An Empirical Study of Feature Engineering on Software Defect Prediction

A Thesis

Submitted to the Graduate School of Science and Technology

of Kyoto Institute of Technology

for

Doctor of Engineering Degree

by

KONDO Masanari

December 10, 2020

Abstract

Software products are pivotal for our daily life such as infrastructure, work, and communication. Therefore, defects in such software products may cause widespread catastrophes. Indeed, several accidents have been reported whose causes were software defects.

Due to such importance of software products, software developers carefully manage the quality of software products by software quality assurance (SQA) activities (e.g., software testing, code review, and CI/CD). For example, software testing inspects if software products meet all the requirements. However, recently software products have become enormous in size and depend on numerous environments; it is difficult to inspect the entire software products by SQA activities.

Defect prediction distinguishes defective software entities (e.g., file) by a defect prediction model. Such a defect prediction model enables developers to allocate their SQA activities to defective entities and reveal more defects than applying SQA activities without such a model. Hence, defect prediction attracts interests by practitioners and researchers, and becomes an active research area in software engineering.

Defect prediction models are usually machine learning models that are trained on software features of past software entities. Since machine learning models rely on such software features, prior studies used feature engineering on defect prediction to improve the prediction performance. Feature engineering is a process to create or improve features by our domain knowledge. For example, several studies retrieved new features from a software product. However, defect prediction still has challenges that can be addressed by feature engineering: (1) the comparison of feature reduction techniques, (2) using the context lines of source code as features, and (3) using semantic properties as features with a deep learning model on change-level defect prediction.

In this thesis, to address these challenges, we (1) conducted a large-empirical comparison across feature reduction and selection techniques, (2) constructed context features retrieved from context lines, and (3) used semantic properties with a deep learning model on change-level defect prediction. Our results showed that (1) feature reduction and selection techniques improve the prediction performance while reducing the number of features, (2) context features improve the prediction performance, and (3) semantic features with a deep learning model significantly outperform a previous deep learning model.

Acknowledgments

This thesis would not have been possible without the support of many exceptional people to whom I am grateful.

First of all, I would like to thank my supervisor Professor Osamu Mizuno, who was also a member of my thesis committee, for his supervision and continuous support throughout this long journey. Since I was an undergraduate student, he gave me numerous opportunities to grow as a researcher. The experiences in such opportunities make me have a wide perspective. Of course, his technical and editorial advice is fundamental to this thesis. Without his taught, I would never write this thesis. Also, he prepared the research environment that made the research faster and faster. In addition, he taught not only research but also life. I cannot thank him enough for all he has done.

I am also very grateful to the members of my thesis committee, Professor Teruhisa Hochin and Professor Yu Shibuya, for their valuable feedback and their useful comments on this thesis.

I would like to thank Assistant Professor Eunjong Choi. She has also supervised me. She has continuously given me advice on my research. In addition, she provided me many opportunities to discuss with knowledgeable researchers and collaborate with them. Also, she told me some key insights to supervise students. Without her support, I would not write this thesis and would not know how to supervise students.

I would like to thank Dr. Eun-Hye Choi. She gave me valuable feedback on my study and writing when I was a novice in this research field. She kindly told me many things, though I didn't know anything about research and its related stuff. Also, she helped me to attend an international conference. Without her, I would not know what should I do at international conferences.

I would like to express my deepest appreciation to Professor Ahmed E. Hassan. He accepted my visit to his laboratory for over one year in total and gave me many opportunities to collaborate with knowledgeable researchers. This visiting experience enhanced my study, knowledge, and English abilities.

I would like to thank Assistant Professor Cor-Paul Bezemer. Since when I was an undergraduate student, he supported me by giving technical advice and feedback, and discussing my study. He provided key insights into this thesis.

I wish to thank Associate Professor Yasutaka Kamei. He also supported me since when I was an undergraduate student. He, of course, gave me advice, however, he also provided me with opportunities to participate in the software engineering community. Such experiences extend my perspective in this community.

I also would like to thank Dr. Gustavo A. Oliva. He invited me to a new research field. This experience extended my research field. In addition, he discussed with me anytime and anywhere. The collaboration study with him improved my research skill significantly.

I also would like to thank Professor Daniel M. German. He provided me with not only academic perspectives but also practical viewpoints. In addition, he continuously gave me numerous comments on not only this thesis but also other publications.

Special thanks to Professor Ryosuke Kubota. When I was a student at the National Institute of Technology, Ube College, I was under the supervision of him. He showed what is researching and how interesting researching is. Without him, I would not start my career at the university.

I was a very lucky man to work and collaborate with some of the brightest researchers. I would like to thank all of my collaborators, Associate Professor Zhen

Ming (Jack) Jiang, Associate Professor Cyrille Artho, Dr. Muhammad Asaduzzaman, Dr. Gopi Krishnan Rajbahadur, and Lecturer Akinori Ihara. They shared their experiences, offered a lot of advices, and taught writing techniques. The collaboration was invaluable to proceed and polish my work.

I would like to acknowledge Dr. Hideto Ogawa and Dr. Makoto Ichii of Hitachi Ltd. and Dr. Shinobu Saito of NTT Corporation. They gave me various feedback from the actual software development field in conferences.

I have had the privilege of working with amazing groups at the Software Engineering Laboratory (SEL), the Software Analysis and Intelligence Lab (SAIL), and the Software Analytics Research Group. Each member has had an impact on me and my work. I am grateful to all members.

The work in this thesis was published on Empirical Software Engineering and IPSJ Journal. I would like to thank all reviewers and editors. They gave me many insightful comments.

Without the support of my family, this thesis would not have been possible. I am particularly grateful to my parents.

List of Publications that Provide the Foundation of This Thesis

(1) Masanari Kondo, Keita Mori, Osamu Mizuno, and Eun-Hye Choi, "Just-in-Time Defect Prediction Applying Deep Learning to Source Code Changes," IPSJ Journal, Vol. 59, No. 4, pp.1250–1261, 2018 (in Japanese).

(2) Masanari Kondo, Cor-Paul Bezemer, Yasutaka Kamei, Ahmed E. Hassan, and Osamu Mizuno, "The impact of feature reduction techniques on defect prediction models," Empirical Software Engineering, Vol. 24, pp.1925–1963, 2019.

(3) Masanari Kondo, Daniel M. German, Osamu Mizuno, and Eun-Hye Choi, "The impact of context metrics on just-in-time defect prediction," Empirical Software Engineering, Vol. 25, pp.890–939, 2020.

List of Figures

List of Tables

Contents

Chapter 1

Introduction

1.1 Background

Defects in software products currently have severe impacts on our daily life. Indeed, Krasner [86] reported that poor quality software products cost approximately $2.8 trillion in the US in 2018 only. Hence, software developers need to carefully inspect their software products and exclude defects.

To exclude defects, developers apply *software quality assurance (SQA) activities* to software products such as software testing, code review, and CI/CD. For example, software testing is an essential phase in software development [13]. In this phase, developers inspect if their software products meet all the requirements. However, SQA activities are time consuming and expensive in software development [156]; and therefore, developers need to be selective when deciding where to focus their limited SQA effort.

Defect prediction models help developers focus their SQA effort on software entities (e.g., package, file, and commit) that are the most likely to include defects [71,77,82,83]. Such defect prediction models attract interests from researchers and practitioners. Indeed, numerous defect prediction studies exist in software engineering. For example, we conducted a systematic literature review [48,63,80] in the top journals (IEEE Transactions on Software Engineering (TSE), ACM Transactions on Software Engineering and Methodology (TOSEM), and Empirical Software Engineering (EMSE)) and conferences (International Conference on Software Engineering (ICSE), Symposium on the Foundations of Software Engineering (FSE), International Conference on Automated Software Engineering (ASE), International Conference on Software Analysis, Evolution and Reengineering (SANER), and International Conference on Mining Software Repositories (MSR)) in the past three years (Jan. 2018 to Oct. 2020). We searched three digital libraries (IEEE Xplore, ACM Digital Library, and Springer) for the papers that include one of three keywords ("defect prediction", "fault prediction", and "bug prediction"). Finally, we read the abstract and excluded non-defect prediction papers. We collected 52 papers that are shown in Table 1.1. Hence, even in the top journals and conferences in the past three years, there are 52 defect prediction studies.

