher than FIM's relative recall of 40.3% and FSM's relative recall of 26.0%. The detection precision of Bugram is 71.2%, which is higher than FIM's precision of 2.4% and FSM's precision of 29.9%. The F1 of Bugram is 61.7%, again higher than FIM's F1 of 4.5% and FSM's F1 of 27.8%. The results suggest that Bugram can find more bugs than examined rule-based approaches and is more precise than them, suggesting Bugram complements existing rule-based bug detection techniques.

In addition, as shown in Table 4.3, among the 25 true bugs detected by Bugram, only two can also be detected by FIM and FSM. The majority (23) of the true bugs can only

be detected by Bugram, showing that Bugram can find real bugs in real-world software that examined rule-based approaches cannot find. In comparison, FIM detected eight true bugs, and 23 refactoring opportunities. FSM reported 67 potential bugs, and nine of which are true bugs, and 11 are refactoring opportunities. Since six true bugs are detected by both FIM and FSM, a total of 11 unique true bugs are detected by these two approaches, nine of which cannot be detected by Bugram. In total, there are 77 unique true positives generated by the three approaches.

Since FSM considers both the control flow and order of tokens, both the numbers of rules and bugs discovered by FSM are much smaller than those of FIM. Table 4.3 shows that FIM reported a total of 9,887 potential bugs, while FSM only reported 67 potential bugs.

As we described in Section 1, Bugram and FIM detected bugs based on different probability distributions of token sequences. Bugram identifies token sequences with absolute low probability as bugs, while FIM and FSM identify token sequences with relatively low probability as bugs. A relatively low probability token sequence might have a high absolute probability with a high rank in all token sequences extracted from a project. Thus, **our results suggest that Bugram and rule-based bug detection techniques complement each other to detect more bugs.**

### 4.5.2 Comparison with JADET, Tikanga, and GrouMiner

As described in Section 4.4.3, we also compare Bugram with three graph- and rule-based bug detection approaches, i.e., JADET [285], Tikanga [284], and GrouMiner [199]. These approaches have been evaluated on Java projects, and their authors have presented the number of detected potential bugs and manually identified true bugs. Since JADET, Tikanga, and GrouMiner each reported many potential bugs for the evaluated projects, to save effort, the authors of the three tools manually verified a subset of reported potential bugs, i.e., top 10 in JADET, top 25% in Tikanga, and top 15 in GrouMiner. For a fair comparison, we apply Bugram on the projects that are evaluated by these approaches, and use the same Bugram parameters that are used in the comparison with PR-Miner, meaning that Bugram parameters are not tuned for these projects. JADET was evaluated on five projects, Tikanga was evaluated on six projects, and GrouMiner was evaluated on nine projects. We exclude four projects which are not publicly available anymore. In total, 16 projects (14 unique) are available (Table 4.4).

Table 4.4 shows the detection results of Bugram and these three bug detection approaches. JADET detected three true bugs on the three projects. Bugram detected five

Table 4.4: Comparison with JADET, Tikanga, and GrouMiner. 'Fixed' denotes the number of true bugs detected by Bugram that have already been fixed in later versions. * denotes the number of unique true bugs detected by Bugram that the tools in comparison failed to detect.

| | **Graph-based tools** | **Bugram** | | |
|---|---|---|---|---|
| **Project** | **JADET** | **Reported** | **TBugs** | **Fixed** |
| AZUREUS 2.5.0 | 1 | 8 | 4 | 4 |
| columba-1.2 | 0 | 4 | **1*** | 1 |
| aspectj-1.5.3 | 2 | 5 | 0 | 0 |
| | **3** | **17** | **5** | **5** |
| Project | **Tikanga** | **Bugram** | | |
| aspectj-1.5.3 | 9 | 5 | 0 | 0 |
| tomcat-6.0.18 | 0 | 13 | **4*** | 2 |
| argouml-0.26 | 1 | 3 | 1 | 1 |
| Vuze_3.1.1.0 | 0 | 8 | 0 | 0 |
| columba-1.4 | 1 | 6 | 1 | 1 |
| | **11** | **35** | **6** | **4** |
| Project | **GrouMiner** | **Bugram** | | |
| columba-1.4 | 1 | 6 | 1 | 1 |
| ant-1.7.1 | 1 | 3 | 1 | 1 |
| log4j-1.2.15 | 0 | 7 | **2*** | 2 |
| aspectjrt-1.6.3 | 1 | 12 | 2 | 1 |
| axis-1.1 | 0 | 6 | **3*** | 1 |
| jedit-3.0 | 1 | 5 | 1 | 1 |
| jigsaw-2.0.5 | 1 | 1 | 1 | 1 |
| struts-1.2.6 | 0 | 8 | 0 | 0 |
| | **5** | **48** | **11** | **8** |

true bugs on the same projects, at least one of which cannot be detected by JADET. Since these papers did not report the full list of detected bugs, we do not know if the bugs detected by JADET and Bugram overlap. However, since JADET detected 0 true bugs in `columba`, while Bugram detected one bug, we know that Bugram detected one bug that JADET cannot detect. We also found that all five true bugs detected by Bugram are fixed in a later version by developers. For the same reason as above, we do not know if the bugs detected by JADET have been fixed in a later version. Tikanga detected 11 true bugs on the five projects, while Bugram detected six true bugs on the same projects, four of which are unique to Bugram (detected in `tomcat`). Four of the six true bugs have already been fixed. GrouMiner detected five true bugs on the eight projects, while Bugram detected 11

true bugs on the same projects, five of which are unique to Bugram (detected in `log4j` and `axis`). Eight of the 11 true bugs have already been fixed. In total, the three approaches detected 19 true bugs in the 14 projects, while Bugram detected 21 true bugs, and we have manually checked that 16 of them have already been fixed in a later version. In addition, at least 10 of the 21 true bugs cannot be detected by these three tools.

Since JADET is an open-source tool, we further apply JADET (with recommended parameters) on the projects listed in Table 4.1. Results shown that JADET did not detect any true bugs, the top 10 potential bugs detected by JADET are missing method calls of JAVA library classes, e.g., `Map`, `List`, `Iterator`, etc. JADET, Tikanga, and GrouMiner are based on object usage graph models, so rules generated by them are method usages of classes (both library classes and project specified classes). For example, one of JADET's representative rules is the method `Iterator.next()` should always follow the method `Iterator.hasNext()`. Violations of this rule will be flagged as potential bugs. However, it is not necessary to use both the two methods in every scenario. Thus, it is possible for JADET to report large numbers of false positives related to the Java library classes listed above. The comparison results show that **Bugram is complementary to these graph- and rule-based approaches.**

In addition, the detection precision of Bugram is better than these three techniques. JADET [285] reported that three of the 30 potential bugs detected in the three projects listed in Table 4.4 are true bugs. Thus, JADET has a detection precision of 10.0%, while Bugram achieves a precision of 29.4% on the same projects. Tikanga's authors manually examined 118 potential bugs on the five projects, 11 of which are true bugs [284], indicating a detection precision of 9.3%. While Bugram achieves a precision of 17.1% on the same five projects. Similarly, GrouMiner has a detection precision of 4.2% on the eight projects [199], while Bugram achieves a precision of 22.9% on the same projects.

### 4.5.3 Examples

**Example Bugs**. We show some of the detected true bugs in Figure 4.8. Specifically, Figure 4.8 shows three examples of detected true bugs in ProGuard and Nutch that are detected by our tool. FIM and FSM fail to detect them. We reported the three bugs to the ProGuard and Nutch developers, and all of them have been confirmed as true bugs. In addition, the bugs in Figure 4.8(a) and Figure 4.8(c) have already been fixed by developers.

The bug in Figure 4.8(a) is caused by an incorrect API usage. Specifically, the instantiation of `ConfigurationWriter (writer)` should be closed in a `finally` block instead of a `try` block. To fix this bug, instead of closing the `writer` in the try block, developers added

(a) **A confirmed and fixed true bug from ProGuard (BugID: 582).**

```
try{
ConfigurationWriter writer = new ConfigurationWriter(file);
writer.write(getProGuardConfiguration());
writer.close();}
catch (Exception ex){...}}
```

(b) **A confirmed true bug from Nutch (BugID: NUTCH-2076).**

```
try {
    currentJob.waitForCompletion(true);
    } finally { ...
      }}...}
```

(c) **A confirmed and fixed true bug from Nutch (BugID: NUTCH-2256).**

```
if (LOG.isDebugEnabled()) {
LOG.info("Crawl␣delay␣for␣queue:␣"...);}
```

Figure 4.8: True bug examples from version 5.2 of ProGuard (a) and version 2.3.1 of Nutch (b and c)

a `finally` block, and closed the `writer` in it. Figure 4.8(b) also shows a true bug. The method `waitForCompletion` might throw several exceptions, e.g., `ClassNotFoundException` and `IOException`, while in this case, developers used this function without handling these potential exceptions. To fix this bug, developers should either use a catch block to handle the exceptions or raise the exceptions to be handled by the calling functions. Figure 4.8(c) shows another true bug. This bug is caused by the inconsistency between the checked log level (debug) and the used log level (info). To fix this bug, developers replaced the method `info()` with the method `debug()` to make it consistent with the checked log level. **Example Refactoring Opportunities**. Figure 4.9 shows an example of refactoring opportunity detected by our tool from Pig project. In this example, developers first define two variables `dt1`, and `dt2` to keep data via method call `get()` of objects `bb1` and `bb2`. Next, in the `switch` block, one of the `case` branch needs data from objects `bb1` and `bb2`, while such data already kept in variables `dt1` and `dt2`. Without reusing these two variables, developers call `get()` of objects `bb1` and `bb2` to obtain data again. This costs extra memory and time and should be refactored by using variables `dt1` and `dt2` directly in lines 6 and 7.

Some of our detected bugs may appear to be simple, **but many (7) of them have been confirmed or fixed (4 confirmed and fixed, 2 confirmed with patches proposed, 1 confirmed) by the developers of these projects, suggesting the value of our approach.**

```
byte dt1 = bb1.get();
byte dt2 = bb2.get();
 switch (dt1) {
 case BinInterSedes.BIGINTEGER:{
 if{
   int sz1 = readSize(bb1, bb1.get());
   int sz2 = readSize(bb2, bb2.get());} }
   ...}
```

Figure 4.9: A refactoring bug example from version 0.15.0 of Pig

Table 4.5: Time cost of `Bugram` (in seconds).

| **Project** | **Total** | **Tokenization** | **Model Building and Bug Detection** |
|---|---|---|---|
| Elasticsearch | 162 | 160 | 2 |
| GeoTools | 878 | 872 | 6 |
| jEdit | 86 | 85 | 1 |
| Proguard | 61 | 60 | 1 |
| Vuze | 312 | 310 | 2 |
| Xalan | 80 | 79 | 1 |
| Hadoop | 447 | 443 | 4 |
| Hbase | 151 | 149 | 2 |
| Pig | 93 | 91 | 2 |
| Solr-core | 97 | 95 | 2 |
| Lucene | 206 | 203 | 3 |
| Opennlp | 50 | 49 | 1 |
| Struts | 223 | 220 | 3 |
| Zookeeper | 42 | 41 | 1 |
| Nutch | 36 | 35 | 1 |
| Cassandra | 157 | 155 | 2 |

### 4.5.4 Execution Time and Space

We collect the time and space costs for all the 16 projects listed in Table 4.1, and details are presented in Table 4.5. We can see that the total execution time for tokenization, model building, and bug detection varies from 50 to 878 seconds. Our largest evaluated project, GeoTools, uses 2GB of memory. As shown in the table, most of the time is spent building ASTs with type information with a fraction of the time on building n-gram models and detecting bugs. The results demonstrate Bugram's practical value.

## 4.6 Threats to Validity

**Implementation of PR-Miner** To compare Bugram with rule-based bug detection approaches, we have reimplemented a rule-based approach PR-Miner [144], since PR-Miner is not publicly available. The PR-Miner paper reported higher precision than what we reported with our implementation of PR-Miner in this study. One possible reason is that PR-Miner has only been evaluated on C projects. Its false positive pruning approach, recommended threshold values of support and confidence may only be effective on C projects, while in this study we evaluate on Java projects. However, for our implementation of PR-Miner, we have tried our best to tune parameters, e.g., threshold values of support and confidence, to obtain the best results. This is our best effort given that PR-Miner is not publicly available. Our comparison is fair since both Bugram and PR-Miner are tuned and evaluated on the same projects.

**Bugs are verified by the authors** Following prior work [30, 70, 144, 223, 269, 284, 285], we manually check whether the potential bugs reported by the tools are true positives. Although this approach is a common practice, this process contains bias since we are not the developers of these projects. We mitigate this threat by sending the bugs to developers for further confirmation.

## 4.7 Summary

In this chapter we introduce Bugram, that leverages n-gram models to detect bugs. Bugram detects potential bugs via calculating and ranking the probabilities of program tokens based on the probability distribution of program tokens in a project. Low probability token sequences are flagged as potential bugs. We evaluate Bugram in two ways. First, we compare it with two rule-based bug detection approaches on 16 projects. Results show that Bugram detects 25 true bugs, and 23 of which cannot be detected by PR-Miner. Second, we further apply Bugram on 14 projects evaluated in three graph- and rule-based tools, i.e., JADET, Tikanga, and GrouMiner. Bugram detects 21 true bugs, at least 10 of which cannot be detected by these three tools. Our results suggest that Bugram is complementary to existing rule-based bug detection approaches.

# Chapter 5

# Leveraging Machine Learning Classifiers to Improve Regression Testing

This chapter presents our new test case prioritization technique, i.e., `QTEP`, which leverages machine learning models to evaluate source code quality and then adapts existing test case prioritization algorithms by considering the weighted source code quality. `QTEP` was presented at the 11th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE'17 [281]).

## 5.1 Motivation

Modern software constantly evolves as developers make source code changes such as fixing bugs, adding new features, refactoring existing code, etc. To ensure that the changes do not introduce new bugs, regression testing is commonly conducted against existing functionalities. However, regression testing can be expensive. Especially for large projects, the regression testing could consume 80% of the overall testing budgets and require weeks to run all test suites [37, 59, 234]. Intuitively, test cases that could reveal bugs should be run earlier so that the developers could have more time to fix the revealed bugs and speed up the system delivery. Along this line, test case prioritization (TCP) has been proposed and intensively studied for regression testing [36, 58, 142, 147, 155, 166, 173, 174, 234, 235, 247, 270, 312, 315]. TCP techniques reorder test cases to maximize a certain objective function,

typically exploring faults earlier [234, 314]. Researchers have applied TCP techniques on projects from Google [59, 147, 314] and Salesforce.com [34] and have shown that TCPs could significantly improve the efficiency of regression testing.

Most of existing TCPs are coverage-based [43,86,101,102,142,153,154,166,234,312,319]. A typical coverage-based TCP technique leverages coverage information between source code and test cases, i.e., static code coverage (static call graph from current version) and dynamic code coverage (dynamic call graph from the last execution), to schedule test cases by maximizing code coverage with different coverage criteria. Coverage-based TCPs assign higher priorities to test cases that have higher dynamic or static code coverages.

Existing TCP techniques often do not take the likely distribution of faults in source code into consideration. In other words, they assume that faults in program source code are equally distributed. However, as reported in existing work [203, 205, 256], the fault distribution in source code is often unbalanced, i.e., around 80% faults are located in about 20% source code [203]. Intuitively, test cases that cover the more fault-prone code are more likely to reveal bugs so that they should be run with a higher priority.

The goal of this study is to propose a quality-aware TCP technique, `QTEP`, that addresses the above limitation of existing TCPs. We evaluate the quality of source code in terms of fault-proneness and then we further use the quality information to prioritize test cases. To achieve the goal, `QTEP` gives more weight to fault-prone source code so test cases that cover the fault-prone code have a higher priority to be executed. We identify fault-prone source code in a software project by using defect prediction models, which is widely used to help developers find bugs [125, 221].

In addition, we apply `QTEP` to both regression and new test cases. Most existing TCP techniques only focus on prioritizing regression test cases [43,101,102,142,166,234,312,319]. However, in real-world testing practice, the test suite for a modified software system often consists of: (1) existing test cases, i.e., regression test cases, which are designed to verify whether the existing functionalities still perform correctly after changed, and (2) new test cases, which are added to test the modification. During software evolution, these two types of test cases are essential for testing the modified software and detecting bugs [153]. Thus, we consider both the two types of test cases in this study.

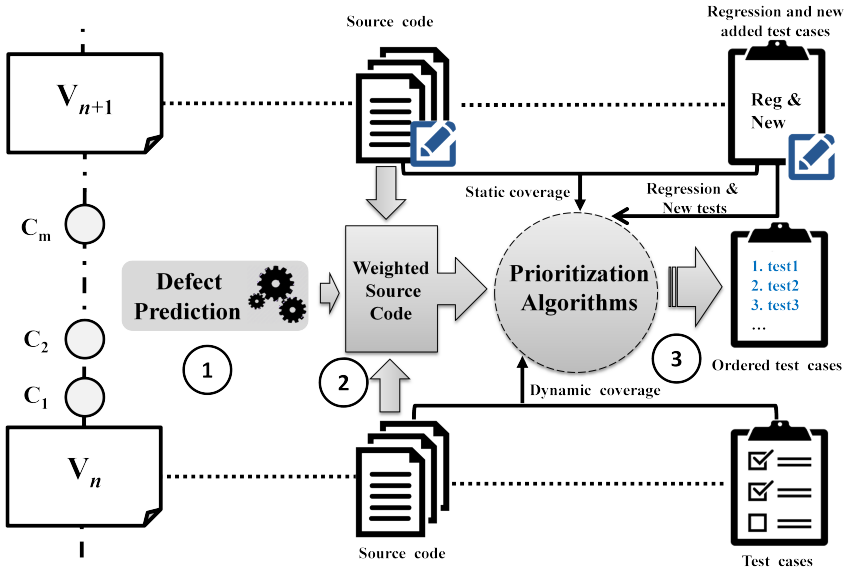



Figure 5.1: Overview of the proposed QTEP. $V_n$ and $V_{n+1}$ are two versions. $C_1$ to $C_m$ are consecutive commits between these two versions that introduce changes to the source code and test cases.

## 5.2 Approach

Figure 5.1 illustrates that our approach consists of three steps: (1) leveraging the defect prediction model based code inspection technique to detect fault-prone source code (Section 5.2.1), (2) weighting source code based on results from the code inspection approach (Section 5.2.2), and (3) adapting existing TCP techniques and evaluating the results (Section 5.2.3).

### 5.2.1 Fault-prone Code Inspection with Defect Prediction Model

Software defect prediction techniques (DP) leverage various software metrics to build machine learning models to predict unknown defects in the source code [82, 104, 170, 190, 192, 256, 280, 289, 318, 326]. Based on the prediction results, software quality assurance teams can focus on the most defective parts of their projects in advance.

Typically, defect prediction models could be categorized as supervised or unsupervised models. Most of existing defect prediction models are supervised [82, 104, 170, 190, 256,

280, 289, 326]. These models leverage past defects from software historical data to build machine learning classifiers and then use the classifiers to predict future bugs. However, not all projects have enough defect data to build a defect prediction model, so unsupervised models [192,318] are also proposed based on the characteristics of defect prediction metrics.

To directly compare QTEP to existing TCP techniques, we reuse 33 versions from 7 open source Java projects from previous TCP studies [55,166,235]. Some of these projects do not have well-maintained past defects. This means we do not have enough defect data to build supervised models. Thus, we adopt the state-of-the-art unsupervised defect prediction model, i.e., CLAMI [192], to detect fault-prone source code in this study. Specifically, to consider different test case prioritization strategies and scenarios, we build CLAMI models at both method and class levels and CLAMI directly outputs the lists of predicted bugs at both levels.

### 5.2.2 Weighting Source Code Units

We leverage the detection results from CLAMI to weight the fault-prone source code units. A code unit could be a statement, a branch, a method, or a class, which depends on different test case prioritization strategies. In this work, we focus on weighting statement-level and method-level code units since we use statement-level and method-level coverage criteria to examine coverage-based TCPs (Section 2.3).

Algorithm 1 shows how to weight statement-level and method-level code units by using detection results (i.e., detected buggy classes and methods) from CLAMI. Initially, the algorithm assigns a default weight to all code units (i.e., all statements and methods). Then, given a code unit, if the class or the method that contains this code unit is detected as buggy, the weight of this code unit will be calculated by accumulating the weights of the buggy class or the buggy method. Otherwise, if the class or the method that contains this code unit is identified as clean, its weight will not be updated. Code units that are not covered by any buggy class or method will be assigned the default weight.

**Parameters**: In the above algorithm, there are three parameters, i.e., *weight_base*, *weight_c*, and *weight_m* that could affect the effectiveness of the proposed QTEP. We describe the setup, tuning, and impact of these three parameters in Section 5.3.6.

- *weight_base* is the base weight for all code units, i.e., the default weight for initializing the weights of all code units.

**Algorithm 1** Weighting source code units

**Input:** Inspection results at class level $C_{Buggy}$ and at method level $M_{Buggy}$. The sets of all methods $M$ and all statements $S$ to be weighted. Parameters $weight\_base$, $weight\_c$, and $weight\_m$.

**Output:** Weighted method set $MWeighted$ and statement set $SWeighted$.

```
1: //Initialization, set default values for examined code units.
2: for each unweighted method MWeighted_i in MWeighted do
3:     MWeighted_i = weight_base;
4: end for
5: for each unweighted statement SWeighted_i in SWeighted do
6:     SWeighted_i = weight_base;
7: end for
8: //Weight methods
9: for each unweighted method M_i in M do
10:    if M_i in M_Buggy then
11:        MWeighted_i += weight_m;
12:    end if
13:    if C_Buggy contains M_i then
14:        MWeighted_i += weight_c;
15:    end if
16: end for
17: //Weight statements
18: for each unweighted statement S_i in S do
19:    if M_Buggy contains S_i then
20:        SWeighted_i += weight_m;
21:    end if
22:    if C_Buggy contains S_i then
23:        SWeighted_i += weight_c;
24:    end if
25: end for
```

- $weight\_c$ is the weight for detected buggy classes by code inspection techniques.
- $weight\_m$ is the weight for detected buggy methods by code inspection techniques.

For CLAMI, we use the class-level prediction results as seeds to weight code units using $weight\_c$, and use the method-level prediction results as seeds to weight code units using $weight\_m$.

### 5.2.3 Quality-Aware Test Case Prioritization

After weighting all the source code units of a project, we then adapt existing coverage-based TCP techniques using these weighted code units. Comparing to existing coverage-based TCPs, `QTEP` leverages the quality-aware coverage information of test cases. As we described in Section 2.3, `QTEP` explores two different types of coverages, i.e., method coverage and statement coverage. In this section, we show how to calculate the quality-aware statement and method coverages of a test case.

A project $P$ has $m$ method-level code units, i.e., $\{mc_1, mc_2, ..., mc_m\}$, and $s$ statement-level code units, i.e., $\{sc_1, sc_2, ..., sc_s\}$. Its test suite $T$ consists of $n$ test cases, i.e., $\{t_1, t_2, ..., t_n\}$. $MWeighted$ ($\{mw_1, mw_2, ..., mw_m\}$) and $SWeighted$ ($\{sw_1, sw_2, ..., sw_s\}$) are the weight sets for the method-level and the statement-level code units respectively.

Given a test case $t_i$ ($1 \leq i \leq n$), we use $QMCoverage[t_i]$ and $QSCoverage[t_i]$ to denote its quality-aware method coverage and statement coverage respectively.

$$QMCoverage[t_i] = \sum_{j=1}^{m} cover(t_i, mc_j) * mw_j \tag{5.1}$$

$$QSCoverage[t_i] = \sum_{j=1}^{s} cover(t_i, sc_j) * sw_j \tag{5.2}$$

where, $cover(t_i, mc_j)$ or $cover(t_i, sc_j)$ is 1, if test case $t_i$ covers code unit $mc_j$ or $sc_j$, otherwise 0. $mw_j$ and $sw_j$ are the weights for method-level code unit $mc_j$ and statement-level code unit $sc_j$ respectively.

Note that, to calculate the quality-aware coverages for test cases, one could leverage different coverage information, i.e., dynamic coverage and static coverage information. With the quality-aware coverages (i.e., $QMCoverage$ and $QSCoverage$) of each test case, `QTEP` further prioritizes test cases with different prioritization strategies (i.e., total and additional).

## 5.3 Experimental Study

### 5.3.1 Research Questions

We answer the following research questions to evaluate the performance of `QTEP`.

**RQ1**. Is `QTEP` more effective than the state-of-the-art coverage-based TCPs for regression test cases only?

**RQ2**. Is `QTEP` more effective than the state-of-the-art coverage-based TCPs for all test cases (both regression and new test cases)?

**RQ3**. How much time can `QTEP` save for finding bugs comparing to existing TCPs?

Table 5.1: Experimental subject programs. **VPair** denotes a version pair. **RTC**, **RTM**, and **RF** are the number of regression test classes, regression test methods, and regression faults respectively. **NTC**, **NTM**, and **NF** are the number of new test classes, new test methods, and mutation faults for new test cases respectively.

| No. | Project | VPair | #RTC | #RTM | #RF | #NTC | #NTM | #NF |
|---|---|---|---|---|---|---|---|---|
| $P_1$ | Time&Money | 3.0-4.0 | 15 | 143 | 1 | 7 | 32 | 1*100 |
| $P_2$ | Time&Money | 4.0-5.0 | 16 | 159 | 1 | 8 | 24 | 1*100 |
| $P_3$ | Mime4J | 0.50-0.60 | 24 | 120 | 3 | 21 | 139 | 3*100 |
| $P_4$ | Mime4J | 0.61-0.68 | 57 | 348 | 4 | 6 | 72 | 4*100 |
| $P_5$ | Jaxen | 1.0b7-1.0b9 | 12 | 24 | 3 | 0 | 0 | - |
| $P_6$ | Jaxen | 1.1b6-1.1b7 | 41 | 243 | 1 | 28 | 250 | 1*100 |
| $P_7$ | Jaxen | 1.1b9-1.1b11 | 69 | 645 | 1 | 7 | 29 | 1*100 |
| $P_8$ | Xml-Security | 1.0-1.1 | 15 | 91 | 2 | 3 | 29 | 2*100 |
| $P_9$ | XStream | 1.20-1.21 | 115 | 637 | 1 | 8 | 38 | 1*100 |
| $P_{10}$ | XStream | 1.21-1.22 | 124 | 698 | 2 | 7 | 58 | 2*100 |
| $P_{11}$ | XStream | 1.22-1.30 | 133 | 768 | 11 | 19 | 134 | 11*100 |
| $P_{12}$ | XStream | 1.30-1.31 | 150 | 885 | 3 | 9 | 76 | 3*100 |
| $P_{13}$ | XStream | 1.31-1.40 | 140 | 924 | 7 | 18 | 180 | 7*100 |
| $P_{14}$ | XStream | 1.41-1.42 | 157 | 1,200 | 5 | 3 | 23 | 5*100 |
| $P_{15}$ | Commons-Lang | 3.02-3.03 | 83 | 1,698 | 1 | 7 | 122 | 1*100 |
| $P_{16}$ | Commons-Lang | 3.03-3.04 | 83 | 1,703 | 2 | 13 | 119 | 2*100 |
| $P_{17}$ | Joda-Time | 0.90-0.95 | 10 | 219 | 2 | 1 | 43 | 2*100 |
| $P_{18}$ | Joda-Time | 0.98-0.99 | 71 | 1,932 | 2 | 9 | 211 | 2*100 |
| $P_{19}$ | Joda-Time | 1.10-1.20 | 90 | 2,420 | 1 | 3 | 415 | 1*100 |
| $P_{20}$ | Joda-Time | 1.20-1.30 | 93 | 2,516 | 3 | 11 | 532 | 3*100 |

### 5.3.2 Supporting Tools

In this study, we focus on coverage-based TCPs, which require both dynamic and static code coverage information of test cases. To collect dynamic code coverage, following existing work [153, 154, 235], we use the ASM bytecode manipulation and analysis framework under FaultTracer tool [320] to collect the dynamic code coverage information for test cases. To collect static code coverage, following existing work [154, 166], we use the WALA framework [7] to collect the static call graphs for the test cases, and traverse the call graphs to obtain the involved methods and statements for each test method and test class. For the unsupervised defect prediction model, i.e., CLAIM, we use its publicly available implementation[1].

All the test prioritization techniques have been implemented in Java and all the experiments were carried out on a 4.0GHz i5-2400 desktop with 6GB of memory.

[1] `https://github.com/lifove/CLAMI/`

### 5.3.3 Subject Systems, Test Cases, and Faults

To facilitate the replication and verification of our experiments, we choose 33 versions from 7 open-source Java projects, which are widely utilized as benchmarks to address real-world test case prioritization problem [55,166,235]. Table 5.1 lists all the projects and the detailed statistical information. The sizes of these systems vary from 5.7K LOC (Time&Money) to 114.1K LOC (Joda-Time).

For regression test cases, following existing work [166, 235], for each listed version-pair, we use the real-world regression faults for regression test cases. Each version-pair has at least one real-world regression fault, which will crash at least one regression test case on the later version. For example, there are 11 regression faults (#RF) in the project $P_{11}$ in Table 5.1.

Since not all benchmark projects have faults for new test cases, following existing work [12, 54, 86, 112, 153], we use mutation faults when considering the new test cases. We generate mutation faults using a set of carefully selected mutation operators [13], e.g., logical, arithmetic, statement deletion, etc. Specifically, we use the *Major* mutation tool [6] to generate these mutation faults for new test cases. Note that not all generated mutation faults can be revealed by test cases, thus we use a subset of detected faults obtained by further running *Major* with all test cases. For each project, we randomly select $m$ mutation faults killed by new test cases only. We set $m$ to be equal to the number of regression faults to simulate the real-world testing scenario. To mitigate the randomness, we repeat this process 100 times. For instance, we randomly select 11 mutation faults and repeat this 100 times as 11*100 (#NF of $P_{11}$ in Table 5.1). Thus, we have 100 fault version-pairs for each of experimental subjects when considering new test cases.

### 5.3.4 Evaluation Measure

We use the Average Percentage Fault Detected (APFD) [233], a widely used metric for evaluating the performance of TCP techniques. APFD measures the average percentage of faults detected over the life of a test suite, and is defined by the following formula:

$$APFD = 1 - \frac{\sum_{i=1}^{num_f} TF_i}{num_t \times num_f} + \frac{1}{2 \times num_t} \tag{5.3}$$

where, $num_t$ denotes the total number of test cases, $num_f$ denotes the total number of detected faults, and $TF_i$ ($1 \leq i \leq num_f$) denotes the smallest number of test cases in sequence that need to be run in order to expose the fault $i$. APFD values range from 0 to

Table 5.2: The experimental scenarios for TCPs.

| Test Granularities | Test Case Types |
|---|---|
| Method (M) | Regression test cases (R) |
| Class (C) | Regression+ New test cases (RN) |

Table 5.3: The experimental independent variables of QTEP.

| IV1: Coverage Techniques | IV2: Coverage Criteria | IV3: Prioritization Strategies |
|---|---|---|
| Dynamic (D) | Method (M) | Total (T) |
| Static-JUPTA (J) | Statement (S) | Additional (A) |

1. For any given test suite, its $num_t$ and $num_f$ are fixed. The higher APFD value signals that the average value of $TF_i$ is lower and thus represents a higher fault-detection rate.

### 5.3.5 Experimental Scenarios

Table 5.2 shows the TCP scenario options in our experiments. By combining these options, **f**our different TCP scenarios can be defined. We first conduct TCP at two different granularity levels, i.e., method (M) and class (C). In addition, following existing work [153], we also conduct TCP for (1) regression test cases only (R), and (2) all test cases (regression and newly added test cases, RN). Running regression test cases only or running all the test cases are two practical testing activities during software evolution [188]. Based on the combinations of these settings, the four scenarios are defined as follows: *regression test method* (M-R), *regression test class* (C-R), *all test method* (*regression and newly added*) (M-RN), and *all test class* (*regression and newly added*) (C-RN). In Section 5.4, we report APFD values for these four scenarios.

Table 5.3 shows independent variables (IVs) used in the experiments. Before conducting TCP experiments under the three scenarios, we set four IVs that affect TCP performance in terms of APFD:

**IV1: Coverage Techniques**. For examining the TCP performance of `QTEP`, we use two representative coverage techniques from the existing coverage-based TCP techniques.

- **Dynamic-coverage-based TCP** is based on the information of the dynamic call graph from the latest run of a subject project. We use test coverage information based on the dynamic call graph to prioritize test cases.

- **Static-coverage-based TCP** ranks test cases based on the information from static call graph. JUPTA is the state-of-the-art static-coverage-based TCP technique [166]. We use JUPTA as a representative static-coverage-based TCP technique for the experiments.

**IV2: Coverage Criteria**. Since all the studied techniques rely on code coverage information, we also investigate the influence of coverage criteria. We study two widely used coverage criteria: (1) *Method* coverage, (2) *Statement* coverage.

**IV3: Prioritization Strategies**. As we described in Section 2.3, the *Total* strategy and *Additional* strategy are widely used in most existing studies to schedule the execution order of test cases [15, 153, 166, 233, 235]. Thus, we also investigate the influence of these two different prioritization strategies.

We can form 8 combinations from the above three IVs (IV1 to IV3) from the existing TCP techniques as baselines. The IV1, IV2, and IV3 in Table 5.3 represent technical options that we can select from the existing coverage-based TCP techniques. Based on acronyms for IV options in Table 5.3, we can list the 8 combinations as follows: `DMT` (i.e., **d**ynamic **m**ethod coverage with **t**otal strategy), `DMA`, `DST`, `DSA`, `JMT`, `JMA`, `JST`, and `JSA`.

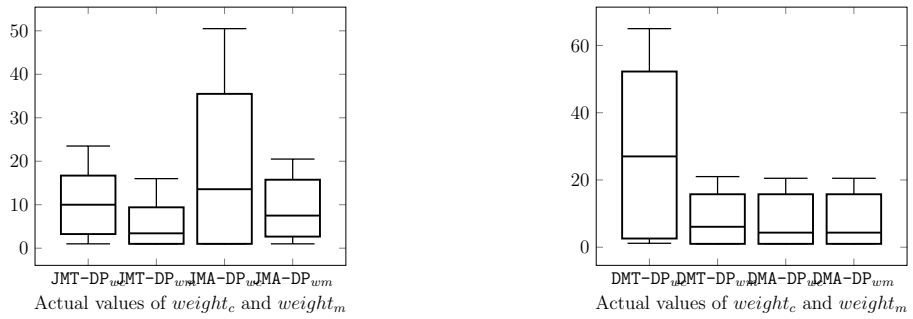
We also use the random TCP as a baseline. The random TCP runs all the test cases randomly, therefore the performance of the random TCP might vary across different runs. To mitigate the randomness, we run the random TCP 500 times on each subject and obtain the average performance in APFD. Following existing work [16], we denote the random TCP technique as `RT`.

In this work, we use a defect prediction model (DP) to detect buggy code and further weighting source code units. By combining all IVs with the defect prediction model, we can define 8 variants of `QTEP`. Based on acronyms for IV options in Table 5.3, the **8** variants of `QTEP` are `DMT-DP`, `DMA-DP`, `DST-DP`, `DSA-DP`, `JMT-DP`, `JMA-DP`, `JST-DP`, and `JSA-DP`.

To investigate the TCP performance of `QTEP`, we compare the 8 combinations from IV1–IV3 and `RT` to the 8 variants in Section 5.4.

### 5.3.6 Parameter Setting

As presented in Section 5.2.2, our algorithm for weighting source code has three parameters, i.e., *weight_base* (default weight for all code units), *weight_c* (weight for detected buggy classes), and *weight_m* (weight for detected buggy methods). Different weights of these parameters could significantly affect the performance of `QTEP`. In this section, we study the




(a) (b)

Figure 5.2: The distribution of the best *weight_c* (*wc*) and *weight_m* (*wm*) for the variants of `QTEP` that are adapted from static-cover-based TCPs (a) and dynamic-coverage-based TCPs (b).

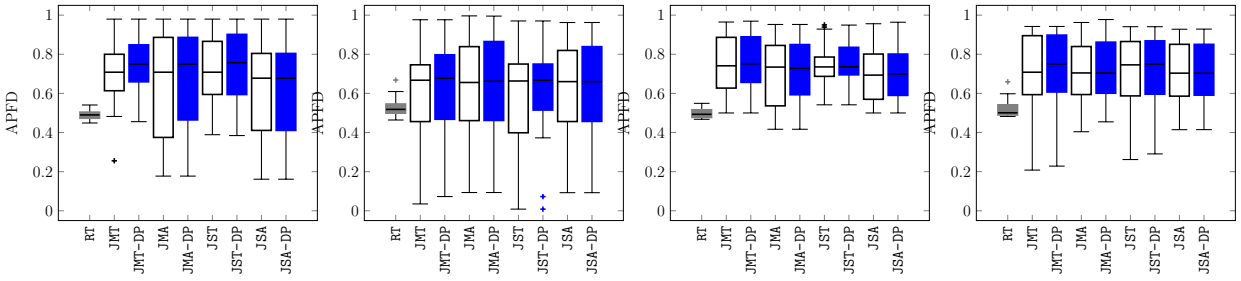
impact of the three parameters of `QTEP` on the performance of prioritizing both regression and all test cases (both regression and new test cases). Specifically, for regression test cases, we select the first version-pair from each project listed in Table 3.4 as experimental subjects. When considering both regression and new test cases, we randomly selected 20 faulty versions from the first version-pair of each project as experimental subjects.

We then tune the parameters for each project (in Table 5.1) using each of the 8 variants from `QTEP` (described in 5.3.5) and evaluate the specific values of the parameters by the average APFD scores at the class level and the method level (with or without new test cases).

For simplifying the tuning process, we set *weight_base* equal to 1. Then, we set *weight_c* equal to $c \times weight\_base$ and *weight_m* equal to $m \times weight\_base$, we experiment $c$ and $m$ with a range from 1 to 100. We use all the combinations of the three weights in the tuning process, which includes $100 \times 100 \times 20$ (#project) $\times 16$ (#variants of `QTEP`) experiments for regression test cases, and $100 \times 100 \times 20$ (#project) $\times 20$ (#mutation fault version) $\times 16$ (#variants of `QTEP`) experiments for all test cases (regression and new test cases).