Defect prediction models are typically constructed by training a machine learning model on *features* that are related to software development. As more informative features improve the prediction performance of such a model, prior studies have been reporting various features so far [9,19,24,28,50,52,54,71,77,85,97,98,110,113,132]. Such features can be classified into some categories such as process features, product features and developer features. For example, one of the suites of the process features is the change features [71,110]. The change features include features that can be retrieved from each commit/change in version control systems. One of the suites of the product features is the object-oriented (OO) features [9,24]. The OO features can be retrieved from software entities written in object-oriented languages such as classes in Java. We discuss the details of the

Table 1.1: All the retrieved papers by our systematic literatuer review.

Venue	Reference
TSE	[158], [30], [114], [11], [173], [57], [100], [170], [161], [123] [165], [152], [56], [58], [179], [36], [130], [93], [120], [138] [21], [166], [67], [172]
TOSEM	[72]
EMSE	[12], [82], [64], [109], [8], [34], [68], [83], [108], [6]
ICSE	[3], [33], [18], [26], [178], [22], [92], [154]
MSR	[27], [62]
ASE	[129], [42]
SANER	[125], [94], [174], [180], [5]

features in Chapter 2.

Whereas using such features is useful in defect prediction, we can also use other features that capture the *semantic properties* of source code to construct defect prediction models [62,66,76,91,105,155,171,176]. Here, we refer to semantic properties as the meaning of source code. For example, Wang et al. [171] converted source code to abstract syntax tree (AST) nodes, and extracted semantic properties with a deep learning model from the AST nodes. The output of the deep learning model is used as their features. Tan et al. [155] used bag-of-words [184] to extract semantic properties. We discuss the details of the semantic properties in Chapter 2.

Because of a variety of features, researchers and practitioners who want to construct defect prediction models need to select an appropriate suite of features for their data. This is because keeping the number of features small avoids some problems such as the problem of multicollinearity [4]. To do so, we can use *feature selection/reduction techniques*. Indeed, several researchers applied feature selection [20, 39, 41, 43, 44, 69, 70, 101, 111, 115, 116, 133, 136, 137, 149, 175, 186] and reduction techniques [20, 28, 41, 112, 117, 119, 133, 134, 175] to defect prediction models. For example, Ghotra et al. [41] compared 30 feature selection techniques on defect prediction models. They found that a correlation-based filter-subset feature selection technique with a BestFirst search method outperformed other feature selection techniques.

Such prior studies aimed to improve the defect prediction performance by investigating features based on their domain knowledge (a.k.a. *feature engineering* [185]), though defect prediction still has challenges that can be addressed by feature engineering. In particular, we found the following three challenges are ignored: (1) the comparison of feature reduction techniques, (2) using the context lines of source code as features, and (3) using semantic properties as features with a deep learning model on change-level defect prediction.

In this thesis, we applied feature engineering to defect prediction to address three remaining challenges. More specifically, we conducted the following investigations:

(1) We conducted an empirical comparison across feature reduction techniques and between feature reduction techniques and feature selection techniques in defect prediction. We found that neural network-based feature reduction techniques outperformed the other feature reduction/selection techniques in terms of the prediction performance on unsupervised defect prediction models, while the best-performing feature selection techniques outperformed the feature reduction techniques on supervised defect prediction models (Chapter 3).

(2) We constructed *context features* and their extensions retrieved from context lines of source code. We found that our extended context features significantly outperformed the other studied features in terms of two out of three evaluation measures. In addition, we found that the prediction performance of context features varies with the parameters of context lines such as the number of context lines (Chapter 4).

(3) We applied a deep learning model (convolutional neural network) to modified source code to retrieve semantic properties. In particular, we constructed a change-level (a.k.a. just-in-time) defect prediction model called *W-CNN*. We found that our W-CNN improved the defect prediction performance compared to a previous just-in-time defect prediction model with a deep learning model (Chapter 5).

1.2 Thesis overview

Here, we described a brief overview of this thesis organization. We described each chapter below.

1.2.1 Features in defect prediction (Chapter 2)

In this chapter, we briefly summarized existing features in defect prediction. In particular, we described features for each feature category: *product features*, *process features*, *organization features*, *developer features*, and *features of semantic properties*. Finally, we clarified three challenges that we described above.

1.2.2 The impact of feature reduction techniques on defect prediction models (Chapter 3)

Feature selection and reduction techniques can help to reduce the number of features in a defect prediction model. Feature selection techniques reduce the number of features in a model by selecting the most important ones, while feature reduction techniques reduce the number of features by creating new, combined features from the original features. Numerous recent studies have investigated the impact of feature *selection* techniques on defect prediction [20, 39, 41, 43, 44, 69, 70, 101, 111, 115, 116, 133, 136, 137, 149, 175, 186]. However, there do not exist large-scale studies in which the impact of multiple feature *reduction* techniques on defect prediction is investigated.

In this chapter, we studied the impact of eight feature reduction techniques on the performance and the variance in performance of five supervised and five unsupervised defect prediction models. In addition, we compared the impact of the studied feature reduction techniques with the impact of the two best-performing feature selection techniques according to prior work [41, 175].

1.2.3 The impact of context features on just-in-time defect prediction (Chapter 4)

Traditional just-in-time defect prediction approaches have been using changed lines of software to predict defective changes in software development. However, they disregard information around the changed lines. Our main hypothesis is that such information has an impact on the likelihood that the change is defective. To take advantage of this information in defect prediction, we considered n-lines ($n = 1, 2, \cdots$) that precede and follow the changed lines (which we call *context lines*), and proposed features that measure them, which we call "*Context Features*." Specifically, these context features are defined as the number of words/keywords in the context lines.

In this chapter, we conducted a large-scale empirical study using six open source software projects. We compared the performance of using our context features, traditional code churn features (e.g., the number of modified subsystems), our *extended context features* which measure not only context lines but also changed lines, and *combination features* that use two extended context features at a prediction model for defect prediction.

1.2.4 Deep learning based just-in-time defect prediction with semantic properties as features (Chapter 5)

Several prior studies used semantic properties of source code as their features to construct a defect prediction model [62,66,76,91,105,155,171,176]. However, they overlooked to use a deep learning model to retrieve semantic properties in just-in-time defect prediction.

In this chapter, we proposed a novel approach for defect prediction called *Word-Convolutional Neural Network (W-CNN)*, which applies a CNN to the modified source code itself. We compared our approach to a prior defect prediction model that uses change features with a deep learning model in seven open-source projects.

1.2.5 Conclusion and future work (Chapter 6)

In this chapter, we concluded the findings and implications of this thesis. In addition, we presented the possibility of future studies on this topic.

1.3 Thesis contribution

In this thesis, we applied feature engineering to the challenges in defect prediction that were overlooked by prior studies. In doing so, we have several contributions to defect prediction in software engineering. We highlighted some of the key contributions as follows:

1. We provided practitioners with advice on which feature selection/reduction technique to use in combination with a defect prediction model (Chapter 3).

2. We showed that context lines are informative to predict defective changes in just-in-time defect prediction. In addition, we formulated the context features based on context lines (Chapter 4).

3. We showed that a deep learning model can be used to retrieve semantic properties from modified source code for just-in-time defect prediction. In addition, we presented that our proposed model can improve the effectiveness of defect prediction with a small overhead on the prediction time (Chapter 5).

Chapter 2

Features in Defect Prediction

In the defect prediction research filed, numerous features have already been proposed and evaluated so far. Hence, some prior studies summarized existing features. For example, Madeyski et al. [95] summarized popular product features and process features. Tiwari et al. [164] conducted a systematic literature review and summarized the studies that are related to the evolution of coupling and cohesion features.

In feature engineering, summarizing existing features provides us with an overview of existing studies. Therefore, in this chapter, we also summarized the existing features. In particular, we classified the existing features into five categories (i.e., *product features*, *process features*, *organization features*, *developer features*, and *features of semantic properties*) and described popular features with some references for each category. Consequently, we clarified three challenges that have not been addressed yet in defect prediction.

2.1 Product features

Product features [9, 24, 50, 52, 98] measure the size or complexity of software entities. For example, one of the most popular features is lines of code (LOC). LOC counts the number of lines for an entity. For example, the number of lines in a file is LOC.

One of the suites of the product features is the object-oriented features [9,24]. The object-oriented features measure the complexity of source code written in object-oriented languages such as Java. The Chidamber and Kemerer (CK) features [24] are one of the most popular suites of the object-oriented features that consists of six features. For example, they proposed the weighted methods per class (WMC). WMC measures the complexity of a class by its methods. The McCabe cyclomatic complexity features (McCabe features) [98] measure the complexity of source code by using the control flow graph of source code. In particular, the McCabe features count the number of nodes, edges, and connected components in the control flow graph. The product features also include other features such as the Halstead features [50]).

The product features are typically applied to a complete software entity such as a file, class, and method. Hence, researchers and practitioners use the product features in coarse grained defect prediction such as *file/package-level defect prediction*[*] frequently.

[*]The aim of file/package-level defect prediction is to identify defective files/packages.

2.2 Process features

Process features (a.k.a. code churn features) measure the size, complexity, or metadata of changes in software development [54, 71, 77, 85, 110, 132]. Some of the product features can be used as process features. For example, LOC can be applied to a change (e.g., commit), though LOC is a product feature. The number of added lines in a commit can be used to measure the size of a commit. In particular, we refer to LOC that counts the number of added lines as lines of code added (LA); we refer to LOC that counts the number of deleted lines as lines of code deleted (LD) [71]. However, changes may not be a complete source code; and therefore, the McCabe features, for example, cannot be used as process features.

One of the most popular suites of process features is the change features [71]. The change features include 14 different features such as LA and LD. The change features can be measured on changes such as commits. Therefore, we typically use the change features on just-in-time defect prediction [38, 70, 71, 83, 176, 177].

2.3 Organization features

Organization features [19, 113] are less popular than the product features and the process features. The interest of the organization features is that the organization features measure the characteristics of not only source code but also its organization. For example, Nagappan et al. [113] proposed eight organization features. One of them is the number of engineers who contribute to source code and are still employed by the company. This feature does not measure any characteristics of source code.

2.4 Developer features

The concept of developer features [97] is similar to the organization features. In particular, the developer features measure the characteristics of not only source code but also developers. Indeed, some prior studies [66, 122] used the characteristics of developers in defect prediction. Matsumoto et al. [97] proposed two types of developer features: the features of developers' activities and the features of modules by developers.

2.5 Features of semantic properties

Features that were described above convert characteristics of software entities, organizations, and developers into numerical values. For example, LOC converts the size of software entities into the number of lines. However, these conversions may discard the characteristics of semantic properties. For example, if we modify the same number of lines on a quick sort program and a fizz buzz program, LOC could not capture the difference of complexity because of ignoring semantic properties. Some product features (e.g., the McCabe features) may capture semantic properties; however, such features need a complete entity (e.g., an entire class).

Therefore, prior studies have investigated methods to capture the semantic properties of source code [62, 66, 76, 91, 105, 155, 171, 176]. In particular, these studies applied text classification models to source code. For example, Mizuno et al. [105] applied a spam filter to a module to classify if a module is defective. Wang et al. [171] applied a deep learning model to learn semantic differences of source code.

Such text classification models provide several key advantages to capture the semantic properties as a suite of features over the other features. These text classification models 1) automatically learn semantic properties as a suite of features 2) can be applied to a part of software entities. Hence, many prior studies investigated the defect prediction performance of semantic properties by using such text classification models.

2.6 Chapter summary

As we described above, numerous features exist in defect prediction. Hence, it is important to carefully select the suite of features that is used on defect prediction models. Indeed, if we used too many features in a defect prediction model, the problem of multicollinearity and the so-called "curse of dimensionality" would be occurred and decrease the prediction performance. Hence, many of the existing defect prediction models used feature selection or reduction techniques [20, 28, 39, 41, 43, 44, 69, 70, 101, 111, 112, 115–117, 119, 133, 134, 136, 137, 149, 175, 186].

Recent studies [41, 175] investigated the impact of feature selection techniques on the performance of defect prediction models. However, to the best of our knowledge, nobody conducts a large-scale empirical study that compares the feature reduction techniques. Hence, we compared the feature reduction techniques on defect prediction models (Chapter 3).

Although the features of semantic properties have been investigated so far [62, 66, 76, 91, 105, 155, 171, 176], they overlooked to use the context lines as features. Hence, we proposed the context features that are retrieved from the context lines and investigated the prediction performance of the defect prediction models with the context features (Chapter 4).

Some of the previous studies used deep learning models to capture semantic properties [62, 91, 171, 176]. However, they overlooked to apply a deep learning model to retrieve semantic properties from changes in just-in-time defect prediction by 2018. Hence, we proposed W-CNN and compared W-CNN to a prior defect prediction model that uses a deep learning model with the change features (Chapter 5).

CHAPTER 3

THE IMPACT OF FEATURE REDUCTION TECHNIQUES ON DEFECT PREDICTION MODELS

An earlier version of this chapter is published in the Empirical Software Engineering Journal (EMSE) [82].

3.1 Introduction

Software developers have limited time to test their software. Hence, developers need to be selective when deciding where to focus their testing effort. Defect prediction models help developers focus their limited testing effort on components that are the most likely to be defective. Because detecting defects at an early stage can considerably reduce the development cost [162], defect prediction models have received widespread attention in software engineering research.

Many software features (e.g., software complexity features) can be used in defect prediction models [9,28,54,77,110]. However, it is important to carefully select the set of features that is used in such models, as using a set of features that is too large does not automatically result in better defect prediction. For example, prior studies showed that reducing the number of features avoids the problem of multicollinearity [37] and the curse of dimensionality [10]. Hence, many of the existing defect prediction models used feature selection or reduction techniques [20,28,39,41,43,44,69,70,101,111,112,115–117,119,133,134,136,137, 149,175,186].

Feature selection techniques reduce the number of features in a model by selecting the most important ones, while feature reduction techniques reduce the number of features by creating new, combined features from the original features. Recent studies [41,175] investigated the impact of feature *selection* techniques on the performance of defect prediction models. However, this chapter describes the first large-scale study on multiple feature *reduction* techniques and their impact on a large number of prediction models.

In this chapter, we compared the impact of the original features, features that are generated using *traditional feature reduction techniques* (i.e., PCA [28], FastMap [35], feature agglomeration [139], random projections [14], TCA [124] and TCA+ [117]), and features that are generated using *neural network-based feature reduction techniques* (i.e., restricted Boltzmann machine (RBM) [151] and autoencoder (AE) [60]) on defect prediction models. In addition, we compared the impact of features that are generated using feature reduction techniques with features that are selected using the best-performing feature selection techniques in prior work (correlation and consistency-based feature selection) [41, 175]. We compared the features along two dimensions: their performance (area under the receiver operating characteristic curve (AUC)), and their performance variance (interquartile range (IQR)). The receiver operating characteristic curve is created by plotting the true positive rate against the false positive rate, hence, the AUC represents the balance between the true positive rate and the false positive rate. The IQR represents the variance of a data distribution.