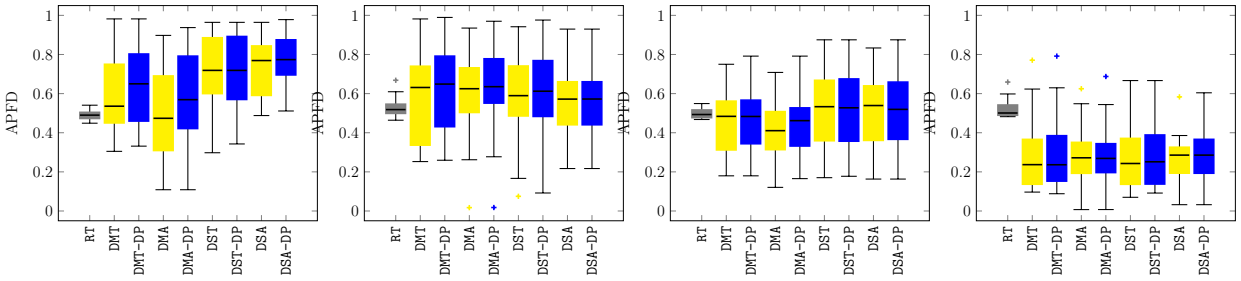
Figure 5.2 (a) and Figure 5.2 (b) show the distribution of the best values of parameters *weight_c* and *weight_m* for all the 8 variants of `QTEP` on the 20 version-pairs. We could see that the best values of *weight_c* and *weight_m* vary dramatically for different projects. On average, for variants of `QTEP` that are adapted from static-coverage-based TCPs, *weight_c* is `16.7` times of *weight_base* and *weight_m* is `13.1` times of *weight_base*. For variants of `QTEP` that are adapted from dynamic-coverage-based TCPs, *weight_c* is `18.7` times of *weight_base* and *weight_m* is `11.6` times of *weight_base*.

In this work, we use the best values of *weight_c* and *weight_m* that are obtained from the first version-pair of a project as default parameters for all the left version-pairs of this




(a) C-R (b) M-R (c) C-RN (d) M-RN

Figure 5.3: Results of static-coverage-based variants of QTEP and static-coverage-based TCPs, i.e., random ●, static coverage-based TCPs ○, static-coverage-based variants of QTEP ●.




(a) C-R (b) M-R (c) C-RN (d) M-RN

Figure 5.4: Results of dynamic-coverage-based TCPs, i.e., random ●, dynamic-coverage-based TCPs ●, dynamic-coverage-based variants of QTEP ●.

project. Note that since project Xml-Security only has one available version-pair from the existing benchmark dataset, thus for this project, we tune parameters and report the performance on the same version-pair.

## 5.4 Results and Analysis

### 5.4.1 RQ1 & RQ2: Performance of QTEP for Regression Test Cases and for All Test Cases

Figure 5.3 and Figure 5.4 show the comparison results in the four scenarios on each of the 20 version-pairs. Specifically, they show the boxplots of the APFD values for the 8 variants of QTEP, the eight variants of coverage-based TCPs, and the random baseline RT in the four scenarios. Each sub-figure presents the detailed APFD results of one type

of QTEP (i.e., static-coverage-based or dynamic-coverage-based variants of QTEP) and the corresponding baseline TCPs on a specific scenario. For example, Figure 5.3 (a) shows the C-R (regression test class) scenario of static-coverage-based techniques, RT, and static-coverage-based variants of QTEP, while Figure 5.4 (a) shows the C-R scenario of dynamic-coverage-based techniques, RT, and dynamic-coverage-based variants of QTEP. Each boxplot presents the APFD distribution (median and upper/lower quartiles) of prioritization results of one variant of QTEP on the 20 version-pairs. We use gray (●), white (○), yellow (●), and blue (●) boxes to represent the random, static-coverage-based, dynamic-coverage-based, and QTEP techniques respectively.

The figures show that overall QTEP could outperform corresponding traditional coverage-based TCPs (○ and ●) and RT (●) for both regression test cases and new test cases at both method-level and class-level TCPs. In addition, static-coverage-based variants of QTEP techniques are overall more effective than dynamic-coverage-based variants of QTEP. Specifically, for C-R, among all examined TCP techniques, JMT-DP produces the best APFD with a median value of 0.75, which is almost 6% higher than the best traditional coverage-based technique, i.e., JMT. For M-R, JMT-DP outperforms all other examined TCPs. While considering new test cases, JMA-DP and JST-DP produce the best performance for C-RN and M-RN respectively.

We further take a closer look at each individual program. To save space, we only show the detailed comparison between the results of static-coverage-based variants of QTEP and the results of the corresponding coverage-based TCPs, since they are overall more effective than dynamic-coverage-based variants of QTEP.

Table 5.4 shows the average APFD values of all static-coverage-based variants of QTEP and the corresponding coverage-based TCPs on each project. Numbers in brackets are the improvements QTEP compared to corresponding coverage-based TCPs. We can see that QTEP variants improve the APFD values for all the projects. However, the improvement varies on different projects. For example, on project Time&Money, JMA-DP achieves the best APFD for C-R (i.e., 0.82), which 23 percentage points higher than the corresponding JMA (i.e., 0.59). While on XStream, the improvement is only one percentage point. In the worst case, e.g., JMA-DP, QTEP does not improve traditional coverage-based TCPs. The same phenomenon is also observed in dynamic-coverage-based variants of QTEP.

Note that we can see that the improvements vary on different projects, this can be because the performance of CLAMI varies on different projects. To explore this, we further compute the Spearman correlation between the false positive rates and the improvements of QTEP on all projects. Results show that the Spearman correlation values for the false

Table 5.4: Comparison between the static-coverage-based variants of QTEP and the corresponding coverage-based TCPs for each project

| Subject | Scenario | JMT | JMT-DP | JMA | JMA-DP | JST | JST-DP | JSA | JSA-DP |
|---|---|---|---|---|---|---|---|---|---|
| Time&Money | C-R | 0.65 | 0.71(+0.06) | 0.59 | 0.82(+0.23) | 0.82 | 0.82 | 0.77 | 0.77 |
| | M-R | 0.24 | 0.27(+0.03) | 0.49 | 0.50(+0.01) | 0.12 | 0.29(+0.17) | 0.47 | 0.49(+0.02) |
| | C-RN | 0.61 | 0.62(+0.01) | 0.63 | 0.64(+0.01) | 0.64 | 0.64 | 0.69 | 0.69 |
| | M-RN | 0.36 | 0.39(+0.03) | 0.60 | 0.59 | 0.34 | 0.43(+0.09) | 0.57 | 0.57 |
| Mime4J | C-R | 0.65 | 0.71(+0.06) | 0.61 | 0.62(+0.01) | 0.73 | 0.86(+0.13) | 0.59 | 0.59 |
| | M-R | 0.71 | 0.75(+0.04) | 0.55 | 0.55 | 0.69 | 0.69 | 0.55 | 0.55 |
| | C-RN | 0.75 | 0.76(+0.01) | 0.65 | 0.71(+0.06) | 0.70 | 0.71(+0.01) | 0.62 | 0.62 |
| | M-RN | 0.68 | 0.70(+0.02) | 0.48 | 0.48 | 0.63 | 0.63 | 0.48 | 0.48 |
| Jaxen | C-R | 0.90 | 0.95(+0.05) | 0.90 | 0.90 | 0.90 | 0.90 | 0.85 | 0.85 |
| | M-R | 0.67 | 0.69(+0.02) | 0.77 | 0.77 | 0.73 | 0.73 | 0.78 | 0.78 |
| | C-RN | 0.80 | 0.80 | 0.74 | 0.75(+0.01) | 0.80 | 0.81(+0.01) | 0.65 | 0.65 |
| | M-RN | 0.65 | 0.68(+0.03) | 0.66 | 0.66 | 0.68 | 0.71(+0.03) | 0.68 | 0.68 |
| Xml-Security | C-R | 0.71 | 0.73(+0.02) | 0.38 | 0.38 | 0.71 | 0.73(+0.02) | 0.38 | 0.38 |
| | M-R | 0.97 | 0.97 | 0.84 | 0.84 | 0.97 | 0.97 | 0.84 | 0.84 |
| | C-RN | 0.50 | 0.50 | 0.42 | 0.42 | 0.54 | 0.54 | 0.50 | 0.50 |
| | M-RN | 0.91 | 0.91 | 0.83 | 0.83 | 0.94 | 0.94 | 0.85 | 0.85 |
| Xstream | C-R | 0.72 | 0.76(+0.04) | 0.64 | 0.65(+0.01) | 0.73 | 0.73 | 0.66 | 0.67(+0.01) |
| | M-R | 0.66 | 0.67(+0.01) | 0.72 | 0.73(+0.01) | 0.66 | 0.68(+0.02) | 0.73 | 0.73 |
| | C-RN | 0.88 | 0.89(+0.01) | 0.82 | 0.82 | 0.86 | 0.87(+0.01) | 0.82 | 0.82 |
| | M-RN | 0.76 | 0.76 | 0.81 | 0.82(+0.01) | 0.76 | 0.77(+0.01) | 0.81 | 0.81 |
| Commons-Lang | C-R | 0.67 | 0.71(+0.04) | 0.75 | 0.76(+0.01) | 0.52 | 0.55(+0.03) | 0.70 | 0.70 |
| | M-R | 0.36 | 0.37(+0.01) | 0.37 | 0.38(+0.01) | 0.20 | 0.24(+0.04) | 0.35 | 0.35 |
| | C-RN | 0.67 | 0.67 | 0.69 | 0.69 | 0.70 | 0.70 | 0.72 | 0.72 |
| | M-RN | 0.67 | 0.67 | 0.61 | 0.62(+0.01) | 0.54 | 0.57(+0.03) | 0.62 | 0.62 |
| Joda-Time | C-R | 0.65 | 0.67(+0.02) | 0.61 | 0.62(+0.01) | 0.62 | 0.62 | 0.47 | 0.50(+0.03) |
| | M-R | 0.70 | 0.72(+0.02) | 0.76 | 0.78(+0.02) | 0.70 | 0.70 | 0.66 | 0.73(+0.07) |
| | C-RN | 0.71 | 0.71 | 0.66 | 0.67(+0.01) | 0.72 | 0.72 | 0.70 | 0.70 |
| | M-RN | 0.83 | 0.83 | 0.80 | 0.80 | 0.85 | 0.85 | 0.78 | 0.78 |

positive rates and the improvements of QTEP are -0.50. This indicates that the performance of QTEP on each project is negatively correlated with the false positive rate of the investigated code inspection technique on this project.

In addition, the above figures (i.e., Figure 5.3 and Figure 5.4) show that all static-coverage-based variants of QTEP generate better results than RT and the improvement ranges from 9 to 30 percentage points. However, we also note that the performance of dynamic-coverage-based variants of QTEP has a dramatically decline when considering M-RN and C-RN compared to static-coverage-based variants of QTEP, and cannot even outperform RT. For example, the APFD of DSA-DP in M-RN is only 0.28, which is 26 percentage points lower than RT. This is because the dynamic coverage information comes from the last

execution of the test suite, which does not contain the new test cases. Thus, the faults that can be revealed only by the new test cases are ignored since the coverage information of new test cases is always unavailable in the dynamic-coverage-based TCPs [153, 154]. While, static coverage information of both regression and new test cases could be obtained by static code analysis. Thus, the performances of static-coverage-based variants of QTEP are similar between with the new test cases and without the new test cases.

For statistical tests, we also conduct the Wilcoxon signed-rank test ($p < 0.05$) to compare the performance of QTEP and existing TCPs. Specifically, we compare each variant of QTEP with its corresponding coverage-based TCP technique on all projects for both regression and new test cases. Results show that two of the 8 variants of QTEP could achieve significantly better performance than the corresponding coverage-based TCPs (i.e., JMT-DP and DSA-DP). For the other eight variants, their performances are slightly better or equal to the corresponding coverage-based TCPs.

**In summary, QTEP is overall more effective than corresponding coverage-based TCPs. While dynamic-coverage-based QTEP variants exhibit significantly better performance on regression test cases than on all test cases, static-coverage-based QTEP variants produce similarly good performance on both regression test cases and all test cases (both regression and new test cases).**

### 5.4.2 RQ3: Comparison of bug finding times of QTEP and existing TCPs.

In RQ1 and RQ2, we show that QTEP is more efficient in finding the failed test cases than traditional TCPs. In this RQ, we further evaluate QTEP and traditional TCPs regarding the time cost for finding bugs by running test cases with the orders generated by QTEP and existing TCPs. Specifically, we run test cases with orders generated by the best variant of QTEP, i.e., JMT-DP and the best traditional TCP, i.e., JMT, on each version pair, during which we collect the time cost of finding all the bugs on each version pair. The time saved by JMT-DP on each project compared to JMT is presented in Table 5.5. We can see that QTEP saves the bug finding time on six of the seven experimental projects and the percentage of time saved varies from 2% to 45.5%.

### 5.4.3 Time and Memory Overhead

Comparing with traditional coverage-based TCPs, the extra running overhead of QTEP depends on the applied code inspection technique, i.e., defect prediction models, and our

Table 5.5: Time saved by QTEP. Numbers in brackets are the percentages of bug-finding-time saved by QTEP compared to the best traditional TCP.

| Subject | Time Saved by QTEP(s) |
|---|---|
| Time&Money | 1.2 (20.1%) |
| Mime4J | 11.1 (8.6%) |
| Jaxen | 13.5 (2.3%) |
| Xml-Security | 0 |
| Xstream | 3.5 (6.2%) |
| Commons-Lang | 69.0 (10.0%) |
| Joda-Time | 110.3 (45.5%) |

Table 5.6: The average time cost of QTEP on each subject.

| Subject | Time (s) | |
|---|---|---|
| | Defect prediction (i.e., CLAMI [192]) | Weighting code |
| Time&Money | 0.4 | $< 0.1$ |
| Mime4J | 0.4 | $< 0.1$ |
| Jaxen | 0.6 | $< 0.1$ |
| Xml-Security | 0.2 | $< 0.1$ |
| Xstream | 0.9 | $< 0.1$ |
| Commons-Lang | 1.2 | $< 0.1$ |
| Joda-Time | 1.6 | $< 0.1$ |

source-code-weight algorithm. To understand the overhead of QTEP, we collect the time and space costs for all experiments, and details are presented in Table 5.6. We can see that the execution time for weighting source code is less than 0.1 seconds and the execution time for running defect prediction models varies from 0.2 to 1.6 seconds. The largest project (in terms of the number of test cases), Joda-Time, uses 96.5MB of memory. As shown in the table, code inspection techniques spent more time than weighting source code units. The total time cost on each project is less than 2 seconds. Overall, the results demonstrate QTEP's practical aspect.

## 5.5 Threats to Validity

To evaluate the quality of prioritization, we choose APFD, which has been extensively used in the field of TCP. However, APFD cannot reflect time and space costs or the severity

of faults. We plan to use more metrics, e.g., APFDc [58], to reduce this threat. In this work, all the experiment subjects are open-source projects and written in Java with JUnit test cases. Although they are popular projects and widely used in TCP research, our findings may not be generalizable to commercial projects or projects in other languages. To mitigate this threat, we plan to explore the effectiveness of `QTEP` on C/C++ projects in the future.

## 5.6 Summary

In this chapter, to address the limitations of existing TCP algorithms, we present a quality-aware TCP technique named `QTEP`. Specifically, we leverage code inspection techniques, i.e., statistical defect prediction models and static bug detection techniques, to detect fault-prone source code and then adapt existing coverage-based TCP algorithms by considering the weighted defectiveness of source code. Our evaluation shows that `QTEP` could improve existing coverage-based TCP techniques for both regression test cases and all test cases. As future work, we plan to explore the impact of fault-revealing capability of test suites and the severities of different bugs on the performance of `QTEP`. We also plan to investigate the different aggregations of code inspection results in `QTEP`.

# Chapter 6

# Leveraging Machine Learning Classifiers to Automate the Identification of Risky Code Review Requests

This chapter presents our approach to improving traditional code review by leveraging machine learning based classifiers to identify the risky code change requests, i.e., changes that have multiple rounds of code review or are reverted. This work has been submitted to the 16th International Conference on Mining Software Repositories (MSR'19).

## 6.1 Motivation and Background

Code changes could be risky, e.g., they may introduce quality issues such as bugs, improper implementations, maintenance issues, etc. As a consequence, reviewing them could require much more code review effort. In this study, we use the code review effort as an indicator to define the regular and risky changes. Specifically, we define a *risky change* as a code change that has multiple rounds of code review or is reverted. All other changes are considered as *regular changes*.

Code review is a manual inspection of source code by humans, aims at identifying potential defects and quality problems in the source code before its deployment in a live environment [62, 162, 182, 228]. However, such a manual inspection could be a time-consuming

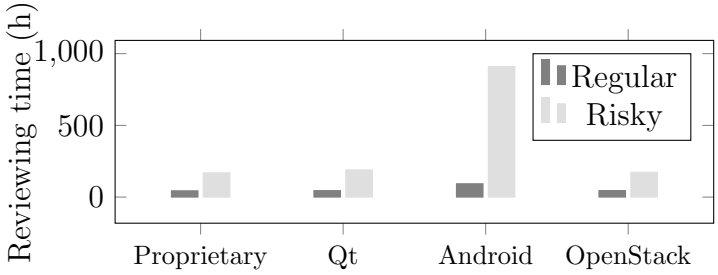



Figure 6.1: Average time cost for reviewing regular and risky changes.

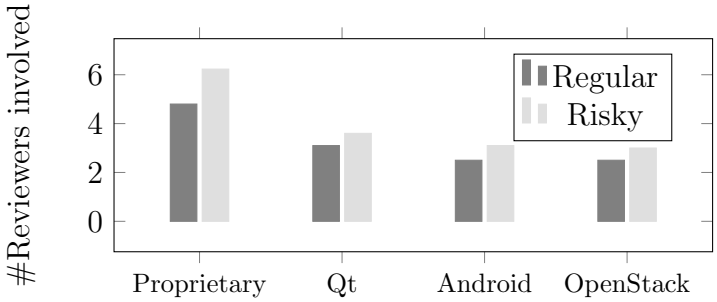



Figure 6.2: Average number of reviewers involved for reviewing changes.

and expensive process [105, 162, 182, 228, 229, 264, 267]. For example, to effectively assess a code change, developers are required to read, understand, and critique the code change [229, 267]. Moreover, if the code changes cannot pass the first round of code review, developers have to conduct a second round of code review and even more rounds until they resolve all the review suggestions.

In this section, we motivate this study by showing the review effort of regular and risky changes, i.e., review time cost, the number of reviewers involved on one proprietary project and three widely-used open-source projects, i.e., Qt, Android, and OpenStack (details are in Table 6.1). Specifically, for each regular change and risky change from the four projects, we collect the time reviewers spent on reviewing it and the number of reviewers involved. Note that we use the difference between the submission timestamp and the resolution timestamp of a review request of a change to assess the review time, since the exact time cost is not recorded in the code review systems of the four projects. We then average the review time and the number of reviewers involved for all changes for each project.

Figure 6.1 shows the average review time for each project. As shown in the figure, the time cost for reviewing risky changes could be 10X of that for reviewing regular changes

(in project Android) and on average is 7X across the four projects. Figure 6.2 shows the average number of reviewers involved to review both the regular and risky changes. As we can see, reviewing risky changes requires more reviewers to collaborate together than reviewing the regular changes in each of the four projects. Our manual inspection on randomly selected 100 risky changes from the four projects shows that after the first round of code review, the initial reviewers often request reviewers with more experience or higher permission to double check or approve the updated changes. Overall, the average number of reviewers for reviewing risky changes is 1.3X as that for reviewing the regular changes.