We conducted experiments on three publicly available datasets that contain software defects (the PROMISE [69], cleaned NASA [144], and AEEEM [28] datasets). Our ultimate goal is to identify which feature reduction techniques yield new, powerful features that preserve the predictive power of the original features, and improve the prediction performance compared to feature selection techniques. We studied the impact of feature reduction techniques on five supervised learning and five unsupervised learning models for defect prediction in our experiments. In particular, we focus on the following research questions:

RQ1: **What is the impact of feature reduction techniques on the performance of defect prediction models?**

Motivation: Reducing the number of features in a model can address the multicollinearity problem [37] and the curse of dimensionality [10]. In this RQ, we studied how feature reduction techniques impact the performance of supervised and unsupervised defect prediction models.

Results: Feature agglomeration and TCA can reduce the number of features, while preserving an AUC that is as good as that of the original features for supervised models. In addition, the AUC of unsupervised defect prediction models is significantly better when preprocessing the features with neural network-based feature reduction techniques than other feature reduction techniques.

RQ2: **What is the impact of feature reduction techniques on the variance of the performance across defect prediction models?**
Motivation: The AUC for a dataset can vary across defect prediction models. Hence, it can be challenging for practitioners to choose the defect prediction model that performs best on their data. If all defect prediction models have a small performance variance for a dataset, practitioners can avoid having to make this challenging choice.
Results: Neural network-based feature reduction techniques (RBM and AE) generate features that improve the variance of the performance across different defect prediction models in many cases when used in a supervised or unsupervised defect prediction model. In addition, almost all feature reduction techniques (except PCA) generate features that improve the variance of the performance across different defect prediction models in many cases when used in an unsupervised defect prediction model.

RQ3: **How do feature selection techniques compare to feature reduction techniques when applied to defect prediction?**
Motivation: Prior work [41,175] showed that several feature selection techniques outperform the original models. In this RQ, we studied how feature selection techniques compare to feature reduction techniques.
Results: For the supervised defect prediction models, the studied feature selection techniques (correlation and consistency-based feature selection) outperform all the studied feature reduction techniques. However, for the unsupervised defect prediction models, the neural network-based feature reduction techniques (RBM and AE) outperform the other studied feature selection/reduction techniques.

Our results provide practitioners with advice on which feature selection/reduction technique to use in combination with a defect prediction model. In particular, we recommend to use a feature selection technique when using a supervised defect prediction model, and a neural network-based feature reduction technique when using an unsupervised defect prediction model, as these feature selection/reduction techniques improve the variance across defect prediction models, while outperforming the other feature reduction techniques.

3.2 Methodology

In this section, we describe our methodology. In particular, we discuss our studied datasets, defect prediction models, feature selection techniques, feature reduction techniques, evaluation measure, our preprocessing steps, and our validation scheme.

3.2.1 Studied datasets

In our work, we used three publicly available datasets (the PROMISE [69], cleaned NASA [118] and AEEEM [28] datasets) that were used by prior work [181] on supervised and unsupervised defect prediction models. Table 3.1 describes the studied datasets. All datasets contain popular software features for measuring source code complexity (see Table 3.2 for a summary of the used features). Each entity in a dataset is labelled as defective or clean.

The PROMISE dataset contains several types of projects. We chose the 10 projects that were used by prior work [181], to ease the comparison of our results with prior work. All studied PROMISE projects have the same number of features. The PROMISE dataset contains the Chidamber and Kemerer (CK) features [24] and an additional set of object-oriented (OO) features.

The NASA dataset [118] contains 11 projects. Each project in the NASA dataset has a different number of features. The NASA dataset contains McCabe features [98] and Halstead features [50]. We used the cleaned version [144] of the NASA dataset, because prior studies [128, 144] showed that the original version of the NASA dataset contains inconsistent and mislabeled data.

The AEEEM dataset [28] contains five projects. All projects have the same

Table 3.1: Description of studied projects

Studied Dataset	Project	# of Entities	# of Defective	% Defective	# of Features*	# of Studied Features*
PROMISE	Ant v1.7	745	166	22.3	24	20
	Camel v1.6	965	188	19.5	24	20
	Ivy v1.4	241	16	6.6	24	20
	Jedit v4.0	306	75	24.5	24	20
	Log4j v1.0	135	34	25.2	24	20
	Lucene v2.4	340	203	59.7	24	20
	POI v3.0	442	281	63.6	24	20
	Tomcat v6.0	858	77	9.0	24	20
	Xalan v2.6	885	411	46.4	24	20
	Xerces v1.3	453	69	15.2	24	20
NASA	CM1	327	42	12.8	38	37
	JM1	7,782	1,672	21.5	22	21
	KC3	194	36	18.6	40	39
	MC1	1,988	46	2.3	39	38
	MC2	125	44	35.2	40	39
	MW1	253	27	10.7	38	37
	PC1	705	61	8.7	38	37
	PC2	745	16	2.1	37	36
	PC3	1,077	134	12.4	38	37
	PC4	1,287	177	13.8	38	37
	PC5	1,711	471	27.5	39	38
AEEEM	Eclipse JDT Core	997	206	20.7	291	212
	Equinox	324	129	39.8	291	212
	Apache Lucene	691	64	9.3	291	212
	Mylyn	1,862	245	13.2	291	212
	Eclipse PDE UI	1,497	209	14.0	291	212

* We removed features that are not related to source code. For instance, the name of the file, name of the class and the version. Hence, the number of studied features are different from the total number of features.

Table 3.2: Studied features

Studied Dataset	Features	Reference
PROMISE	CK (OO)	Chidamber et al. [24] and Basili et al. [9]
NASA	McCabe	McCabe [98]
	Halstead	Halstead [50]
AEEEM	CK (OO)	Chidamber et al. [24] and Basili et al. [9]
	number of previous defects	Kim et al. [77]
	change features	Moser et al. [110]
	complexity code change features	Hassan [54]
	churn of CK and OO	D'Ambros et al. [28]
	entropy of CK and OO	D'Ambros et al. [28]

number of features. Like the PROMISE dataset, the AEEEM dataset contains the CK and OO features. However, the AEEEM data also contains the number of previous defects [77], change features [110], complexity code change features [54], and the churn and entropy of the CK and OO [28] features.

3.2.2 Studied defect prediction models

We focused on defect prediction models that were used by prior work [181], to make our results easier to compare. We studied five supervised models and five unsupervised models. Below we give a brief overview of the ideas behinds these models. For a detailed overview, we refer the reader to the original papers in which these models were introduced. We studied the following supervised defect prediction models:

- *Logistic Regression (LR) [99]:* LR is one of the most commonly used models for defect prediction. LR expresses the relationship between one or more independent variables (i.e., the original features) and one dependent variable (i.e., defective or clean) using a polynomial expression and a *sigmoid function* [51].

- *Decision Tree (J48) [131]:* J48 is a decision tree implementation in WEKA [45]. The decision tree uses a tree structure to decide on the dependent variable. In this tree, each node corresponds to one of the independent variables with a condition. J48 traverses the tree from the root to a leaf, while taking into account the input entity and the conditions in the tree. Each leaf corresponds to a value of the dependent variable.

- *Random Forest (RF) [61]:* RF is a popular *ensemble learning* model. RF builds multiple decision trees based on subsets of training data that are randomly selected. RF decides on a value of the dependent variable by taking the result of a majority of the decision trees.

- *Naive Bayes (NB) [182]:* NB is a probability-based classifier that follows Bayes' theorem. Bayes' theorem describes the probability of an event, given knowledge of conditions that could be related to the event.

- *Logistic Model Tree (LMT) [89]:* LMT is a classifier which combines a decision tree and a logistic regression model. Like the decision tree, LMT follows a tree structure. However, LMT uses logistic regressions instead of values in the leaves.

We used the `caret` library in `R` [88] to optimize the parameters of the supervised models as suggested by Tantithamthavorn et al. [159].

We studied the following unsupervised defect prediction models:

- *Spectral Clustering (SC) [169]:* SC labels entities using a graph that is based on similarities across entities. In this graph, each node is an entity and each edge represents the similarity of the entities it connects. SC cuts sparse edges in this graph by classifying eigenvectors of the *Laplacian matrix* [169] of the graph. Following this process, SC divides the entities into two groups.

- *k-means (KM) [53]:* KM is a popular clustering approach. KM classifies entities based on the distances between entities and the center of a class (i.e., the mean of all entities in that class). In this chapter, we used the Euclidean distance as the distance metric.

- *Partition Around Medoids (PAM) [74]:* PAM is an approach that is similar to KM. While KM uses the center of a class, PAM uses *medoids*. A medoid is an entity of which the sum of all the distances to the other entities in a class is at its minimum. Because PAM uses the medoid instead of the center, PAM is less likely to be affected by outliers than KM.

- *Fuzzy C-Means (FCM) [32]:* FCM is also an approach that is similar to KM. While KM classifies each entity to only one class in its process, FCM allows entities to be a member of more than one class. The membership is expressed as a probability.

- *Neural Gas (NG) [96]:* NG is an approach that is similar to a *self-organizing map* [81]. NG generates *weighted points* which have random features. Hence, the weighted points are distributed across the feature space. For each learning iteration, the features of the weighted points are updated by distances to closer entities. Finally, the weighted points become the class centers.

We used the default parameters of the implementations for the unsupervised models. We set the number of clusters to two, as defect prediction is a binary problem. Table 3.3 shows the libraries that we used for the implementation of the defect prediction models.

Labeling technique in the unsupervised models: The unsupervised models classify the data in two unlabeled clusters. We adopted the following heuristic to identify the defective cluster: *"For most features, software entities containing defects generally have larger values than software entities without defects"* [181]. In particular, we used the sum of row average of the normalized features in each cluster, to decide which cluster contains the defects [181]. To calculate the sum of row average, we first summed the entity values in each cluster, respectively. Then, we

Table 3.3: Packages used for experiments

Defect Prediction Models	**Packages**
LR	The `caret` package in `R`
RF	The `caret` package in `R`
NB	The `caret` package in `R`
J48	The `caret` package in `R`
LMT	The `caret` package in `R`
SC	Zhang et al.'s implementation [181]
KM	The `cclust` package in `R`
PAM	The `cluster` package in `R`
FCM	The `e1071` package in `R`
NG	The `cclust` package in `R`

calculated the average values for each cluster. The cluster with the larger average value was identified as the cluster with the defective entities.

3.2.3 Studied feature reduction techniques

In this section, we discuss the studied feature reduction techniques. We studied two types of feature reduction techniques: *traditional* and *neural network-based* feature reduction techniques. We give a brief overview of the core concepts of each feature reduction technique. For more precise details, we refer to the references that are mentioned for each technique. Figure 3.1 shows a visualization of the traditional feature reduction techniques (PCA, FM, FA, TCA/TCA+, RP). Figure 3.2 gives an overview of neural network-based feature reduction techniques (RBM and AE).

Traditional feature reduction techniques

We studied the following traditional feature reduction techniques.

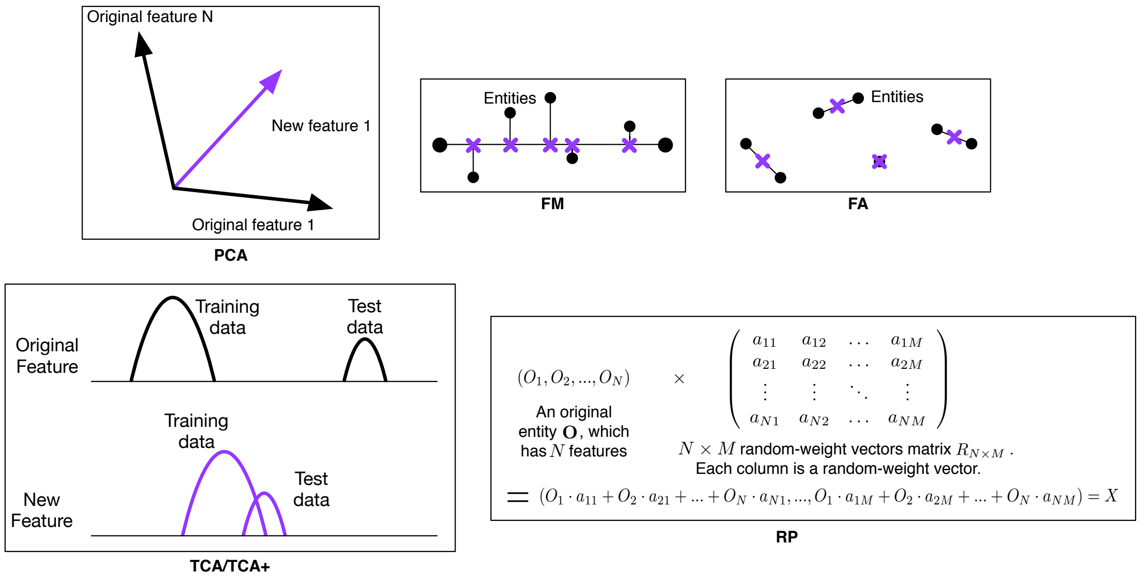

Original feature N
New feature 1
Original feature 1
PCA
Entities
FM
Entities
FA
Training data
Test data
Original Feature
Training data
Test data
New Feature
TCA/TCA+
$(O_1, O_2, ..., O_N)$
An original entity $\mathbf{O}$, which has N features
$\times$
$\begin{pmatrix} a_{11} & a_{12} & \dots & a_{1M} \\ a_{21} & a_{22} & \dots & a_{2M} \\ \vdots & \vdots & \ddots & \vdots \\ a_{N1} & a_{N2} & \dots & a_{NM} \end{pmatrix}$
$N \times M$ random-weight vectors matrix $R_{N \times M}$.
Each column is a random-weight vector.
$= (O_1 \cdot a_{11} + O_2 \cdot a_{21} + ... + O_N \cdot a_{N1}, ..., O_1 \cdot a_{1M} + O_2 \cdot a_{2M} + ... + O_N \cdot a_{NM}) = X$
RP

Figure 3.1: A visual overview of the core concepts of the traditional feature reduction techniques. The black symbols represent the original features (or entities in FM and FA) and the purple symbols represent the newly-generated features (or entities in FM and FA). RP (Random projection) transforms an original entity to a new entity using M random-weight vectors.

- *Principal Component Analysis (PCA):* PCA is one of the most commonly used feature reduction techniques in defect prediction [20,28,41,47,133,134,175]. PCA reduces the number of features by projecting the original set of features on a smaller number of principal components.

- *FastMap (FM):* For N original features, FM [35] first generates a (N-1)-dimensional orthogonal hyper-plane of the line between two entities that are far from each other. Second, FM projects the other entities on this hyper-plane. Because FM projects the entities on the $N-1$ orthogonal hyper-plane, we can reduce one feature from the original features. FM repeats this procedure until we get the required number of new features. For instance, if we want three features to visualize our data from the N original features, we repeat the procedure N-3 times.