From Figures 6.1 and 6.2, we can conclude that reviewing risky changes requires significantly more effort than reviewing regular changes. Intuitively, finding these risky changes before starting the code review process can provide developers early information about their changes so that they have a chance to improve the changes and further accelerate the code review process.

In this work, we first propose a feedback-driven and heuristics-based approach to obtain change intents for understanding the changes. We then characterize risky changes by using various features extracted from change metadata and the change intents. We further explore the feasibility of automatically classifying risky changes. We conduct our study on a large-scale proprietary project and three large-scale open-source projects, i.e., Qt, Android, and OpenStack. Our results show that, (i) code changes with specific intents are more like to be risky, namely new features and refactoring changes have a significantly higher probability to be risky, (ii) machine learning based prediction models can efficiently help classify risky changes where the best prediction model using Random Forest, achieves AUC values larger than 0.71 on each of the four subjects, and (iii) prediction models built for code changes with specific intents achieve better performance than prediction models without considering the change intent, the improvement in AUC can be up to 19 percentage points and is 7.4 percentage points on average.

## 6.2 Approach

Figure 6.3 illustrates that our approach consists of four steps: (1) labeling each history change as regular or risky to indicate whether the change has multiple rounds of code review effort or is reverted (Section 6.2.1), (2) analyzing the change intents (Section 6.2.2), (3) extracting features to represent the changes (Section 6.2.3), and (4) using the features and labels to build and train prediction models and apply the models to predict new changes (Section 6.2.4).

Table 6.1: Projects used in this study. **Lang** is the program language of each subject. **#CR** is the number of code changes. **Rate** is the rate of risky changes, which is measured in a percentage.

| **Project** | **Lang** | **First Date** | **Last Date** | **#CR** | **Rate** |
|---|---|---|---|---|---|
| Proprietary | C# | 5/05/2015 | 5/22/2018 | >100K | ∼15 |
| Qt | C | 5/17/2011 | 5/25/2012 | 23,041 | 39.28 |
| Android | JAVA | 7/18/2011 | 5/31/2012 | 7,120 | 31.75 |
| OpenStack | Python | 7/18/2011 | 5/31/2012 | 6,430 | 43.31 |

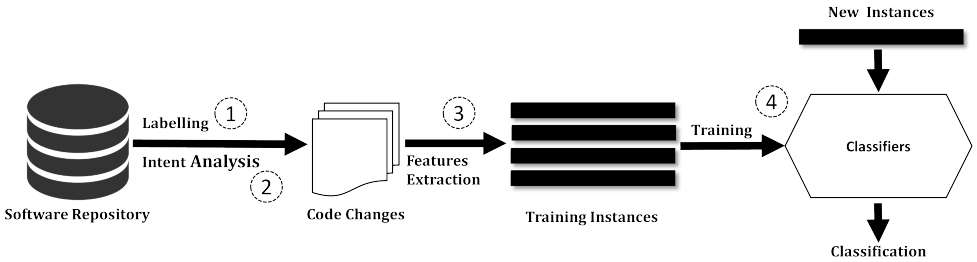



Figure 6.3: The overview of our risky change prediction approach.

### 6.2.1 Labeling Risky Changes

The first step of our approach is to label each change as regular or risky based on its code review history. Specifically, for the proprietary project, we extracted its code review database, and checked code review iterations each change has. If the number is larger than two, we labeled it as risky otherwise we labeled it as regular. For the three open-source projects, since their code review systems do not maintain the corresponding code review iteration counts, we use a heuristic approach to collect the regular and risky changes, i.e., a code review request of a change may have multiple rounds of code review, for each round of code review, developers are required to submit a patchset to be reviewed, we counted the number of patchsets for the code review request of each code change. If the number of submitted patchsets is larger than one, we labeled the change as risky otherwise we labeled it as regular. In addition, We also consider the reverted changes as risky.

The reason to use these thresholds is that during our early-stage correlation analysis, we found there is a strong correlation between the code review iteration larger than two and introducing bugs or maintenance issues for changes in the code review system of the proprietary project. We have also observed a similar correlation relation on the open source

Table 6.2: Heuristics for categorizing changes.

| **Change Intent** | **Description** | **Heuristics** |
|---|---|---|
| Bug Fix | changes are made to fix bugs | 1. the commit message contains keywords: “bug” or “fix”<br>AND 2. the commit message does not contain keywords:“test case’ or “unit test” |
| Resource | changes are made to update non-source code resources, configurations, or documents | 1. the commit message contains keywords: “conf” or “license” or “legal”<br>OR 2. if no keyword is matched in step 1, the changed files do not involve source/test files |
| Feature | changes are made to implement new or update existing features | 1. the commit message contains keywords: “update” or “add” or “new” or “create” or “add” or “implement feature”<br>OR 2. changes in the ‘Other’ category that contain keywords: “enable” or “add” or “update” or “implement” or “improve” |
| Test | changes are made to add new or update existing test cases | 1. the commit message contains the keyword: “test”<br>OR 2. the changed files contain only test files or resource files |
| Refactor | changes are made to refactor existing code | the commit message contains the keyword: “refactor” |
| Merge | changes are made to merge branches | the commit message contains keywords: “merge” or “merging” or “integrate” or “integrated” or “integrated” |
| Deprecate | changes are made to remove deprecated code | the commit message contains keywords: “deprecat” or “delete” or “clean up” |
| Auto | changes that are committed by automated accounts or bots | the change is submitted by automated accounts or bots |
| Others | changes that are not in the above categories | - |

projects, i.e., changes with more than one patchsets have a higher probability to introduce bugs or maintenance issues.

### 6.2.2 Intent Analysis

Many approaches have been proposed to characterize and classify changes based on the change intents [9, 90, 217, 253, 308]. Most of them consider the high-level change intents, e.g., corrective, adaptive, or perfective [253,308]. In this study, we leverage the fine-grained change intent categories proposed in Hindle et al. [90], which is showed in its Table 3, to categorize changes. Note that there are more than 20 different categories described in [90]. We started with manual analysis on randomly selected 200 changes from the proprietary project, and found that some of the categories have very few number of changes, e.g., 'Legal', 'Build', 'Branch' have less than 3 changes. We then group the small categories into larger ones for obtaining more instances. For example, we have grouped 'Legal', 'Data', 'Versioning', 'Platform Specific', and 'Documentation' into '**Resource**', grouped 'Rename' and 'Token Replace' into '**Refactor**', etc. Finally, we use eight types of change intents to categorize changes. Note that we also have an '**Other**' category for changes that do not fall into any of the eight categories.

Table 6.2 shows the nine types of changes, their descriptions, and the heuristics we used to automatically classify changes. Instead of manually labeling changes, in this work we automate the classification process by using well-refined heuristics. Thus, the accuracy of heuristics could significantly affect the result of this study. To improve the accuracy of the classification of changes, we used a feedback-driven approach to design and refine the heuristics for each change intent and the details are as follows:

- **Step 1**: With a randomly selected 200 instances, the first two authors first classify them into the nine categories manually and independently. Specifically, after reading the commit message and checking the changed files of a change, they label the change based on their experience. For the the classification conflicts (less than 5%), the third author inspects them independently and the first three authors make a decision for each conflict together. Then they initialize the heuristics for each category.
- **Step 2**: With the initialized heuristics, we classify all changes into at least one of the nine categories. For each category, we randomly collect 50 instances and the authors work together to manually check its accuracy.

- **Step 3**: If the accuracy of a category is lower than 80%, we further refine the heuristics and then redo **Step 2**, otherwise we keep the heuristics for classifying changes.

Taking the '**Test**' category as an example, in **Step 1**, we found that most changes from it have keywords "test" or "testing" in their commit messages. Thus, the initialized heuristics we designed for '**Test**' is that the commit message of a change contains the keywords "test". In **Step 2**, we randomly checked 50 of the collected changes labeled as '**Test**'. Our manual inspection revealed that almost a half of them were false positives, and we also found that most of the false positives have irrelevant commit messages, e.g., "... use backup config if test fails ..." and "... send a test message ...". To improve the heuristics, in **Step 3**, we added another heuristic, which is the changed files can only be test files (e.g., file names or paths contain the keyword "test") or resource files. Then we redo **Step 2** again, by using the new heuristics, the accuracy of '**Test**' category is around 90%.

We use the above steps to refine the heuristics of each category to ensure the classification of change intents has a higher accuracy. Table 6.2 shows the final heuristics.

### 6.2.3 Feature Extraction

In this study, we use the following features for building machine learning based risky change prediction models.

- **Change Intent**: As code changes could be classified into different categories based on their intents, we assume that changes with different intents have different impacts on the quality of changes. We use a vector to represent the change intents of a change. Each element in the vector is a binary value, i.e., 1 or 0, representing whether the change has a specific intent or not. The change intents we considered are listed in Table 6.2.

- **Revision History**: As presented in previous research [125], the revision history of a file can be a factor to predict its quality. In this study, we also explore the impact of revision history on predicting risky changes. Specifically, given a code change, we consider the number of files in this change that have been revised in the last 30 and 90 days, and the number of revision on all the involved files of this change in the last 30 and 90 days.

- **Owner Experience**: This set of features represent the experience of a change's committer. We use a committer's commit history information to represent her/his experience, which includes the total number of changes submitted, the total number of risky changes submitted, etc. We assume that the committer's experience affects the quality of the changes submitted.

- **Word2Vec Features**: Word embedding is a feature learning technique in natural language processing where individual words are no longer treated as unique features, but represented as $d$-dimensional vector of real numbers that capture their contextual semantic meanings [29, 171]. We train the embedding model by using all data from each project. With the trained word embedding model, each word can be transformed into a $d$-dimensional vector where $d$ is set to 100 as suggested in previous studies [304, 310]. Meanwhile a code change can be transformed into a matrix in which each row represents a term in its commit message. We then transform the code change matrix into a vector by averaging all the word vectors the code change contains, as described in [310].

- **Process Features**: Various process features have been shown to help predict software bugs [104,220,256]. In this study, we also consider the following process features: code addition, code deletion, number of changed files, and the types of changed files. Note that for the types of changed files, following existing studies [90, 91], we group files into source files, test files, configuration files, scripts, documentations, and others based on their suffixes and file paths. Specifically, we consider files with extensions: .java, .cs, .py, .js, .c, .cpp, .cc, .cp, .cxx, .c++, .h, .hpp, .hh, .hp, .hxx, and .h++, as the source files. Among them, files that contain 'test' in the paths or file names are considered as test files. Files with extensions: .script, .sh, .bash are considered as scripts. Files with extensions: .xml, .conf, .MF are considered as configuration files. Files with extensions: .htm, .html. .css, .txt, are considered as documentation files. And, remaining files are considered as others.

- **Metadata**: In addition to the above features, we also use metadata features. Specifically, given a code change, we collect its commit minute (0, 1, 2, ..., 59), commit hour (0, 1, 2, ..., 23), commit day in a week (Sunday, Monday, ..., Saturday), commit day in a month (0, 1, 2, ..., 30), commit month in a year (0, 1, 2, ..., 11), and source file/path names.

In total, in this study we leverage 132 features to build machine learning based risky change prediction models, i.e., eight are from change intents, four are from revision history,

five are from owner experience, 100 are from word2vec features, ten are from process features, and five are from metadata features.

### 6.2.4 Building Models and Predicting Risky Changes

After we obtain the features for each code change, we split the data into the training dataset and test dataset. We build and train the machine learning based prediction models on the training dataset. Then we use the test dataset to evaluate the performance of the models.

Note that different from classifying code changes in Chapter 3, classifying code review requests are time independent. There are two reasons, first code changes are committed chronologically or there can be conflicts while code review requests are independent and are unnecessary to be chronological. Second, the labelling process of code changes is time sensitive, e.g., the ground-truth of a code change is obtained by blaming later bug-fixing changes as shown in Figure 3.11, while the label of a code review request comes from the real code review actives when a code review is submitted, which does not depend on other code review requests. In this work, we do not split the dataset by following a certain order in time as we did in Section 3.4 for change-level defect prediction. Following existing studies [89, 104, 120, 122, 220, 262], we use the widely used 10-fold cross-validation to evaluate the prediction models. The process of 10-fold cross-validation is: 1) separating the data set into 10 partitions randomly; 2) using one partition as the test data and the other nine partitions as the training data; 3) repeating step 2) with a different partition as the test data until all data have a predicted label; 4) computing the evaluation results through comparison between the predicted labels and the actual labels of the data.

## 6.3 Experimental Study

### 6.3.1 Research Questions

**RQ1:** What are the distributions of risky and regular changes regarding change intents?

**RQ2:** Is it feasible to predict risky changes by using machine learning based classifiers with features extracted from change metadata and change intents?

**RQ3:** Do the specific prediction models (classifiers trained on changes with a specific intent) outperform the general models (classifiers trained on all changes)?

**RQ4:** Does the performance of predicting risky changes with a single intent differ from predicting risky changes with multiple intents?

In RQ1, we aim at understanding the distributions of changes regarding the change intents. In RQ2, we explore the feasibility of leveraging machine learning models to predict risky changes. In RQ3, we aim to explore whether machine learning classifiers built for changes with a specific intent can generate better performance than classifiers built on all data without considering the change intents. In RQ4, we investigate the difference in predicting risky changes with a single intent and changes with multiple intents.

### 6.3.2 Experiment Data

In order to address our research questions, we perform an empirical study of software projects that actively adopt the code review process. We begin with the review dataset of Android, Qt, and OpenStack provided by Hamasaki et al. [76]. The three projects adopt the Gerrit[1] code review system. We also expand the review dataset to include code review data from a large-scale proprietary project, which uses a custom code review system. Details of the projects are in Table 6.1.

Android[2] is an operating system for mobile devices that is developed by Google. Qt[3] is a cross-platform application and UI framework. OpenStack[4] is an open-source software platform for cloud computing. The last project is a widely used web service from proprietary.

### 6.3.3 Evaluation Measures

To measure the performance of predicting risky changes, we use four metrics: *Precision*, *Recall*, and $F1$. These three metrics are widely adopted to evaluate prediction tasks [170, 190, 326]. AUC is the area under the ROC curve, which measures the overall discrimination ability of a classifier. It has been widely used to evaluate classification algorithms in prediction tasks [65, 191, 216, 318]. The AUC is a threshold-independent performance measure that evaluates the ability of classifiers in discriminating between defective and clean modules. The AUC score for a perfect model would be 1, for random guessing would

[1]https://www.gerritcodereview.com/
[2]https://www.android.com/
[3]https://www.qt.io/developers/
[4]https://www.openstack.org/

be 0.5, and for a model predicting all instances as true or false would be 0. A machine learning model is considered applicable to classify a given dataset if the AUC score is larger than 0.7 [262].

## 6.4 Results and Analysis

Table 6.3: Taxonomy of code changes. **#Ch** is the number of code changes. **Perc** is the percentage of changes with a specific change intent among all the changes. **Rate** is the rate of risky changes, which is measured in a percentage. **Single** contains changes that have only one intent. **Multiple** contains changes that have more than one intent.

| **Change Intent** | **Proprietary** | | **Qt** | | | **Android** | | | **OpenStack** | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | **Perc** | **Rate** | **#Ch** | **Perc** | **Rate** | **#Ch** | **Perc** | **Rate** | **#Ch** | **Perc** | **Rate** |
| Bug Fix | ∼20 | 19.1 | 9,042 | 39.24 | 35.43 | 2,143 | 30.10 | 35.88 | 3,380 | 52.57 | 46.95 |
| Resource | ∼39 | 8.98 | 5,326 | 23.12 | 28.56 | 1,561 | 21.92 | 29.66 | 2,300 | 35.77 | 34.26 |
| Feature | ∼12 | 33.55 | 3,526 | 15.30 | 55.05 | 1,735 | 24.37 | 46.63 | 771 | 12.00 | 60.83 |
| Test | ∼4 | 15.14 | 3,840 | 16.67 | 38.75 | 663 | 9.31 | 35.6 | 1,005 | 15.63 | 54.23 |
| Refactor | ∼2 | 41.97 | 696 | 3.02 | 49.28 | 117 | 1.64 | 50.43 | 195 | 3.03 | 65.64 |
| Merge | ∼4 | 14.67 | 217 | 0.94 | 35.02 | 245 | 3.44 | 13.06 | 31 | 0.48 | 48.39 |
| Deprecate | ∼6 | 18.52 | 3,826 | 16.61 | 36.96 | 752 | 10.56 | 37.9 | 905 | 14.07 | 44.75 |
| Auto | ∼10 | 0 | / | / | / | / | / | / | / | / | / |
| Others | ∼15 | 20.96 | 4,036 | 17.52 | 40.21 | 1,513 | 21.25 | 35.23 | 577 | 8.97 | 34.66 |
| **Single** | ∼87 | 17.12 | 17,200 | 74.65 | 40.69 | 5,802 | 81.50 | 38.12 | 4,209 | 65.47 | 41.29 |
| **Multiple** | ∼13 | 14.71 | 5,841 | 25.35 | 35.13 | 1,317 | 18.50 | 32.88 | 2,220 | 34.53 | 47.07 |

### 6.4.1 RQ1: Distribution of Changes Regarding Change Intents

Following the change intent taxonomy approach described in Section 6.3.1, we automatically label each change from the four projects. As reported in existing studies [66, 177], software changes could be made for multiple purposes, e.g., a change could be made for correcting bugs and refactoring existing code at the same time. Thus, we also consider labeling changes with multiple intents. Table 6.3 shows the number of changes, the percentage among all changes, and the percentage of risky changes for each change intent in the four projects. In addition, we also show the numbers of changes that have single and multiple intents. Note that the '**Auto**' changes only exist in proprietary project and we

Table 6.4: Comparison of different classifiers on predicting risky code changes. The best F1 scores and AUC values are highlighted in bold.

| Project | ADTree | | Logistic | | NB | | SVM | | RF | |
|---|---|---|---|---|---|---|---|---|---|---|
| | F1 | AUC | F1 | AUC | F1 | AUC | F1 | AUC | F1 | AUC |
| Proprietary | 0.40 | 0.65 | 0.37 | 0.72 | 0.37 | 0.72 | 0.41 | 0.63 | **0.46** | **0.76** |
| Qt | 0.53 | 0.62 | 0.54 | 0.72 | 0.41 | 0.66 | 0.34 | 0.59 | **0.57** | **0.71** |
| Android | 0.58 | 0.68 | 0.54 | 0.71 | 0.58 | 0.70 | 0.50 | 0.65 | **0.58** | **0.74** |
| OpenStack | 0.60 | 0.64 | 0.64 | 0.76 | 0.62 | 0.68 | 0.66 | 0.70 | **0.66** | **0.76** |

find all of them are regular changes, thus we exclude these changes for building change prediction models. Since there exist overlaps among different change intents, the sum of percentages of change intents is larger than 1.

As shown in Table 6.3, the distribution of changes regarding intents varies in different projects. We could also see that changes are unevenly distributed regarding the intents. For example, changes with intents '**Bug Fix**' and '**Resource**' are dominating across the four projects, e.g., they take up more 50% of all the changes, while the percentages of changes with intents '**Refactor**' and '**Merge**' are less than 4%. Note that the distribution in our dataset is consistent with that of manually categorized changes from existing study [90], in which changes under categories **Corrective** (i.e., addressing failures), **Adaptive** (i.e., changes for data and processing environment), and **Perfective** (i.e., addressing inefficiency, performance, and maintainability issues) are dominating with a percentage higher than 60%, which also confirms the effectiveness of our automated heuristic-based change classification (details are presented in Section 6.2.2).