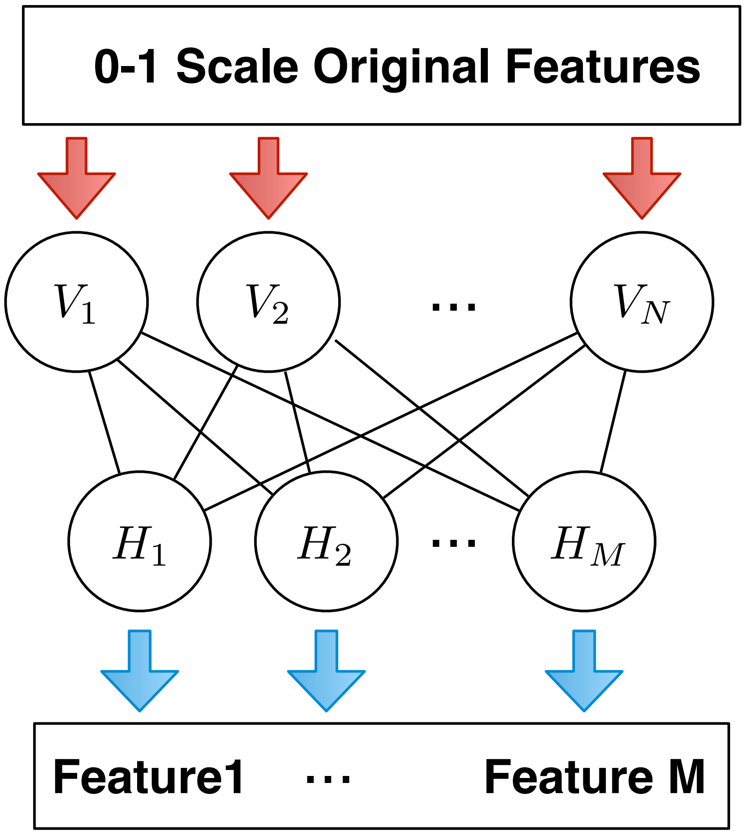


Figure 3.2: An overview of neural network-based feature reduction techniques (RBM and AE). RBM and AE convert the original features (V_i), which values must range between 0 and 1, into M new features (H_i). Note that the original input data may need to be preprocessed to satisfy the 0-1 range requirement.

- *Feature Agglomeration (FA):* FA is a simple hierarchical clustering algorithm [139]. FA starts by creating a new feature from each original feature. Then, FA merges the two nearest (based on their Euclidean distance) features into one feature, and repeats this process until the desired number of features is reached.

- *Transfer Component Analysis (TCA and TCA+):* TCA [124] creates new features from the original features by projecting them on so-called transfer components (similar to PCA). However, the goal of TCA is not to reduce the number of features, but to reduce the gap between the distribution of the training and testing data. During this process, the number of features is often reduced. Hence, TCA can be used as a feature reduction technique. TCA+ [117] is an extension of TCA, which optimizes the data using a preprocessing step according to the gap between the distribution of the training and testing data, such as scaling the original features between 0 and 1 instead of using the z-score.

- *Random Projection (RP):* RP projects the original N-dimensional features onto M generated features ($M \ll N$) using a $N \times M$ random-weight vectors matrix [14]. The equation of RP is as follows:

$$\mathrm{X} = \mathbf{O} \times R_{N\times M}$$

where X is a generated M-dimensional vector entity, $\mathbf{O} = (O_1, O_2, ..., O_N)$ is an original entity, and $R_{N\times M}$ is a random-weight vectors matrix. For example, if we want three features from N original features, we prepare three random-weight vectors with N random values in each of them. The random values are selected such that X represents the original features.

Neural network-based feature reduction techniques

We studied the following neural network-based feature reduction techniques.

- *Restricted Boltzmann Machine (RBM):* An RBM [151] automatically extracts important information from the original features as weights and biases on a two-layered neural network. Each node in the first-layer corresponds to an original feature, and each node in the second-layer corresponds to a new feature. We use the output of the second-layer as the new features.

- *Autoencoder (AE):* AE [60] and RBM are similar, but trained differently. In RBM, the network is trained based on a probability distribution. In AE, the network is trained using the difference between the original and the generated features.

3.2.4 Studied feature selection techniques

We studied the correlation-based (CFS) and consistency-based feature selection techniques (ConFS). These techniques were reported as the best-performing feature selection techniques in prior studies [41,175]. Below, we give a brief overview of these techniques.

- Correlation-based feature selection (CFS) [46]: CFS selects a subset of features based on their *correlation*. The selected features have strong correlations with the class label (clean or defective), while having a low correlation with each other.

- Consistency-based feature selection (ConFS) [29]: ConFS uses the *consistency* of the class label across the entities instead of the correlation. For example, if file A has a feature set (10, 20, 40, defective) and file B has a feature set (10, 20, 30, clean), we can identify the defective and clean entities using the third feature. However, if a feature reduction technique removes the third feature, file A and file B have the same feature set except for the class label. In that case, these entities are inconsistent. Using this information, ConFS selects the best feature subset from the original features.

3.2.5 Area under the receiver operating characteristic curve (AUC)

We used the Area Under the receiver operating characteristic Curve (AUC) as the performance measure since AUC is not affected by the skewness of defect data [158, 181]. The receiver operating characteristic (ROC) curve is created by plotting the false positive rate (on the x-axis) and the true positive rate (on the y-axis) at various thresholds. In our experiment, the false positive rate is defined as the portion of clean entities that are identified as defective; the true positive is defined as the portion of defective entities that are identified as defective. The threshold is used to label an entity as clean or defective by checking whether its predicted probability is over the threshold. The AUC is the area under the ROC curve. The values of the AUC range between 0 and 1; a perfect classifier has an AUC of 1, while a random classifier has an AUC of 0.5.

3.2.6 Preprocessing

Most feature reduction techniques require the data to be preprocessed. We detail the preprocessing step below.

Preprocessing for traditional feature reduction techniques

The traditional feature reduction techniques require features that are normalized to a mean of 0 and a variance of 1 using the z-score [181]. The z-score is calculated as follows:

$$\mathbf{X}_z = \frac{\mathbf{X}_{org} - \mu}{\sigma} \tag{3.1}$$

where μ is a mean of the value of the feature for all entities and σ is the standard deviation of the value of the feature for all entities.

Preprocessing for neural network-based feature reduction techniques

The neural network-based feature reduction techniques require either binary features or features that are between 0 and 1. Hence, we scale the original features as follows:

$$\mathbf{X}_{scaled} = \frac{\mathbf{X}_{org} - X_{\min}}{X_{\max} - X_{\min}} \tag{3.2}$$

where $\mathbf{X}_{org}$ is a vector of the value of a particular feature for all entities. $X_{\min}$ is the smallest value of the feature and $X_{\max}$ is the largest value of the feature for all entities [1].

3.2.7 Out-of-sample bootstrap sampling

Bootstrap sampling is a validation technique that is used to estimate the performance of a model for unseen data. The technique is based on random sampling with replacement. *Out-of-sample bootstrap sampling* is a bootstrap sampling technique that estimates the future performance of a defect prediction model more accurately than a cross-validation scheme [158, 160]. Hence, we used the out-of-sample bootstrap sampling technique instead of a conventional validation technique such as 10-fold cross-validation. The process of the out-of-sample bootstrap sampling is as follows:

1. Sample N data points following the distribution of the original dataset, with N data points, while allowing for replacement.

2. Train a model using the sampled N data points, and test it using the data points that were not sampled.

3. Repeat steps 1 and 2 M times.

4. Report the average/median performance as the performance estimate.

We used the out-of-sample bootstrap sampling under the condition where $M = 100$ and we report the median performance.

3.3 Experimental setup

In this section, we give an overview of the setup of our experiments. The results are presented in Section 3.4. Figure 3.3 shows the steps of our experiments. We first conducted the out-of-sample bootstrap sampling on our studied datasets to generate and select features using each of the studied feature reduction and selection techniques. We then preprocessed the original features of each *bootstrap sample* as described in Section 3.2.6. We generated eight new feature sets (one for each feature reduction technique) for each bootstrap sample. Hence, we generated 800 new feature sets using feature reduction in total. Furthermore, the two studied feature selection techniques selected two feature subsets (one for each feature selection technique) for each bootstrap sample. Hence, we selected 200 feature subsets using feature selection in total.