While '**Feature**' and '**Refactor**' have significantly higher rates of risky changes (Wilcoxon signed-rank test, $p < 0.05$), this is reasonable since both the '**Feature**' and '**Refactor**' introduce new functionalities or restructure existing code snippets, which are easy to be risky. Category '**Resource**' has a significantly lower rate of risky changes across the four projects (Wilcoxon signed-rank test, $p < 0.05$). This may be because, compared to all other categories, changes in the '**Resource**' category modify the source code rarely.

Software code changes are unevenly distributed regarding change intents. Changes with specific change intents, i.e., '**Feature**' and '**Refactor**', have a significantly higher probability to be risky.

Table 6.5: Performance of predicting risky code changes with single and multiple change intents. Better F1 scores or AUC values are highlighted in bold.

| Change Intent | Proprietary | | Qt | | Android | | OpenStack | |
|---|---|---|---|---|---|---|---|---|
| | F1 | AUC | F1 | AUC | F1 | AUC | F1 | AUC |
| Single | **0.48** | **0.78** | **0.58** | **0.75** | **0.58** | **0.73** | **0.67** | **0.80** |
| Multiple | 0.41 | 0.75 | 0.54 | 0.70 | 0.57 | 0.71 | 0.66 | 0.77 |

### 6.4.2 RQ2: Feasibility of Predicting Risky Changes

This question explores whether machine learning algorithms can learn models that predict new risky changes. We use off-the-shelf machine learning algorithms from Weka [74]. We use the change intent, change history, owner experience, Word2Vec features, process features and metadata features (details are in Section 6.2.3) to build these prediction models.

We experiment with five widely used [89,104,138,170,280,326] classification algorithms, i.e., Alternating Decision Tree (ADTree), Logistic Regression (Logistic), Naive Bayes (NB), Support Vector Machine (SVM), and Random Forest (RF). Note that this work does not intend to find the best-fitting classifiers or models, but to explore the feasibility of predicting risky changes by using machine learning algorithms. Existing work [262] showed that selecting optimal parameter settings for machine learning algorithms could achieve better performance, thus we tune each of the classifiers with various parameters and use the ones that could achieve the best AUC value as our experiment settings. For each project, we build classification models and use the commonly used 10-fold cross-validation method to evaluate the prediction models [89, 104, 120, 122, 220, 262]. Taking the OpenStack as an example, for tuning Random Forest, we tune parameter **numIterations** as suggested in [74], specifically we experiment with 20 discrete values, i.e., 50, 100, 150, 200, 250, 300, 350, 400, 450, 500, 550, 600, 650, 700, 750, 800, 850, 900, 950, and 1,000, results show that when **numIterations** equals to 900, Random Forest has the best prediction performance in terms of F1 on OpenStack.

Table 6.4 shows the F1 and AUC values of each machine learning algorithm on the four experimental projects. Overall, of the five classifiers, RF consistently outperforms the others on each project. The improvements of RF compared to the other four classifiers range from 5.0 percentage points to 13.0 percentage points in AUC and 5.0 percentage points to 9.0 percentage points in F1. RF achieves similar AUC values on both the proprietary project and open-source projects, while it has a significantly lower F1 score (Wilcoxon

signed-rank test, $p < 0.05$) on the proprietary project. This may be because the CompanyX project has a lower rate of risky changes, e.g., as shown in Table 6.1, the risky rates of open-source projects are around 20.0 percentage points higher than that of the proprietary project, which makes the data distribution more unbalanced in the proprietary project. Previous studies showed that the unbalance issue could decline the F1 scores [256, 259]. As revealed in existing work [259], that the unbalance issue of a dataset does not impact the AUC measure and they suggested that a machine learning model is considered applicable to classify a given dataset if the AUC is larger than 0.7. Hence, we use the AUC to compare prediction models. We could find that among the five examined machine learning classifiers, two of them, i.e., Logistic and RF, achieve AUC values larger than 0.7 on each of the four experimental projects, which confirms the feasibility of predicting risky changes by using machine learning algorithms.

Machine learning based prediction models can help predict risky changes. The best model (i.e., RF) achieves AUC values larger than 0.71 on each experimental project.

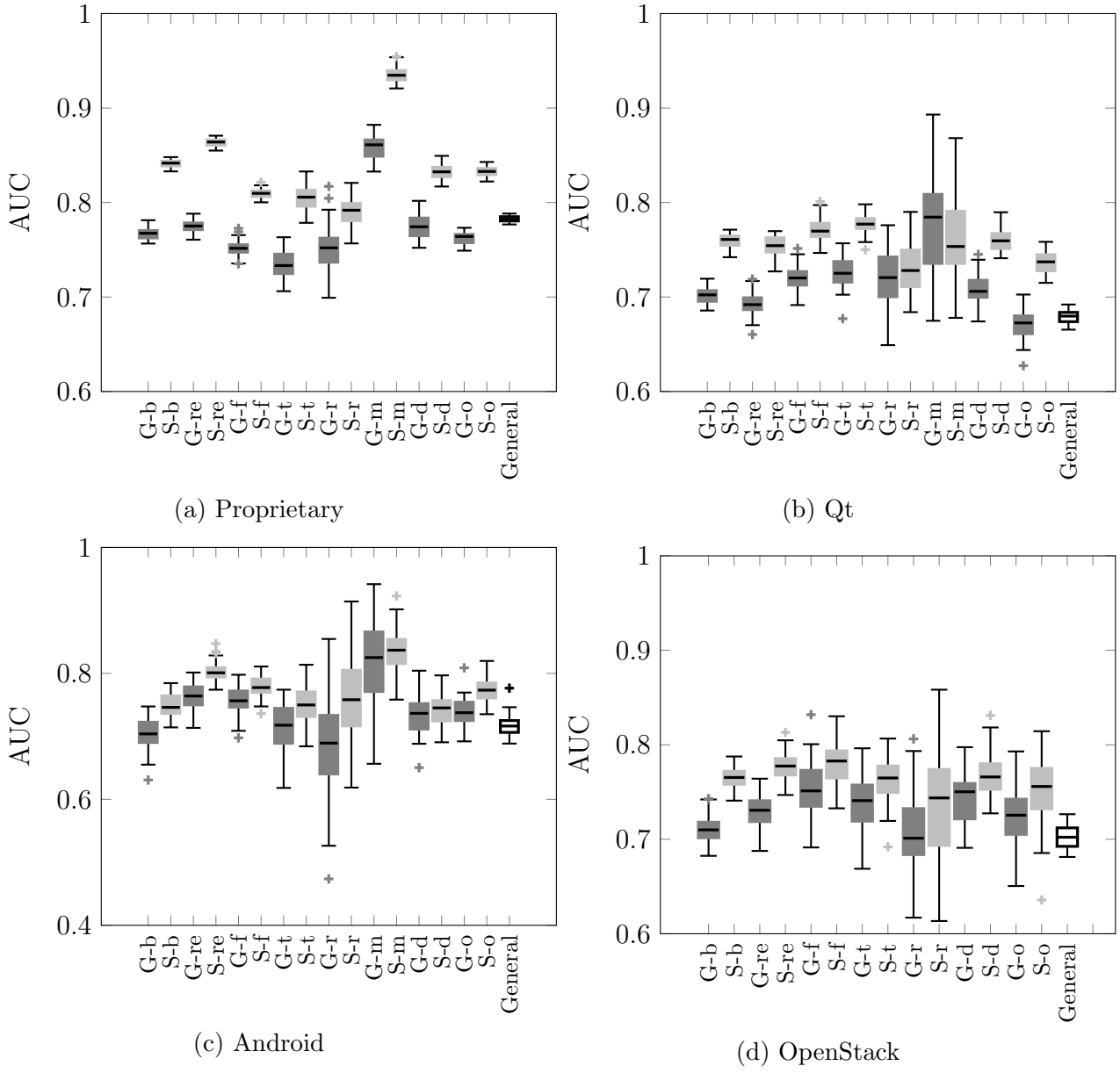
### 6.4.3 RQ3: Specific Models vs. General Models

In **RQ2**, we show that it is feasible to leverage machine learning classifiers to predict risky changes with the features extracted from changes' metadata and intents. In this RQ, we further explore whether the machine learning classifiers built and trained on changes with a specific change intent, i.e., specific model, could achieve better performance than machine learning classifiers built and trained on all changes, i.e., general model. Specifically, for each project, we build and train the RF-based specific prediction models on changes with one specific change intent. We tune each of the RF-based classifiers with various parameter values and use the ones that could achieve the best AUC value as our experiment settings. In addition, we use 10-fold cross-validation method to evaluate the prediction models. The general model on the project is trained and evaluated on all changes without considering the change intents. Note that we exclude the specific model for category '**Merge**' on the project OpenStack, because it has very few numbers of instances.

Table 6.6 shows the comparison between the performance of the specific prediction models and the general models. Regarding F1, we can see that at least half of specific models outperform the general models across the four experimental projects. For example, six out of the eight specific models on the proprietary project generate better F1 values than the general model, the improvement is up to 25.0 percentage points and is 6.0 percentage points, on average. We observe a similar situation on Qt and OpenStack, i.e., overall

Table 6.6: Comparison between machine learning classifiers built on changes from specific change intents and machine learning classifiers built on all changes (i.e., general). Numbers in parenthesis are the differences between the specific models and the general models. Better F1 scores or AUC values that are larger than that of the general models are highlighted in bold. Note that we exclude the specific model for 'Merge' on the project OpenStack, since it only has 31 instances, which is not enough for training a machine learning classifier.

| **Change Intent** | **Proprietary** | | | | **Qt** | | | | **Android** | | | | **OpenStack** | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | P | R | F1 | AUC | P | R | F1 | AUC | P | R | F1 | AUC | P | R | F1 | AUC |
| Bug Fix | 0.68 | 0.40 | **0.51**(+0.05) | **0.84**(+0.08) | 0.70 | 0.47 | 0.57(+0.00) | **0.77**(+0.06) | 0.64 | 0.49 | 0.55(-0.03) | **0.76**(+0.02) | 0.72 | 0.66 | **0.69**(+0.03) | **0.79**(+0.03) |
| Resource | 0.48 | 0.24 | 0.32(-0.14) | **0.79**(+0.03) | 0.68 | 0.39 | 0.50(-0.07) | **0.76**(+0.05) | 0.67 | 0.51 | 0.58(+0.00) | **0.82**(+0.08) | 0.68 | 0.55 | 0.61(-0.05) | **0.81**(+0.05) |
| Feature | 0.69 | 0.55 | **0.61**(+0.15) | **0.81**(+0.05) | 0.73 | 0.73 | **0.73**(+0.16) | **0.78**(+0.07) | 0.67 | 0.67 | **0.67**(+0.09) | **0.77**(+0.03) | 0.77 | 0.85 | **0.81**(+0.15) | **0.81**(+0.05) |
| Test | 0.61 | 0.22 | 0.32(-0.14) | **0.82**(+0.06) | 0.71 | 0.55 | **0.62**(+0.05) | **0.79**(+0.08) | 0.62 | 0.53 | 0.58(+0.00) | 0.74(+0.00) | 0.74 | 0.77 | **0.76**(+0.10) | **0.80**(+0.04) |
| Refactor | 0.70 | 0.62 | **0.65**(+0.19) | **0.79**(+0.03) | 0.70 | 0.64 | **0.67**(+0.10) | **0.76**(+0.05) | 0.70 | 0.68 | **0.69**(+0.11) | **0.76**(+0.02) | 0.75 | 0.81 | **0.78**(+0.12) | **0.78**(+0.02) |
| Merge | 0.86 | 0.60 | **0.71**(+0.25) | **0.95**(+0.19) | 0.58 | 0.51 | 0.55(-0.02) | **0.77**(+0.06) | 0.50 | 0.22 | 0.30(-0.28) | **0.81**(+0.07) | / | / | / | / |
| Deprecate | 0.66 | 0.37 | **0.47**(+0.01) | **0.84**(+0.08) | 0.68 | 0.51 | **0.58**(+0.01) | **0.77**(+0.06) | 0.66 | 0.56 | **0.61**(+0.03) | **0.78**(+0.04) | 0.72 | 0.65 | **0.69**(+0.03) | **0.80**(+0.04) |
| Others | 0.68 | 0.50 | **0.58**(+0.12) | **0.83**(+0.07) | 0.65 | 0.51 | 0.57(+0.00) | **0.74**(+0.03) | 0.60 | 0.51 | 0.55(-0.03) | **0.76**(+0.02) | 0.62 | 0.51 | 0.56(-0.10) | **0.78**(+0.02) |
| General | 0.51 | 0.41 | 0.46 | 0.76 | 0.60 | 0.54 | 0.57 | 0.71 | 0.61 | 0.55 | 0.58 | 0.74 | 0.68 | 0.65 | 0.66 | 0.76 |




(a) Proprietary

(b) Qt

(c) Android

(d) OpenStack

Figure 6.4: Comparison of perdition performance between the specific models and the general models. The model with a prefix "G" means using the trained general model to predict changes with a specific intent and "b" represents 'Bug Fix', "re" represents 'Resource', "f" represents 'Feature', "t" represents "Test", "r" represents "Refactor", "m" represents "Merge", "d" represents "Deprecate", and "o" represents "Others". For example **G-b** means using the trained general model to predict changes with the 'Bug Fix' intent. **S-b** is the specific model trained and evaluated on changes with the 'Bug Fix' intent.

specific models are better than the general models, the improvements are up to 16.0 and 15.0 percentage points on Qt and OpenStack respectively. However, we also observe an exception in Android, although five out of the eight specific models generate better (or the same) F1 values than the general model, the overall improvement is negative, the reason is that the '**Merge**' category has an F1 value that is 28.0 percentage points lower than that of the general model. This is because the '**Merge**' category has a much lower risky rate (i.e., 13.1%) than that of all other categories (range from 29.0% to 50.4%) in Android, which makes the '**Merge**' unbalanced. Previous studies showed that the unbalance issue of data could decline the F1 scores [256, 259]. Regarding AUC, we can observe that all the specific models outperform the general models across the four experimental projects. The improvement could be up to 19.0 percentage points and is 7.4 percentage points on average. Thus, from the comparison shown in Table 6.6, we conclude that overall the specific models achieve better prediction performance than the general models.

Above all, we show that the specific models (built on changes with a specific change intent) are overall better than the general models (built on all the changes). One could also argue that using the general models to predict changes with a specific intent may have better performance than the corresponding specific model. To explore this issue, we further examine the performance of leveraging the general models to predict changes with a specific intent. Specifically, for each change intent, we randomly divide its changes into training dataset and test dataset (66% for training, 34% for test). For the specific model, we use its training data to train the model and evaluate its performance on its test dataset. For the corresponding general model, we combine the training data from all specific models, and evaluate its performance on the test dataset of a specific model. We repeat the data splitting, model training, and evaluation 50 times to reduce bias.

Figure 6.4 shows the boxplots of the 50 times classification for each specific model and its corresponding general model on each project. In addition, we also show the boxplots of overall-general models (i.e., using 66% of all changes to train the models and evaluate the models on the left 34% changes without considering change intents). As we mentioned in Section 6.4.2, the AUC is more stable for evaluating the performance of machine learning classifiers, in this RQ, we only use AUC to evaluate the prediction models. Each boxplot presents the AUC distribution (median and upper/lower quartiles) of a prediction model. We use gray (●), light gray (●), and white (○) to represent the specific models, corresponding general models, and the overall-general models respectively. We could observe that overall the specific models outperform the general models on almost all the change intents across the four experimental projects. Specifically, for the proprietary project, the mean AUC values of the specific models are around ten percentage points higher than that of the corresponding general models. For the open-source projects, the mean AUC values of

the specific modes are around five percentage points higher than that of the corresponding general models. In addition, all the special models outperform the overall-general models.

> Prediction models built on changes with specific change intents achieve better performance than the general prediction models that do not consider the change intents.

### 6.4.4 RQ4: Single Intent vs. Multiple Intents

As showed in RQ1 (Section 6.4.1), in this study we consider labelling changes with multiple intents. This RQ explores the performance of predicting changes with a single intent and changes with multiple intents. Specifically, for each project, we build and train the RF-based prediction models with changes with only a single intent and changes with multiple intents respectively. We tune each of the RF-based classifiers with various parameter values and use the ones that could achieve the best AUC value as our experiment settings. We also use the 10-fold cross-validation method to evaluate the models.

Table 6.5 shows the F1 scores and AUC values of the prediction models for changes with single and multiple change intents in the four experimental projects. As we can see, the prediction models for changes with a single intent significantly outperform the prediction models for changes with multiple intents in both F1 and AUC across the four experimental projects (Wilcoxon signed-rank test, $p < 0.05$). In terms of F1, the improvement could be up to 7.0 percentage points and is 3.3 percentage points, on average. For AUC, the improvement could be up to 5.0 percentage points and is 3.3 percentage points, on average. One of the possible reasons for this difference is that changes with a single intent are mainly made for one specific purpose, which makes them easier to be distinguished by machine learning classifiers than complex changes with multiple intents.

> Machine learning classifiers generate better performance on changes with a single change intent than changes with multiple change intents.

## 6.5 Threats to Validity

**Internal Validity** The main threat to internal validity is about the annotation of change intents, subjectivity of annotation, and miscategorization. The annotation relied on our manually refined heuristics, and although this approach is a common practice, this process contains bias since we are not the developers of these projects. To mitigate this, authors worked independently to annotated the data and refined the heuristics. In addition, we

chose a setup that ensures that every heuristic is cross-validated and the classification conflicts have to pass a third inspection.

**External Validity** In this work, we use a proprietary project and three open-source projects to evaluate our proposed approach. Since they adopt different code review systems, i.e., the proprietary project adopts a custom code review system and the open-source projects adopt the Gerrit code review system. Thus, the proposed approach might not work for projects that adopt other code review systems.

## 6.6 Summary

This chapter presents the first study of risky changes (i.e., changes that have multiple rounds of code review effort or are reverted) by considering the change intents. We conduct our study on a large-scale proprietary project, and three open-source projects, i.e., Qt, Android, and OpenStack. Experiment results show that: (i) changes with specific intents are more like to be risky, (ii) machine learning based prediction models could help identify risky changes, and (iii) prediction models built for changes with specific intents achieve better performance than prediction models without considering the intents.

# Chapter 7

# Conclusion and Future Work

## 7.1 Conclusion

In this thesis, we demonstrated that machine learning with its capability in knowledge representation, learning, natural language processing, classification, etc., can be used to extract invaluable information from software artifacts to improve existing software reliability practices such as defect prediction, static bug detection, regression test, and code review. We saw four examples that support this claim.

In Chapter 3, we saw that a great deal of defect prediction research relies on manually designed features that encoding the statistical characteristics of programs. However, these features often fail to capture the semantic difference of programs, and such a capability is needed for building accurate prediction models. To bridge the gap between programs' semantics and defect prediction features, this thesis leverages deep learning techniques to learn a semantic representation of programs automatically from source code and further build and train defect prediction models based on these semantic features. We examine the effectiveness of the deep learning based defect prediction approaches on both the open source and commercial projects. Results show that the learned semantic features can significantly outperform existing defect prediction models.