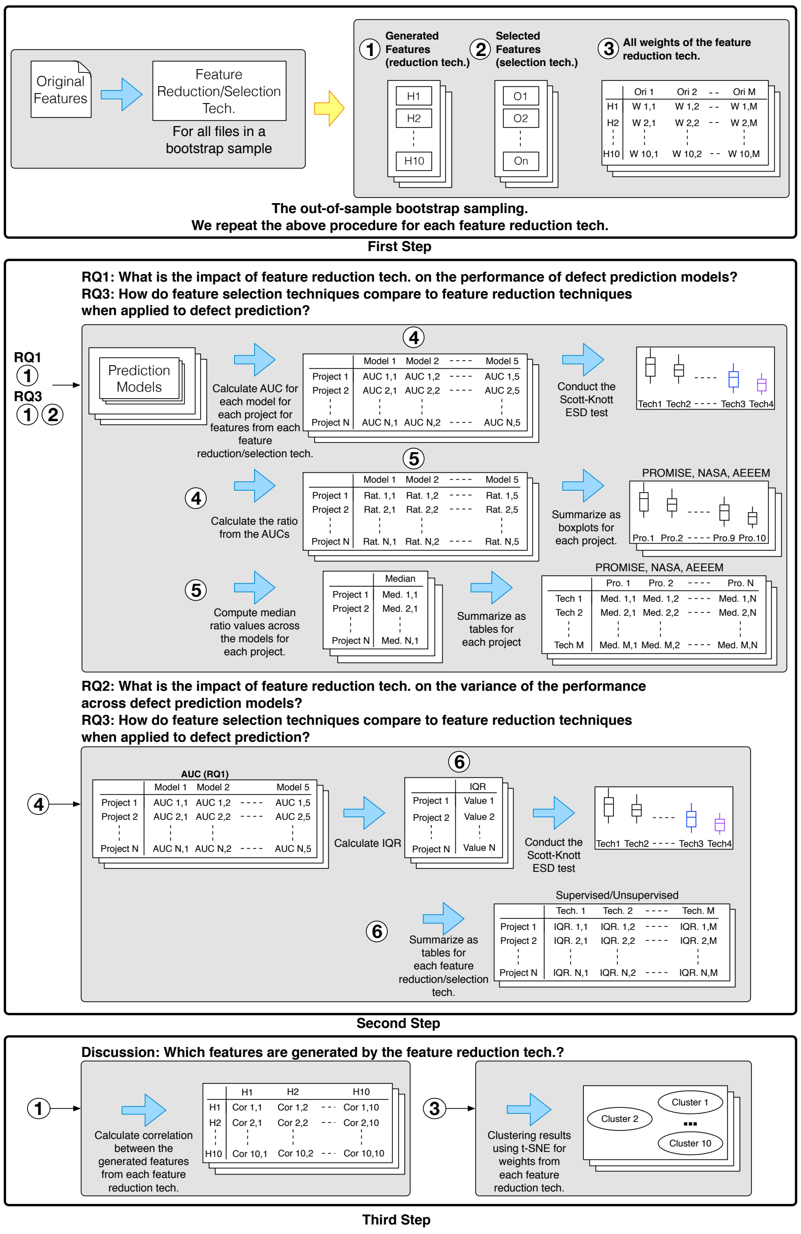

Original Features
Feature Reduction/Selection Tech.
For all files in a bootstrap sample
1 Generated Features (reduction tech.)
2 Selected Features (selection tech.)
3 All weights of the feature reduction tech.
H1 H2 H10
O1 O2 On
Ori 1 Ori 2 Ori M
W 1,1 W 1,2 W 1,M
W 2,1 W 2,2 W 2,M
W 10,1 W 10,2 W 10,M
The out-of-sample bootstrap sampling.
We repeat the above procedure for each feature reduction tech.
First Step
RQ1: What is the impact of feature reduction tech. on the performance of defect prediction models?
RQ3: How do feature selection techniques compare to feature reduction techniques when applied to defect prediction?
RQ1 1
RQ3 1 2
Prediction Models
Calculate AUC for each model for each project for features from each feature reduction/selection tech.
4
Model 1 Model 2 Model 5
Project 1 AUC 1,1 AUC 1,2 AUC 1,5
Project 2 AUC 2,1 AUC 2,2 AUC 2,5
Project N AUC N,1 AUC N,2 AUC N,5
Conduct the Scott-Knott ESD test
Tech1 Tech2 Tech3 Tech4
4 Calculate the ratio from the AUCs
5
Project 1 Rat. 1,1 Rat. 1,2 Rat. 1,5
Project 2 Rat. 2,1 Rat. 2,2 Rat. 2,5
Project N Rat. N,1 Rat. N,2 Rat. N,5
Summarize as boxplots for each project.
PROMISE, NASA, AEEEM
Pro.1 Pro.2 Pro.9 Pro.10
5 Compute median ratio values across the models for each project.
Median
Project 1 Med. 1,1
Project 2 Med. 2,1
Project N Med. N,1
Summarize as tables for each project
PROMISE, NASA, AEEEM
Pro. 1 Pro. 2 Pro. N
Tech 1 Med. 1,1 Med. 1,2 Med. 1,N
Tech 2 Med. 2,1 Med. 2,2 Med. 2,N
Tech M Med. M,1 Med. M,2 Med. M,N
RQ2: What is the impact of feature reduction tech. on the variance of the performance across defect prediction models?
RQ3: How do feature selection techniques compare to feature reduction techniques when applied to defect prediction?
4
AUC (RQ1)
Model 1 Model 2 Model 5
Project 1 AUC 1,1 AUC 1,2 AUC 1,5
Project 2 AUC 2,1 AUC 2,2 AUC 2,5
Project N AUC N,1 AUC N,2 AUC N,5
Calculate IQR
6
IQR
Project 1 Value 1
Project 2 Value 2
Project N Value N
Conduct the Scott-Knott ESD test
Tech1 Tech2 Tech3 Tech4
6 Summarize as tables for each feature reduction/selection tech.
Supervised/Unsupervised
Tech. 1 Tech. 2 Tech. M
Project 1 IQR. 1,1 IQR. 1,2 IQR. 1,M
Project 2 IQR. 2,1 IQR. 2,2 IQR. 2,M
Project N IQR. N,1 IQR. N,2 IQR. N,M
Second Step
Discussion: Which features are generated by the feature reduction tech.?
1
Calculate correlation between the generated features from each feature reduction tech.
H1 H2 H10
H1 Cor 1,1 Cor 1,2 Cor 1,10
H2 Cor 2,1 Cor 2,2 Cor 2,10
H10 Cor 10,1 Cor 10,2 Cor 10,10
3
Clustering results using t-SNE for weights from each feature reduction tech.
Cluster 1
Cluster 2
Cluster 10
Third Step

Figure 3.3: Overview of our experimental design. We first generate/select 100 (the out-of-sample bootstrap) feature sets using each feature reduction/selection technique for each studied dataset. The second step is different for each RQ. We conduct correlation analysis and clustering analysis for discussion in the third step.

The smallest number of features in the studied datasets is 20 (i.e., in the PROMISE dataset). Hence, to be able to observe the impact of a feature reduction technique, we configured each feature reduction technique to generate 10 features (H1–H10). However, PCA uses variance to decide on the number of generated features [28,41]. Therefore, each bootstrap sample results in a different number of generated features using PCA. We configured PCA to retain 95% of the variance in the data [28, 41]. The median number of generated features by PCA in our experiments was 12 in the PROMISE dataset, 10 in the NASA dataset and 34 in the AEEEM dataset.

The experimental setup for each RQ is discussed in the next section.

3.4 Results

In this section, we present the results of our experiments. For each RQ, we discuss the motivation, approach and results.