In Chapter 4, we saw that existing rule-based static bug detection techniques often miss detecting bugs, if patterns do not appear frequently enough and are not inferred. To solve this issue, this thesis proposes `Bugram`, which leverages n-gram language models instead of rules to detect bugs. Specifically, `Bugram` models program tokens sequentially, using the n-gram language model. Token sequences from the program are then assessed according

to their probability in the learned model, and low probability sequences are marked as potential bugs. The assumption is that low probability token sequences in a program are unusual, which may indicate bugs, bad practices, or unusual/special uses of code of which developers may want to be aware. Results on 16 open-source projects show that `Bugram` is complementary to existing bug detection approaches to detect more bugs and generates less false positives.

In Chapter 5, we saw that most of existing test prioritization techniques rely on maximizing coverage information between source code and test cases to schedule test cases for finding bugs earlier. While they often do not consider the likely distribution of faults in source code. However, software faults are not often equally distributed in source code, e.g., around 80% faults are located in about 20% source code. Intuitively, test cases that cover the faulty source code should have higher priorities, since they are more likely to find faults. To solve this issue, this thesis proposes `QTEP`, which leverages machine learning models to evaluate source code quality and then adapts existing test case prioritization algorithms by considering the weighted source code quality. Evaluation on seven open-source projects shows that `QTEP` can significantly outperform existing test case prioritization techniques to find failed test cases early.

In Chapter 6, we saw that current code review practice relies on manual inspection to reveal potential quality issues in the submitted code change requests, which is inefficient and time-consuming. To solve this issue, this thesis presents the first study to understand, characterize, and automatically predict risky changes (i.e., changes that have multiple rounds of code review or are reverted) by considering their change intents and various process features. Evaluation on one proprietary project and three large-scale open-source projects (i.e., Qt, Android, and OpenStack) shows our approach is effective and can further improve the current code review process.

Taken together, these four studies provide examples of using machine learning to generate research results that can improve software reliability.

## 7.2 Future Work

The approaches that are proposed in this thesis show promising results of leveraging machine learning technologies to improve software reliability practices. Based on the findings presented in this thesis, we have identified several possible directions for future research.

**Leveraging Deep Learning to Accelerate Software Analytics.** Previous studies have shown that deep learning could help solve many software analytics problems, e.g., software

defect prediction (Chapter 3), fault localization [133], code and API suggestion [72, 290], malware classification [209,317], test report classification [110]), software traceability [107], etc. Along this line, we plan to explore more studies on applications of deep learning in software analytics. For example, the problem of automatic program, which aims at automatically finding a solution to software bugs without human intervention, has been studied for years [136, 181]. However, most of existing approaches are still not applicable for fixing complex bugs [184]. The intuition of this project is open-source community has tens of thousands of bug-fixing history records, i.e., patches. These patches can provide valuable human knowledge, exploiting such human knowledge can help automatically fix similar bugs in a new program. In this project, we plan to leverage deep learning to automatically distill expertise and past efforts of developers to help fix new bugs.

In addition, we would also like to explore the potential applications of deep learning algorithms for improving test case generation, program generation, and code comment generation.

**Learning to Improve Imbalanced Defect Prediction.** Software projects typically have imbalanced data for defect prediction, i.e., low buggy rate or few buggy instances in training dataset [193, 256], which significantly narrows down the representativeness of training dataset and induces poor prediction performance. Existing approaches towards solving this problem fall into two main directions: a) increasing buggy rate via duplicating buggy instances in training dataset with various re-sample techniques (i.e., resampling). b) borrowing buggy data from other projects to build the prediction model (i.e., cross-project prediction). However, resampling techniques simply duplicating existing buggy data, which cannot introduce new types of bugs or improve the representativeness of training dataset. Thus, the improvement of resampling techniques on defect prediction is limited. Cross-project prediction techniques often introduce noises, which limit the expected improvement. In addition, cross-project prediction requires non-trivial effort to find a similar project with enough buggy data instances, which means the performance is not guaranteed with different source projects. To address the imbalanced data problem in defect prediction, different from the above two approaches, in this project, we propose to leverage mutation generation techniques to synthesize mutation faults to increase the buggy rate and representativeness of buggy instances in training data.

**Learning to Improve Static Bug Detection.** There is a flurry of rule-based static bug detection techniques [2–4, 144], many of which have already been adopted in the industry, e.g., FindBugs [4], Facebook-Infer [3], and Google Error-Prone [2], etc. One major challenge of static bug detection techniques is the large number of false positives they report, i.e., reported bugs are not true bugs. Specifically, 30-90% of reported warnings by static bug detection tools are false positives [79]. Such a large number of false positives often

make developers reluctant to use bug detection tools entirely due to the overhead of alert inspection. In this project, we focus on the fundamental questions behind the false positives and answer the following research questions: Why do the rule-based static bug finders generate so many false positives? How can we improve these patterns and avoid false positives? This project involves an empirical study to summarize the reasons behind the false positives and approaches to addressing the false positives. The result of this project will make existing rule-based bug detection more accurate and also could provide invaluable guides for both academic researchers and industry developers to build their rule-based bug detection tools accurately.

**Designing An Integrated Software Reliability Assurance Framework.** Software quality assurance is gaining increasing attention throughout the software lifecycle. This thesis presents four machine learning based techniques to improve existing software reliability practices, i.e., software bug detection, static bug detection, regression test, and code review. However, most of the current widely used software reliability practices are independent. Developers often have to switch among different tools to conduct different quality assurance tasks, which is time-consuming and requires non-trivial manual effort. A framework that integrates existing software reliability assurance tasks used in different stages of the software development can accelerate the quality assurance throughout the entire software development cycle. For example, a framework that integrates software bug detection and automatic bug repair tools can provide developers the integrated software bug detection and bug repair service. We believe such an integrated software reliability assurance framework can help developers deliver high-quality software with speed and agility.

# Bibliography


[1] Bugs are expensive. `https://www.tricentis.com/software-fail-watch`.

[2] Error Prone. `http://errorprone.info`.

[3] Facebook-Infer. `https://fbinfer.com`.

[4] FindBugs. `http://findbugs.sourceforge.net`.

[5] PMD. `https://pmd.github.io`.

[6] Major. `http://mutation-testing.org/`, 2017.

[7] Wala. `https://github.com/wala/WALA`, 2017.

[8] M. Acharya, T. Xie, J. Pei, and J. Xu. Mining API Patterns as Partial Orders from Source Code: From Usage Scenarios to Specifications. In *FSE'07*, pages 25–34.

[9] A. Alali, H. Kagdi, and J. I. Maletic. What's a typical commit? a characterization of open source software repositories. In *ICPC'08*, pages 182–191.

[10] M. Allamanis, E. T. Barr, and C. Sutton. Learning Natural Coding Conventions. In *FSE'14*, pages 281–293.

[11] S. Amasaki, Y. Takagi, O. Mizuno, and T. Kikuno. A Bayesian Belief Network for Assessing the Likelihood of Fault Content. In *ISSRE'03*, pages 215–226.

[12] J. H. Andrews, L. C. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments? In *ICSE'05*, pages 402–411.

[13] J. H. Andrews, L. C. Briand, Y. Labiche, and A. S. Namin. Using mutation analysis for assessing and comparing testing coverage criteria. *TSE'06*, pages 608–624, 2006.

[14] J. Anvik, L. Hiew, and G. C. Murphy. Who should fix this bug? In *ICSE'06*, pages 361–370.

[15] M. J. Arafeen and H. Do. Test case prioritization using requirements-based clustering. In *ICST'13*, pages 312–321.

[16] A. Arcuri and L. Briand. A practical guide for using statistical tests to assess randomized algorithms in software engineering. In *ICSE'11*, pages 1–10.

[17] E. Arisholm, L. C. Briand, and M. Fuglerud. Data mining techniques for building fault-proneness models in telecom java software. In *ISSRE'07*, pages 215–224.

[18] T.-D. B Le, D. Lo, C. Le Goues, and L. Grunske. A learning-to-rank based fault localization approach using likely invariants. In *ISSTA'16*, pages 177–188.

[19] A. Bacchelli and C. Bird. Expectations, outcomes, and challenges of modern code review. In *ICSE'13*, pages 712–721.

[20] L. R. Bahl, P. Brown, P. V. de Souza, and R. Mercer. A Tree-Based Statistical Language Model for Natural Language Speech Recognition. *Acoustics, Speech and Signal Processing, IEEE Transactions on*, 37(7):1001–1008, 1989.

[21] T. Ball. On the limit of control flow analysis for regression test selection. *ISSTA'98*, 23(2):134–142.

[22] J. Bansiya and C. G. Davis. A hierarchical model for object-oriented design quality assessment. *TSE'02*, 28(1):4–17.

[23] M. Barnett, C. Bird, J. Brunet, and S. K. Lahiri. Helping developers help themselves: Automatic decomposition of code review changesets. In *ICSE'15*, pages 134–144.

[24] R. Bavishi, M. Pradel, and K. Sen. Context2name: A deep learning-based approach to infer natural variable names from usage contexts. *arXiv preprint arXiv:1809.05193*, 2018.

[25] O. Baysal, O. Kononenko, R. Holmes, and M. W. Godfrey. The influence of non-technical factors on code review. In *WCRE'13*, pages 122–131.

[26] O. Baysal, O. Kononenko, R. Holmes, and M. W. Godfrey. Investigating technical and non-technical factors influencing modern code review. *Empirical Software Engineering*, 21(3):932–959, 2016.

[27] M. Beller, A. Bacchelli, A. Zaidman, and E. Juergens. Modern code reviews in open-source projects: Which problems do they fix? In *MSR'14*, pages 202–211.

[28] Y. Bengio. Learning Deep Architectures for AI. *Foundations and Trends in Machine Learning*, 2(1):1–127, 2009.

[29] Y. Bengio, R. Ducharme, P. Vincent, and C. Janvin. A neural probabilistic language model. *The Journal of Machine Learning Research*, 3:1137–1155, 2003.

[30] L. Benjamin and T. Zimmermann. DynaMine: Finding Common Error Patterns by Mining Software Revision Histories. In *FSE'05*, pages 296–305.

[31] J. Black, E. Melachrinoudis, and D. Kaeli. Bi-criteria models for all-uses test suite reduction. In *ICSE'04*, pages 106–115.

[32] D. M. Blei, A. Y. Ng, and M. I. Jordan. Latent dirichlet allocation. *the Journal of Machine Learning Research*, 3:993–1022, 2003.

[33] D. Bowes, S. Counsell, T. Hall, J. Petric, and T. Shippey. Getting defect prediction into industrial practice: the elff tool. In *ISSREW'17*, pages 44–47.

[34] B. Busjaeger and T. Xie. Learning for test prioritization: an industrial case study. In *FSE'16*, pages 975–980.

[35] J. C. Campbell, A. Hindle, and J. N. Amaral. Syntax Errors Just Aren't Natural: Improving Error Reporting with Language Models. In *MSR'14*, pages 252–261.

[36] R. Carlson, H. Do, and A. Denton. A clustering approach to improving test case prioritization: An industrial case study. In *ICSM'11*, pages 382–391.

[37] K. Cem. Improving the maintainability of automated test suites. In *QW'97*.

[38] R.-Y. Chang, A. Podgurski, and J. Yang. Finding what's not there: a new approach to revealing neglected conditions in software. In *ISSTA'07*, pages 163–173.

[39] E. Charniak. Statistical Language Learning. In *First MIT Press paperback edition*. MIT Press, 1996.

[40] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer. Smote: synthetic minority oversampling technique. *Journal of Artificial Intelligence Research*, 16:321–357, 2002.

[41] B. Chen and Z. M. J. Jiang. Characterizing and detecting anti-patterns in the logging code. In *ICSE'17*, pages 71–81.

[42] J. Chen, Y. Bai, D. Hao, Y. Xiong, H. Zhang, and B. Xie. Learning to prioritize test programs for compiler testing. In *ICSE'17*, pages 700–711.

[43] J. Chen, Y. Bai, D. Hao, Y. Xiong, H. Zhang, L. Zhang, and B. Xie. Test case prioritization for compilers: A text-vector based approach. In *ICST'16*, pages 266–277.

[44] J. Chen, W. Hu, L. Zhang, D. Hao, S. Khurshid, and L. Zhang. Learning to accelerate symbolic execution via code transformation. In *ECOOP'18*, volume 109.

[45] T.-H. Chen, S. W. Thomas, M. Nagappan, and A. E. Hassan. Explaining software defects using topic models. In *MSR'12*, pages 189–198.

[46] S. R. Chidamber and C. F. Kemerer. A Metrics Suite for Object Oriented Design. *TSE'94*, 20(6):476–493.

[47] Chollak, Devin. Software bug detection using the n-gram language model. Master's thesis, 2015.

[48] D. Ciresan, U. Meier, and J. Schmidhuber. Multi-column deep neural networks for image classification. In *CVPR'12*, pages 3642–3649.

[49] N. Cliff. *Ordinal methods for behavioral data analysis*. 2014.

[50] J. Czerwonka, R. Das, N. Nagappan, A. Tarvo, and A. Teterev. Crane: Failure prediction, change analysis and test prioritization in practice–experiences from windows. In *ICST'11*, pages 357–366.

[51] H. K. Dam, T. Tran, and A. Ghose. Explainable software analytics. In *ICSE'18: New Ideas and Emerging Results*, pages 53–56.

[52] H. K. Dam, T. Tran, T. Pham, S. W. Ng, J. Grundy, and A. Ghose. Automatic feature learning for vulnerability prediction. *arXiv preprint arXiv:1708.02368*, 2017.

[53] J. Devlin, J. Uesato, S. Bhupatiraju, R. Singh, A.-r. Mohamed, and P. Kohli. Robustfill: Neural program learning under noisy i/o. *arXiv preprint arXiv:1703.07469*, 2017.

[54] H. Do and G. Rothermel. A controlled experiment assessing test case prioritization techniques via mutation faults. In *ICSM'05*, pages 411–420.

[55] H. Do, G. Rothermel, and A. Kinneer. Empirical studies of test case prioritization in a junit testing environment. In *ISSRE'04*, pages 113–124.

[56] F. Dong, J. Wang, Q. Li, G. Xu, and S. Zhang. Defect prediction in android binary executables using deep neural network. *Wireless Personal Communications*, 102(3):2261–2285, 2018.

[57] F. B. e Abreu and R. Carapuça. Candidate metrics for object-oriented software within a taxonomy framework. *JSS'94*, 26(1):87–96.

[58] S. Elbaum, A. Malishevsky, and G. Rothermel. Incorporating varying test costs and fault severities into test case prioritization. In *ICSE'01*, pages 329–338.

[59] S. Elbaum, G. Rothermel, and J. Penix. Techniques for improving regression testing in continuous integration development environments. In *FSE'14*, pages 235–245.

[60] K. O. Elish and M. O. Elish. Predicting defect-prone software modules using support vector machines. *JSS'08*, 81(5):649–660.

[61] E. Engstrom, P. Runeson, and G. Wikstrand. An empirical evaluation of regression testing based on fix-cache recommendations. In *ICST'10*, pages 75–78.

[62] M. Fagan. Design and code inspections to reduce errors in program development. In *Software pioneers*, pages 575–607. 2002.

[63] Y. Fan, X. Xia, D. Lo, and S. Li. Early prediction of merged code changes to prioritize reviewing tasks. *Empirical Software Engineering*, pages 1–48, 2018.

[64] W. B. Frakes and R. Baeza-Yates. Information retrieval: data structures and algorithms. 1992.

[65] J. Friedman, T. Hastie, and R. Tibshirani. *The elements of statistical learning*, volume 1. Springer series in statistics New York, NY, USA:, 2001.

[66] Y. Fu, M. Yan, X. Zhang, L. Xu, D. Yang, and J. D. Kymer. Automated classification of software change messages by semi-supervised latent dirichlet allocation. *IST'15*, 57:369–377.

[67] N. Gayatri, S. Nickolas, A. Reddy, S. Reddy, and A. Nickolas. Feature selection using decision tree induction in class level metrics dataset for software defect predictions. In *WCECS'10*, pages 124–129.

[68] Ç. E. GEREDE and Z. MAZAN. Will it pass? predicting the outcome of a source code review. *Turkish Journal of Electrical Engineering & Computer Sciences*, 26(3):1343–1353, 2018.

[69] M. Gligoric, L. Eloussi, and D. Marinov. Practical regression test selection with dynamic file dependencies. In *ISSTA'15*, pages 211–222.

[70] N. Gruska, A. Wasylkowski, and A. Zeller. Learning from 6,000 Projects: Lightweight Cross-Project Anomaly Detection. In *ISSTA'10*, pages 119–130.

[71] X. Gu, H. Zhang, and S. Kim. Deep code search. In *ICSE'18*, pages 933–944.

[72] X. Gu, H. Zhang, D. Zhang, and S. Kim. Deep api learning. In *FSE'16*, pages 631–642.

[73] B. Guo and M. Song. Interactively decomposing composite changes to support code review and regression testing. In *COMPSAC'17*, volume 1, pages 118–127.

[74] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten. The weka data mining software: an update. *ACM SIGKDD explorations newsletter*, 11(1):10–18, 2009.

[75] M. H. Halstead. *Elements of Software Science (Operating and programming systems series)*. Elsevier Science Inc., 1977.

[76] K. Hamasaki, R. G. Kula, N. Yoshida, A. Cruz, K. Fujiwara, and H. Iida. Who does what during a code review? datasets of oss peer review repositories. In *MSR'13*, pages 49–52.

[77] J. Han and C. Moraga. The influence of the sigmoid function parameters on the speed of backpropagation learning. *From Natural to Artificial Neural Computation*, pages 195–201, 1995.

[78] S. Han, D. R. Wallace, and R. C. Miller. Code Completion from Abbreviated Input. In *ASE'09*, pages 332–343.

[79] Q. Hanam, L. Tan, R. Holmes, and P. Lam. Finding patterns in static analysis alerts: improving actionable alert ranking. In *MSR'14*, pages 152–161.

[80] S. Hangal and M. S. Lam. Tracking Down Software Bugs Using Automatic Anomaly Detection. In *ICSE'02*, pages 291–301.

[81] R. Harrison, S. J. Counsell, and R. V. Nithi. An evaluation of the mood set of object-oriented software metrics. *TSE'98*, 24(6):491–496.

[82] A. E. Hassan. Predicting faults using the complexity of code changes. In *ICSE'09*, pages 78–88.

[83] H. He and E. A. Garcia. Learning from imbalanced data. *TKDE'09*, 21(9):1263–1284.

[84] Z. He, F. Peters, T. Menzies, and Y. Yang. Learning from open-source projects: An empirical study on defect prediction. In *ESEM'13*, pages 45–54.

[85] V. J. Hellendoorn, P. T. Devanbu, and A. Bacchelli. Will They Like This?: Evaluating Code Contributions with Language Models. In *MSR'15*, pages 157–167.

[86] C. Henard, M. Papadakis, M. Harman, Y. Jia, and Y. Le Traon. Comparing white-box and black-box test prioritization. In *ICSE'16*, pages 523–534.

[87] K. Herzig, S. Just, and A. Zeller. It's not a bug, it's a feature: how onlinelassification impacts bug prediction. In *ICSE'13*, pages 392–401.

[88] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu. On the Naturalness of Software. In *ICSE'12*, pages 837–847.

[89] A. Hindle, D. M. German, M. W. Godfrey, and R. C. Holt. Automatic classication of large changes into maintenance categories. In *ICPC'09*, pages 30–39.

[90] A. Hindle, D. M. German, and R. Holt. What do large commits tell us?: a taxonomical study of large commits. In *MSR'08*, pages 99–108.

[91] A. Hindle, M. W. Godfrey, and R. C. Holt. Release pattern discovery via partitioning: Methodology and case study. In *MSR'07*, page 19.

[92] G. E. Hinton, S. Osindero, and Y.-W. Teh. A fast learning algorithm for deep belief nets. *Neural computation'06*, 18(7):1527–1554.

[93] G. E. Hinton and R. R. Salakhutdinov. Reducing the dimensionality of data with neural networks. *Science'06*, 313(5786):504–507.

[94] T. V.-D. Hoang, R. J. Oentaryo, T.-D. B. Le, and D. Lo. Network-clustered multi-modal bug localization. *TSE'18*.

[95] D. Hovemeyer and W. Pugh. Finding bugs is easy. *OOPSLA'04*, 39(12):92–106.

[96] C.-H. Hsiao, M. Cafarella, and S. Narayanasamy. Using Web Corpus Statistics for Program Analysis. In *OOPSLA'14*, pages 49–65.

[97] X. Hu, G. Li, X. Xia, D. Lo, and Z. Jin. Deep code comment generation. In *ICPC'18*, pages 200–210.

[98] Y. Huang, Q. Zheng, X. Chen, Y. Xiong, Z. Liu, and X. Luo. Mining version control system for automatically generating commit comment. In *ESEM'17*, pages 414–423.

[99] V. Jagannath, Q. Luo, and D. Marinov. Change-aware preemption prioritization. In *ISSTA'11*, pages 133–143.

[100] G. Jeong, S. Kim, and T. Zimmermann. Improving bug triage with bug tossing graphs. In *FSE'09*, pages 111–120.

[101] B. Jiang and W. Chan. Bypassing code coverage approximation limitations via effective input-based randomized test case prioritization. In *COMPSAC'13*, pages 190–199.

[102] B. Jiang, Z. Zhang, W. K. Chan, and T. Tse. Adaptive random test case prioritization. In *ASE'09*, pages 233–244.

[103] L. Jiang, G. Misherghi, Z. Su, and S. Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *ICSE'07*, pages 96–105.

[104] T. Jiang, L. Tan, and S. Kim. Personalized defect prediction. In *ASE'13*, pages 279–289.

[105] Y. Jiang, B. Adams, and D. M. German. Will my patch make it? and how fast?: Case study on the linux kernel. In *MSR'13*, pages 101–110.

[106] M. Jimenez, M. Cordy, Y. Le Traon, and M. Papadakis. On the impact of tokenizer and parameters on n-gram based code analysis. 2018.

[107] G. Jin, C. Jinghui, and C.-H. Jane. Semantically enhanced software traceability using deep learning techniques. In *ICSE'17*, pages 3–14.

[108] X. Jing, F. Wu, X. Dong, F. Qi, and B. Xu. Heterogeneous cross-company defect prediction by unified metric representation and cca-based transfer learning. In *FSE'15*, pages 496–507.

[109] X.-Y. Jing, S. Ying, Z.-W. Zhang, S.-S. Wu, and J. Liu. Dictionary learning based software defect prediction. In *ICSE'14*, pages 414–423.

[110] W. Junjie, C. Qiang, W. Song, and W. Qing. Domain adaptation for test report classification in crowdsourced testing. In *ICSE'17*, pages 83–92.

[111] M. Jureczko and L. Madeyski. Towards identifying software project clusters with regard to defect prediction. In *PROMISE'10*, page 9.

[112] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, and G. Fraser. Are mutants a valid substitute for real faults in software testing? In *FSE'14*, pages 654–665.

[113] H. Kagdi, M. L. Collard, and J. I. Maletic. An Approach to Mining Call-Usage Patterns with Syntactic Context. In *ASE'07*, pages 457–460.

[114] Y. Kamei, T. Fukushima, S. McIntosh, K. Yamashita, N. Ubayashi, and A. E. Hassan. Studying just-in-time defect prediction using cross-project models. *Empirical Software Engineering*, 21(5):2072–2106, 2016.

[115] Y. Kamei, E. Shihab, B. Adams, A. E. Hassan, A. Mockus, A. Sinha, and N. Ubayashi. A large-scale empirical study of just-in-time quality assurance. *TSE'13*, 39(6):757–773.

[116] A. Karpathy, J. Johnson, and L. Fei-Fei. Visualizing and understanding recurrent networks. In *ICLR'16 Workshop*.

[117] W. M. Khoo, A. Mycroft, and R. Anderson. Rendezvous: A Search Engine for Binary Code. In *MSR'13*, pages 329–338.

[118] T. Khoshgoftaar and N. Seliya. Tree-based software quality estimation models for fault prediction. In *Software Metrics'02*, pages 203–214.

[119] D. Kim, J. Nam, J. Song, and S. Kim. Automatic patch generation learned from human-written patches. In *ICSE'13*, pages 802–811.

[120] J.-H. Kim. Estimating classification error rate: Repeated cross-validation, repeated hold-out and bootstrap. *Computational statistics & data analysis*, 53(11):3735–3745, 2009.

[121] M. Kim, J. Nam, J. Yeon, S. Choi, and S. Kim. REMI: defect prediction for efficient api testing. In *FSE'15*, pages 990–993.

[122] S. Kim, E. J. Whitehead Jr, and Y. Zhang. Classifying software changes: Clean or buggy? *TSE'08*, 34(2):181–196.

[123] S. Kim, H. Zhang, R. Wu, and L. Gong. Dealing with noise in defect prediction. In *ICSE'11*, pages 481–490.

[124] S. Kim, T. Zimmermann, K. Pan, and E. J. J. Whitehead. Automatic identification of bug-introducing changes. In *ASE'06*, pages 81–90.

[125] S. Kim, T. Zimmermann, E. J. Whitehead Jr, and A. Zeller. Predicting faults from cached history. In *ICSE'07*, pages 489–498.

[126] B. Kitchenham, E. Mendes, and G. H. Travassos. Cross versus within-company cost estimation studies: A systematic review. *TSE'07*, 33(5):316–329.

[127] A. G. Koru and H. Liu. Building effective defect-prediction models in practice. *IEEE software*, 22(6):23–29, 2005.

[128] P. Kreutzer, G. Dotzler, M. Ring, B. M. Eskofier, and M. Philippsen. Automatic clustering of code changes. In *MSR'16*, pages 61–72.

[129] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems'12*, pages 1097–1105.

[130] M. Laali, H. Liu, M. Hamilton, M. Spichkova, and H. W. Schmidt. Test case prioritization using online fault detection information. In *21th Ada-Europe International Conference on Reliable Software Technologies*, pages 78–93, 2016.

[131] O. Laitenberger. A survey of software inspection technologies. In *Handbook of Software Engineering and Knowledge Engineering: Volume II: Emerging Technologies*, pages 517–555. 2002.

[132] S. Lal and A. Sureka. A Static Technique for Fault Localization Using Character N-gram Based Information Retrieval Model. In *ISEC'12*, pages 109–118.

[133] A. Lam, A. Nguyen, H. Nguyen, and T. Nguyen. Combining deep learning with information retrieval to localize buggy files for bug reports. In *ASE'15*, pages 476–481.

[134] J. Lawall and D. Lo. An Automated Approach for Finding Variable-constant Pairing Bugs. In *ASE'10*, pages 103–112.

[135] T.-D. B. Le, R. J. Oentaryo, and D. Lo. Information retrieval and spectrum based bug localization: better together. In *FSE'15*, pages 579–590.

[136] C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *ICSE'12*, pages 3–13.

[137] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer. Genprog: A generic method for automatic software repair. *TSE'12*, 38(1):54.

[138] T. Lee, J. Nam, D. Han, S. Kim, and H. P. In. Micro interaction metrics for defect prediction. In *FSE'11*, pages 311–321.

[139] J. Li, P. He, J. Zhu, and M. R. Lyu. Software defect prediction via convolutional neural network. In *QRS'17*, pages 318–328.

[140] L. Li, H. Feng, W. Zhuang, N. Meng, and B. Ryder. Cclearner: A deep learning-based clone detection approach. In *ICSME'17*, pages 249–260.

[141] X. Li, Y. Liang, H. Qian, Y.-Q. Hu, L. Bu, Y. Yu, X. Chen, and X. Li. Symbolic execution of complex program driven by machine learning based constraint solving. In *ASE'16*, pages 554–559.

[142] Z. Li, M. Harman, and R. M. Hierons. Search algorithms for regression test case prioritization. *TSE'07*, 33(4):225–237.

[143] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. CP-Miner: A Tool for Finding Copy-paste and Related Bugs in Operating System Code. In *OSDI'04*, pages 20–20.

[144] Z. Li and Y. Zhou. PR-Miner: Automatically Extracting Implicit Programming Rules and Detecting Violations in Large Software Code. In *FSE'05*, pages 306–315, 2005.

[145] Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng, and Y. Zhong. Vuldeepecker: A deep learning-based system for vulnerability detection. *arXiv preprint arXiv:1801.01681*, 2018.

[146] B. Liang, P. Bian, Y. Zhang, W. Shi, W. You, and Y. Cai. AntMiner: Mining More Bugs by Reducing Noise Interference. In *ICSE'16*, pages 333–344.

[147] J. Liang, S. Elbaum, and G. Rothermel. Redefining prioritization: continuous prioritization for continuous integration. In *ICSE'18*, pages 688–698.

[148] M. Linares-Vásquez, M. White, C. Bernal-Cárdenas, K. Moran, and D. Poshyvanyk. Mining android app usages for generating actionable gui-based execution scenarios. In *MSR'15*, pages 111–122.

[149] P. Liu, X. Zhang, M. Pistoia, Y. Zheng, M. Marques, and L. Zeng. Automatic text input generation for mobile testing. In *ICSE'17*, pages 643–653.

[150] Y. Liu, D. Poshyvanyk, R. Ferenc, T. Gyimóthy, and N. Chrisochoides. Modeling class cohesion as mixtures of latent topics. In *ICSM'09*, pages 233–242.

[151] F. Long, P. Amidon, and M. Rinard. Automatic inference of code transforms for patch generation. In *FSE'17*, pages 727–739.

[152] F. Long and M. Rinard. Staged program repair with condition synthesis. In *FSE'15*, pages 166–178.

[153] Y. Lu, Y. Lou, S. Cheng, L. Zhang, D. Hao, Y. Zhou, and L. Zhang. How does regression test prioritization perform in real-world software evolution? In *ICSE'16*, pages 535–546.

[154] Q. Luo, K. Moran, and D. Poshyvanyk. A large-scale empirical comparison of static and dynamic test case prioritization techniques. In *FSE'16*, pages 559–570.

[155] Q. Luo, K. Moran, D. Poshyvanyk, and M. Di Penta. Assessing test case prioritization on real faults and mutants. *arXiv preprint arXiv:1807.08823*, 2018.

[156] S. Ma, Y. Liu, W.-C. Lee, X. Zhang, and A. Grama. Mode: Automated neural network model debugging via state differential analysis and input selection. In *FSE'18*.

[157] L. MacLeod, M. Greiler, M.-A. Storey, C. Bird, and J. Czerwonka. Code reviewing in the trenches: Understanding challenges and best practices. *IEEE Software*, 2017.

[158] C. D. Manning and H. Schütze. *Foundations of Statistical Natural Language Processing*. MIT press, 1999.

[159] K. Mao, M. Harman, and Y. Jia. Sapienz: multi-objective automated testing for android applications. In *ISSTA'16*, pages 94–105.

[160] F. Martin et al. *Refactoring: Improving the Design of Existing Code*. 1999.

[161] R. Martin. Oo design quality metrics. *An analysis of dependencies*, 12:151–170, 1994.

[162] V. Mashayekhi, J. M. Drake, W.-T. Tsai, and J. Riedl. Distributed, collaborative software inspection. *IEEE software*, 10(5):66–75, 1993.

[163] T. J. McCabe. A complexity measure. *TSE'76*, (4):308–320.

[164] S. McIntosh, Y. Kamei, B. Adams, and A. E. Hassan. The impact of code review coverage and code review participation on software quality: A case study of the qt, vtk, and itk projects. In *MSR'14*, pages 192–201.

[165] S. McIntosh, Y. Kamei, B. Adams, and A. E. Hassan. An empirical study of the impact of modern code review practices on software quality. *Empirical Software Engineering*, 21(5):2146–2189, 2016.

[166] H. Mei, D. Hao, L. Zhang, L. Zhang, J. Zhou, and G. Rothermel. A static approach to prioritizing junit test cases. *TSE'12*, 38(6):1258–1275.

[167] A. Memon, Z. Gao, B. Nguyen, S. Dhanda, E. Nickell, R. Siemborski, and J. Micco. Taming google-scale continuous testing. In *ICSE'17*, pages 233–242.

[168] A. Meneely, L. Williams, W. Snipes, and J. Osborne. Predicting failures with developer networks and social network analysis. In *FSE'08*, pages 13–23.

[169] T. Menzies, J. Greenwald, and A. Frank. Data mining static code attributes to learn defect predictors. *TSE'07*, 33(1):2–13.

[170] T. Menzies, Z. Milton, B. Turhan, B. Cukic, Y. Jiang, and A. Bener. Defect prediction from static code features: current results, limitations, new approaches. *ASE'10*, 17(4):375–407.

[171] T. Mikolov, I. Sutskever, K. Chen, G. Corrado, and J. Dean. Distributed representations of words and phrases and their compositionality. In *NIPS'13*, pages 3111–3119.

[172] J. Miller, M. Wood, and M. Roper. Further experiences with scenarios and checklists. *Empirical Software Engineering*, 3(1):37–64, 1998.

[173] B. Miranda and A. Bertolino. Does code coverage provide a good stopping rule for operational profile based testing? In *AST'16*, pages 22–28.

[174] B. Miranda and A. Bertolino. Scope-aided test prioritization, selection and minimization for software reuse. *JSS'16*, pages 1–22.

[175] S. Mirarab and L. Tahvildari. A prioritization approach for software test cases on bayesian networks. *FASE'07*, pages 4422–0276.

[176] N. Mirzaei, H. Bagheri, R. Mahmood, and S. Malek. Sig-droid: Automated system input generation for android applications. In *ISSRE'15*, pages 461–471.

[177] A. Mockus and L. G. Votta. Identifying reasons for software changes using historic databases. In *ICSM'00*, page 120.

[178] A. Mockus and D. M. Weiss. Predicting risk of software changes. *Bell Labs Technical Journal*, 5(2):169–180, 2000.

[179] N. Moha, Y.-G. Gueheneuc, L. Duchien, and A.-F. Le Meur. Decor: A method for the specification and detection of code and design smells. *TSE'10*, 36(1):20–36.

[180] A.-r. Mohamed, G. E. Dahl, and G. Hinton. Acoustic modeling using deep belief networks. *Audio, Speech, and Language Processing, IEEE Transactions on*, 20(1):14–22, 2012.

[181] M. Monperrus. Automatic software repair: a bibliography. *CSUR'18*, 51(1):17.

[182] R. Morales, S. McIntosh, and F. Khomh. Do code review practices impact design quality? a case study of the qt, vtk, and itk projects. In *SANER'15*, pages 171–180.

[183] R. Moser, W. Pedrycz, and G. Succi. A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In *ICSE'08*, pages 181–190.

[184] M. Motwani, S. Sankaranarayanan, R. Just, and Y. Brun. Do automated program repair techniques repair hard and important bugs? *Empirical Software Engineering*, 23(5):2901–2947, 2018.

[185] L. Mou, G. Li, Z. Jin, L. Zhang, and T. Wang. Tbcnn: A tree-based convolutional neural network for programming language processing. *Unpublished manuscript: http://arxiv. org/abs/1409.5718*, 2014.

[186] D. Movshovitz-Attias and W. W. Cohen. Natural Language Models for Predicting Programming Comments. In *ACL'13*, pages 35–40.

[187] G. Murphy and D. Cubranic. Automatic bug triage using text categorization. In *SEKE'04*.

[188] G. J. Myers, C. Sandler, and T. Badgett. *The art of software testing.* 2011.

[189] N. Nagappan and T. Ball. Using software dependencies and churn metrics to predict field failures: An empirical case study. In *ESEM'07*, pages 364–373.

[190] N. Nagappan, T. Ball, and A. Zeller. Mining metrics to predict component failures. In *ICSE'06*, pages 452–461.

[191] J. Nam, W. Fu, S. Kim, T. Menzies, and L. Tan. Heterogeneous defect prediction. *TSE'17*, 44(9):874–896.

[192] J. Nam and S. Kim. Clami: Defect prediction on unlabeled datasets. In *ASE'15*, pages 452–463.

[193] J. Nam and S. Kim. Heterogeneous defect prediction. In *FSE'15*, pages 508–519.

[194] J. Nam, S. J. Pan, and S. Kim. Transfer defect learning. In *ICSE'13*, pages 382–391.

[195] G. Navarro. A guided tour to approximate string matching. *ACM Comput. Surv.*, 33(1), 2001.

[196] S. Nessa, M. Abedin, E. Wong, L. Khan, and Y. Qi. Software Fault Localization Using N-gram Analysis. In *WASA'08*, pages 548–559.

[197] A. T. Nguyen, T. T. Nguyen, T. N. Nguyen, D. Lo, and C. Sun. Duplicate bug report detection with a combination of information retrieval and topic modeling. In *ASE'12*, pages 70–79.

[198] T. T. Nguyen, A. T. Nguyen, H. A. Nguyen, and T. N. Nguyen. A Statistical Semantic Language Model for Source Code. In *FSE'13*, pages 532–542.

[199] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. M. Al-Kofahi, and T. N. Nguyen. Graph-based mining of multiple object usage patterns. In *FSE'09*, pages 383–392.

[200] T. T. Nguyen, T. N. Nguyen, and T. M. Phuong. Topic-based defect prediction. In *ICSE'11*, pages 932–935.

[201] T. B. Noor and H. Hemmati. A similarity-based approach for test case prioritization using historical failure data. In *ISSRE'15*, pages 58–68.

[202] Y. Oda, H. Fudaba, G. Neubig, H. Hata, S. Sakti, T. Toda, and S. Nakamura. Learning to Generate Pseudo-code from Source Code Using Statistical Machine Translation. In *ASE'15*, pages 824—829.

[203] T. J. Ostrand and E. J. Weyuker. The distribution of faults in a large industrial software system. In *FSE'07*, pages 55–64.

[204] T. J. Ostrand, E. J. Weyuker, and R. M. Bell. Programmer-based fault prediction. In *PROMISE'10*, pages 19:1–19:10.

[205] T. J. Ostrand, E. J. Weyuker, and R. M. Bell. Predicting the location and number of faults in large software systems. *TSE'05*, 31(4):340–355, 2005.

[206] E. Paikari, M. M. Richter, and G. Ruhe. Defect prediction using case-based reasoning: An attribute weighting technique based upon sensitivity analysis in neural networks. *SEKE'12*.

[207] F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, A. De Lucia, and D. Poshyvanyk. Detecting bad smells in source code using change history information. In *ASE'13*, pages 268–278.

[208] S. J. Pan, I. Tsang, J. Kwok, and Q. Yang. Domain adaptation via transfer component analysis. *Neural Networks, IEEE Transactions on*, pages 199–210, 2011.

[209] R. Pascanu, J. W. Stokes, H. Sanossian, M. Marinescu, and A. Thomas. Malware classification with recurrent networks. In *ICASSP'15*, pages 1916–1920.

[210] H. Perl, S. Dechand, M. Smith, D. Arp, F. Yamaguchi, K. Rieck, S. Fahl, and Y. Acar. Vccfinder: Finding potential vulnerabilities in open-source projects t o assist code audits. In *CCS'15*, pages 426–437.

[211] M. Pinzger, N. Nagappan, and B. Murphy. Can developer-module networks predict failures? In *FSE'08*, pages 2–12.

[212] A. Porter, H. Siy, and L. Votta. A review of software inspections. *Advances in Computers*, 42:39–76, 1996.

[213] M. Pradel and T. R. Gross. Automatic Generation of Object Usage Specifications from Large Method Traces. In *ASE'09*, pages 371–382.

[214] M. Pradel and K. Sen. Deepbugs: A learning approach to name-based bug detection. *arXiv preprint arXiv:1805.11683*, 2018.

[215] L. Prechelt and A. Pepper. Why software repositories are not used for defect-insertion circumstanc e analysis more often: A case study. *IST'14*, 56(10):1377–1389.

[216] J. C. Pruessner, C. Kirschbaum, G. Meinlschmid, and D. H. Hellhammer. Two formulas for computation of the area under the curve represent measures of total hormone concentration versus time-dependent change. *Psychoneuroendocrinology*, 28(7):916–931, 2003.

[217] R. Purushothaman and D. E. Perry. Toward understanding the rhetoric of small source code changes. *TSE'05*, 31(6):511–526.

[218] T.-S. Quah and M. M. T. Thwin. Application of neural networks for software quality prediction using object-oriented metrics. In *ICSM'03*, pages 116–125.

[219] A. Radford, R. Jozefowicz, and I. Sutskever. Learning to generate reviews and discovering sentiment. *arXiv preprint arXiv:1704.01444*, 2017.

[220] F. Rahman and P. Devanbu. How, and why, process metrics are better. In *ICSE'13*, pages 432–441.

[221] F. Rahman, S. Khatri, E. T. Barr, and P. Devanbu. Comparing static bug finders and statistical prediction. In *ICSE'14*, pages 424–434.

[222] F. Rahman, D. Posnett, and P. Devanbu. Recalling the "imprecision" of cross-project defect prediction. In *FSE'12*, pages 61:1–61:11.

[223] M. K. Ramanathan, A. Grama, and S. Jagannathan. Path-Sensitive Inference of Function Precedence Protocols. In *ICSE'07*, pages 240–250.

[224] B. Ray, V. Hellendoorn, Z. Tu, C. Nguyen, S. Godhane, A. Bacchelli, and P. Devanbu. On the "Naturalness" of Buggy Code. In *ICSE'16*, pages 428–439, 2016.

[225] V. Raychev, M. Vechev, and E. Yahav. Code Completion with Statistical Language Models. In *PLDI'14*, pages 419–428.

[226] M. Renieres and S. P. Reiss. Fault localization with nearest neighbor queries. In *ASE'03*, pages 30–39.

[227] P. C. Rigby. Open source peer review–lessons and recommendations for closed source. 2012.

[228] P. C. Rigby, D. M. German, and M.-A. Storey. Open source software peer review practices: a case study of the apache server. In *ICSE'08*, pages 541–550.

[229] P. C. Rigby and M.-A. Storey. Understanding broadcast based peer review on open source software projects. In *ICSE'11*, pages 541–550.

[230] A. Rimsa, M. d' Amorim, F. M. Q. Pereira, and R. S. Bigonha. Efficient static checker for tainted variable attacks. *Science of Computer Programming*, 80:91–105, 2014.

[231] R. Rosenfield. Two Decades of Statistical Language Modeling: Where Do We Go from Here? 2000.

[232] G. Rothermel and M. J. Harrold. A safe, efficient regression test selection technique. *TOSEM'97*, 6(2):173–210.

[233] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold. Prioritizing test cases for regression testing. *TSE'11*, 27(10):929–948.

[234] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold. Test case prioritization: An empirical study. In *ICSM'99*, pages 179–188.

[235] R. K. Saha, L. Zhang, S. Khurshid, and D. E. Perry. An information retrieval approach for regression test prioritization based on program changes. In *ICSE'15*, pages 268–279.

[236] M. Sampson, L. Zhang, A. Morrison, N. J. Barrowman, T. J. Clifford, R. W. Platt, T. P. Klassen, and D. Moher. An Alternative to the Hand Searching Gold Standard: Validating Methodological Search Filters Using Relative Recall. *BMC Medical Research Methodology*, 6(1), 2006.

[237] A. L. Santos, G. Prendi, H. Sousa, and R. Ribeiro. Stepwise api usage assistance using n-gram language models. *JSS'17*, 131:461–474.

[238] R. Sarikaya, G. E. Hinton, and A. Deoras. Application of deep belief networks for natural language understanding. *TASLP'14*, 22(4):778–784.

[239] F. Seide, G. Li, and D. Yu. Conversational speech transcription using context-dependent deep neural networks. In *INTERSPEECH'11*.

[240] A. Shi, A. Gyori, M. Gligoric, A. Zaytsev, and D. Marinov. Balancing trade-offs in test-suite reduction. In *FSE'14*, pages 246–256.

[241] A. Shi, T. Yung, A. Gyori, and D. Marinov. Comparing and combining test-suite reduction and regression test selection. In *FSE'15*, pages 237–247.

[242] T. Shippey, D. Bowes, and T. Hall. Automatically identifying code features for software defect prediction: Using ast n-grams. *Information and Software Technology*, 2018.

[243] S. Shoham, E. Yahav, S. Fink, and M. Pistoia. Static Specification Mining Using Automata-Based Abstractions. In *ISSTA'07*, pages 174–184.

[244] M. S. Siddik and K. Sakib. Rdcc: An effective test case prioritization framework using software requirements, design and source code collaboration. In *ICCIT'14*, pages 75–80.

[245] J. Śliwerski, T. Zimmermann, and A. Zeller. When do changes induce fixes? In *MSR'05*, pages 1–5.

[246] D. Spadini, M. Aniche, M.-A. Storey, M. Bruntink, and A. Bacchelli. When testing meets code review: Why and how developers review tests. In *ICSE'18*, pages 677–687.

[247] A. Srivastava and J. Thiagarajan. Effectively prioritizing tests in development environment. In *ISSTA'02*, pages 97–106.

[248] M. Stein, J. Riedl, S. J. Harner, and V. Mashayekhi. A case study of distributed, asynchronous software inspection. In *ICSE'97*, pages 107–117.

[249] T. Su, G. Meng, Y. Chen, K. Wu, W. Yang, Y. Yao, G. Pu, Y. Liu, and Z. Su. Guided, stochastic model-based gui testing of android apps. In *FSE'17*, pages 245–256.

[250] B. Sun, G. Shu, A. Podgurski, and B. Robinson. Extending Static Analysis by Mining Project-specific Rules. In *ICSE'12*, pages 1054–1063.

[251] A. Sureka and P. Jalote. Detecting Duplicate Bug Report Using Character N-gram-based Features. In *APSEC'10*, pages 366–374.

[252] A. Sutherland and G. Venolia. Can peer code reviews be exploited for later information needs? In *ICSE-Companion'09*, pages 259–262.

[253] E. B. Swanson. The dimensions of maintenance. In *ICSE'76*, pages 492–497.

[254] A. Tamrawi, T. T. Nguyen, J. M. Al-Kofahi, and T. N. Nguyen. Fuzzy set and cache-based approach for bug triaging. In *FSE'11*, pages 365–375.

[255] L. Tan, D. Yuan, G. Krishna, and Y. Zhou. /* iComment: Bugs or Bad Comments? */. In *SOSP'07*, pages 145–158.

[256] M. Tan, L. Tan, S. Dara, and C. Mayeux. Online defect prediction for imbalanced data. In *ICSE'15*, pages 99–108.

[257] S. H. Tan, D. Marinov, L. Tan, and G. T. Leavens. @tComment: Testing Javadoc Comments to Detect Comment-code Inconsistencies. In *ICST'12*, pages 260–269.

[258] X. Tang, S. Wang, and K. Mao. Will this bug-fixing change break regression testing? In *ESEM'15*, pages 1–10.

[259] C. Tantithamthavorn, A. E. Hassan, and K. Matsumoto. The impact of class rebalancing techniques on the performance and interpretation of defect prediction models. *arXiv preprint arXiv:1801.10269*, 2018.

[260] C. Tantithamthavorn, S. McIntosh, A. Hassan, and K. Matsumoto. An empirical comparison of model validation techniques for defect prediction models. *TSE'16*, 43:1–18.

[261] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, A. Ihara, and K. ichi Matsumoto. The impact of mislabelling on the performance and interpretation of defect prediction models. In *ICSE'15*, pages 812–823.

[262] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, and K. Matsumoto. Automated parameter optimization of classification techniques for defect prediction models. In *ICSE'16*, pages 321–332.

[263] W. Tao and L. Wei-hua. Naive bayes software defect prediction model. In *CiSE'10*, pages 1–4.

[264] Y. Tao, Y. Dang, T. Xie, D. Zhang, and S. Kim. How do software engineers understand code changes?: an exploratory study in industry. In *FSE'12*, page 51.

[265] Y. Tao and S. Kim. Partitioning composite code changes to facilitate code review. In *MSR'15*, pages 180–190.

[266] S. W. Thomas, H. Hemmati, A. E. Hassan, and D. Blostein. Static test case prioritization using topic models. *EMSE'14*, 19(1):182–212.

[267] P. Thongtanunam, C. Tantithamthavorn, R. G. Kula, N. Yoshida, H. Iida, and K.-i. Matsumoto. Who should review my code? a file location-based code-reviewer recommendation approach for modern code review. In *SANER'15*, pages 141–150.

[268] S. Thummalapenta and T. Xie. Mining Exception-Handling Rules as Sequence Association Rules. In *ICSE'09*, pages 496–506.

[269] S. Thummalapenta and T. Xie. Alattin: Mining Alternative Patterns for Defect Detection. *Automated Software Engineering*, 18(3-4):292–323, 2011.

[270] P. Tonella, P. Avesani, and A. Susi. Using the case-based ranking methodology for test case prioritization. In *ICSM'06*, pages 123–133.

[271] Z. Tu, Z. Su, and P. Devanbu. On the Localness of Software. In *FSE'14*, pages 269–280.

[272] B. Turhan, T. Menzies, A. B. Bener, and J. Di Stefano. On the relative value of cross-company and within-company data for defect prediction. *Empirical Software Engineering*, 14(5):540–578, 2009.

[273] S. Varun Kumar and M. Kumar. Test case prioritization using fault severity. *IJCST'10*, 1(1).

[274] L. G. Votta Jr. Does every inspection need a meeting? *FSE'93*, 18(5):107–114.

[275] F. Wang, T.-T. Quach, J. Wheeler, J. B. Aimone, and C. D. James. Sparse coding for n-gram feature extraction and training for file fragment classification. *TSE'18*, 13(10):2553–2562.

[276] J. Wang, B. Shen, and Y. Chen. Compressed c4. 5 models for software defect prediction. In *QSIC'12*, pages 13–16.

[277] J. Wang, S. Wang, and Q. Wang. Is there a golden feature set for static warning identification?: an experimental evaluation. In *ESEM'18*, page 17.

[278] S. Wang, D. Chollak, D. Movshovitz-Attias, and L. Tan. Bugram: Bug detection with n-gram language models. In *ASE'16*, pages 708–719.

[279] S. Wang, T. Liu, J. Nam, and L. Tan. Deep semantic feature learning for software defect prediction. *TSE'18*.

[280] S. Wang, T. Liu, and L. Tan. Automatically learning semantic features for defect prediction. In *ICSE'16*, pages 297–308.

[281] S. Wang, J. Nam, and L. Tan. QTEP: Qality-aware test case prioritization. In *FSE'17*, pages 523–534.

[282] S. Wang, W. Zhang, and Q. Wang. Fixercache: Unsupervised caching active developers for diverse bug triage. In *ESEM'14*, page 25.

[283] S. Wang, W. Zhang, Y. Yang, and Q. Wang. Devnet: exploring developer collaboration in heterogeneous networks of bug repositories. In *ESEM'13*, pages 193–202.

[284] A. Wasylkowski and A. Zeller. Mining Temporal Specifications from Object Usage. *Automated Software Engineering*, 18(3-4):263–292, 2011.

[285] A. Wasylkowski, A. Zeller, and C. Lindig. Detecting object usage anomalies. In *FSE'07*, pages 35–44.

[286] S. Watanabe, H. Kaiya, and K. Kaijiri. Adapting a fault prediction model to allow inter languagereuse. In *PROMISE'08*, pages 19–24.

[287] W. Weimer and G. C. Necula. Mining Temporal Specifications for Error Detection. pages 461–476, 2005.

[288] M. Wen, R. Wu, and S.-C. Cheung. How well do change sequences predict defects? sequence learning from software changes. *TSE'18*.

[289] E. J. Weyuker, T. J. Ostrand, and R. M. Bell. Using developer information as a factor for fault prediction. In *PROMISE'07*, page 8.

[290] M. White, C. Vendome, M. Linares-Vásquez, and D. Poshyvanyk. Toward Deep Learning Software Repositories. In *MSR'15*, pages 334–345.

[291] C. Williams and J. Hollingsworth. Automatic Mining of Source Code Repositories to Improve Bug Finding Techniques. In *TSE'05*, volume 31, pages 466–480.

[292] C. C. Williams and J. K. Hollingsworth. Recovering System Specific Rules from Software Repositories. In *MSR'05*, pages 1–5.

[293] I. H. Witten and E. Frank. *Data Mining: Practical machine learning tools and techniques*. Morgan Kaufmann, 2005.

[294] E. Wong, T. Liu, and L. Tan. Clocom: Mining existing source code for automatic comment generation. In *SANER'15*, pages 380–389.

[295] E. Wong, J. Yang, and L. Tan. Autocomment: Mining question and answer sites for automatic comment generation. In *ASE'13*, pages 562–567.

[296] E. Wong, L. Zhang, S. Wang, T. Liu, and L. Tan. Dase: Document-assisted symbolic execution for improving automated software testing. In *ICSE'15*, pages 620–631.

[297] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa. A survey on software fault localization. *TSE'16*, 42(8):707–740.

[298] W. E. Wong and Y. Qi. Bp neural network-based effective fault localization. *SEKEJ'09*, 19(04):573–597.

[299] X. Xia, D. Lo, S. J. Pan, N. Nagappan, and X. Wang. Hydra: Massively compositional model for cross-project defect prediction. *TSE'16*, 42:977–998.

[300] X. Xia, D. Lo, X. Wang, and X. Yang. Collective personalized change classification with multiobjective search. *TR'16*, 65(4):1810–1829.

[301] T. Xie and J. Pei. MAPO: Mining API Usages from Open Source Repositories. In *MSR'06*, pages 54–57.

[302] X. Xie, W. Zhang, Y. Yang, and Q. Wang. Dretom: Developer recommendation based on topic models for bug resolution. In *PROMISE'12*, pages 19–28.

[303] Y. Xiong, X. Liu, M. Zeng, L. Zhang, and G. Huang. Identifying patch correctness in test-based program repair. In *ICSE'18*, pages 789–799.

[304] B. Xu, D. Ye, Z. Xing, X. Xia, G. Chen, and S. Li. Predicting semantically linkable knowledge in developer online forums via convolutional neural network. In *ASE'16*, pages 51–62.

[305] D. Xu, S. Nair, Y. Zhu, J. Gao, A. Garg, L. Fei-Fei, and S. Savarese. Neural task programming: Learning to generalize across hierarchical tasks. *arXiv preprint arXiv:1710.01813*, 2017.

[306] J. Xuan, H. Jiang, Z. Ren, and W. Zou. Developer prioritization in bug repositories. In *ICSE'12*, pages 25–35.

[307] J. Xuan, M. Martinez, F. Demarco, M. Clement, S. L. Marcote, T. Durieux, D. Le Berre, and M. Monperrus. Nopol: Automatic repair of conditional statement bugs in java programs. *TSE'17*, 43(1):34–55.

[308] M. Yan, Y. Fu, X. Zhang, D. Yang, L. Xu, and J. D. Kymer. Automatically classifying software changes via discriminative topic model: Supporting multi-category and cross-project. *JSS'16*, 113:296–308.

[309] J. Yang, A. Zhikhartsev, Y. Liu, and L. Tan. Better test cases for better automated program repair. In *FSE'17*, pages 831–841.

[310] X. Yang, D. Lo, X. Xia, L. Bao, and J. Sun. Combining word embedding with information retrieval to recommend similar bug reports. In *ISSRE'16*, pages 127–137.

[311] X. Yang, D. Lo, X. xia, Y. Zhang, and J. Sun. Deep learning for just-in-time defect prediction. In *QRS'15*, pages 17–26.

[312] S. Yoo and M. Harman. Regression testing minimization, selection and prioritization: a survey. *STVR'12*, 22(2):67–120.

[313] S. Yoo, M. Harman, and D. Clark. Fault localization prioritization: Comparing information-theoretic and coverage-based approaches. *TOSEM'13*, 22(3):19.

[314] S. Yoo, R. Nilsson, and M. Harman. Faster fault finding at google using multi objective regression test optimisation. In *FSE'11*.

[315] Y. T. Yu and M. F. Lau. Fault-based test suite prioritization for specification-based testing. *IST'12*, 54(2):179–202.

[316] Z. Yu, H. Hu, C. Bai, K.-Y. Cai, and W. Wong. GUI Software Fault Localization Using N-gram Analysis. In *HASE'11*, pages 325–332.

[317] Z. Yuan, Y. Lu, Z. Wang, and Y. Xue. Droid-sec: Deep learning in android malware detection. In *SIGCOMM'14*, pages 371–372.

[318] F. Zhang, Q. Zheng, Y. Zou, and A. E. Hassan. Cross-project defect prediction using a connectivity-based unsupervised classifier. In *ICSE'16*, pages 309–320.

[319] L. Zhang, D. Hao, L. Zhang, G. Rothermel, and H. Mei. Bridging the gap between the total and additional test-case prioritization strategies. In *ICSE'13*, pages 192–201.

[320] L. Zhang, M. Kim, and S. Khurshid. Faulttracer: a change impact and regression fault analysis tool for evolving java programs. In *FSE'12*, page 40.

[321] L. Zhang, D. Marinov, L. Zhang, and S. Khurshid. An empirical study of junit test-suite reduction. In *ISSRE'11*, pages 170–179.

[322] L. Zhang, G. Rosenblatt, E. Fetaya, R. Liao, W. E. Byrd, M. Might, R. Urtasun, and R. Zemel. Neural guided constraint logic programming for program synthesis. *arXiv preprint arXiv:1809.02840*, 2018.

[323] M. Zhang, X. Li, L. Zhang, and S. Khurshid. Boosting spectrum-based fault localization using pagerank. In *ISSTA'17*, pages 261–272.

[324] W. Zhang, S. Wang, Y. Yang, and Q. Wang. Heterogeneous network analysis of developer contribution in bug repositories. In *ICCSC'13*, pages 98–105.

[325] T. Zimmermann, N. Nagappan, H. Gall, E. Giger, and B. Murphy. Cross-project defect prediction: a large scale experiment on data vs. domain vs. process. In *FSE'09*, pages 91–100.

[326] T. Zimmermann, R. Premraj, and A. Zeller. Predicting defects for eclipse. In *PROMISE'07*, pages 9–9.