3.4.1 RQ1: What is the impact of feature reduction techniques on the performance of defect prediction models?

Motivation: Reducing the number of features that are used in a defect prediction model can be beneficial for addressing the curse of dimensionality and multi-collinearity of the model. There exist two ways to reduce the number of features in a model: (1) by *selecting* the most important features, and (2) by *reducing* the number of features by creating new, combined features from the original features. Prior work has systematically studied the impact of feature selection techniques on defect prediction [41, 175], but no work has conducted a large-scale study of the impact of feature reduction techniques on defect prediction. Hence, in this RQ, we studied the impact of feature reduction techniques on the performance (AUC) of defect prediction models.

Approach: We used each feature set that was generated by a feature reduction technique as input to the studied five supervised and five unsupervised defect prediction models. We used the AUC as the performance measure. Because we calculated the AUC of a defect prediction model using the out-of-sample bootstrap sampling 100 times for each feature reduction technique, each model has 100 AUC values. Hence, we used the median value to represent the median performance of a defect prediction model using a certain feature reduction technique. Because we studied 26 projects, our experiments yielded 260 median AUC values for each feature reduction technique (5 supervised models*26 projects+5 unsupervised models*26 projects). For comparison, we also calculated the performance of the studied defect prediction models without applying a feature reduction technique (indicated as *ORG*). Note that we did within-project defect prediction in our experiments.

We used the Scott-Knott ESD test [160] (using a 95% significance level) to compare the median AUC values across feature reduction techniques. The Scott-Knott test is a hierarchical clustering algorithm that ranks the distributions of values. In particular, distributions that are not statistically significantly different are placed in the same rank. The Scott-Knott ESD test is an extension of the Scott-Knott test, which not only ranks based on significance, but also on Cohen's *d* effect size [25]. The Scott-Knott ESD test places distributions which are not significantly different, or have a negligible effect size, in the same rank. We used the `ScottKnottESD` `R` package[*] that was provided by Tantithamthavorn [159].

Project-level analysis: the aforementioned procedure combines the results of all projects. However, this procedure prevents us from understanding differences for each project. Hence, we also studied the performance at the project-level.

We compared the ratios of the AUCs (the median AUCs across all bootstrap samples) of each feature reduction technique to the original models. We calculated this ratio as follows:

$$\text{ratio} = \frac{AUC_{FR}}{AUC_{ORG}}$$

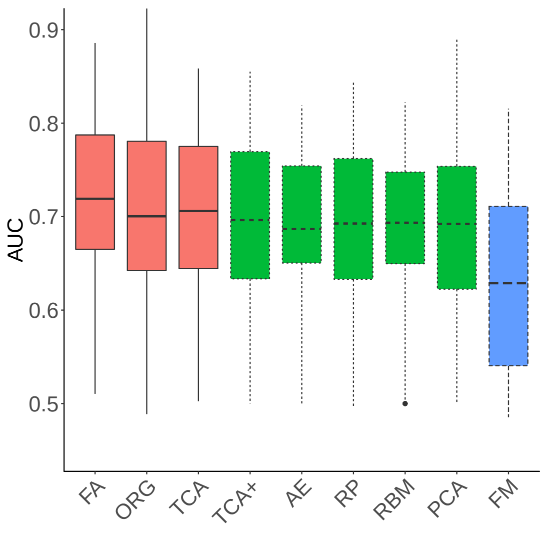
[*]`https://github.com/klainfo/ScottKnottESD`

Where AUC_{ORG} is the AUC of a prediction model using the original features, and AUC_{FR} is the AUC of a prediction model using the features that were generated by a particular feature reduction technique. Hence, a ratio larger than 1 indicates that the feature reduction technique improved the AUC compared to the original models, while a ratio smaller than 1 indicates that the feature reduction technique reduced the AUC. We computed the median ratio across the five studied supervised and unsupervised prediction models.

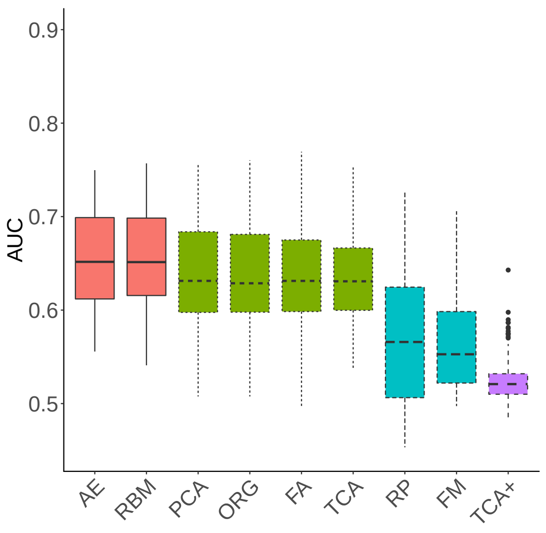
We used the aforementioned ratio to analyze performance at the project-level. The project-level analysis shows the impact of the different feature reduction techniques in every single project and dataset. Figure 3.6 shows the distributions of the ratios for each studied project for the supervised and unsupervised prediction models, respectively. Each boxplot contains 40 ratio values (5 prediction models * 8 feature reduction techniques). In addition, we show the median ratios for the best-performing feature reduction techniques as tables for deeper analysis (Table 3.4). These median ratios were computed from five AUC values (one for each studied prediction model).

Results: **FA and TCA can preserve the performance of the original defect prediction models, while at the same time reducing the number of features.** Figure 3.4(a) and Figure 3.4(b) show that the performance of the supervised and unsupervised defect prediction models does not decrease when applying FA or TCA. Hence, these feature reduction techniques can safely be applied to reap the benefits of a reduced number of features. In particular, FA and TCA work well for supervised models. Interestingly, the performance of the supervised and unsupervised defect prediction models is significantly lower when using TCA+ (which is an extension of TCA), compared to the original TCA.

The neural network-based feature reduction techniques (RBM and AE), significantly outperform traditional feature reduction techniques for the unsupervised defect prediction models. Figure 3.4(b) shows the AUC values and the results of the Scott-Knott ESD test for the unsupervised models after applying the



(a) The supervised models



(b) The unsupervised models

Figure 3.4: The Scott-Knott ESD test results for the supervised (logistic regression, random forest, naive Bayes, decision tree, and logistic model tree) and the unsupervised (spectral clustering, *k*-means, partition around medoids, fuzzy C-means, neural-gas) models. Each color indicates a rank: models in different ranks have a statistically significant difference in performance. Each boxplot has 130 median AUC values (5 defect prediction models times 26 projects). The x-axis refers to the feature reduction techniques; the y-axis refers to the AUC values.

studied feature reduction techniques.

The highest rank contains only the two studied neural network-based techniques: RBM and AE. Hence, the neural network-based feature reduction techniques can significantly improve the AUC compared to the original models and other feature reduction techniques. However, these neural network-based feature reduction techniques do not outperform ORG for the supervised models. In Section 3.5 we further investigate why neural network-based feature reduction techniques work well for the unsupervised, but not for the supervised defect prediction models.

The supervised models with feature reduction techniques significantly out-

Table 3.4: The median AUC ratios of the feature reduction techniques. A ratio larger than 1 indicates that the feature reduction/selection technique improved the AUC compared to the original models. The gray cells refer to the ratios that are greater than 1.0. The "Improved" row indicates the number of projects for which a feature reduction/selection technique improved the performance.

(a) The supervised models

		RBM	AE	PCA	FM	FA	RP	TCA	TCA+	CFS	ConFS
PROMISE	Ant v1.7	1.015	1.004	0.983	0.896	0.995	0.956	0.962	0.962	1.012	1.006
	Camel v1.6	0.973	0.979	0.921	0.880	0.996	0.962	0.948	0.948	0.962	0.956
	Ivy v1.4	1.065	1.081	1.005	1.005	1.026	0.999	1.036	1.020	1.062	1.037
	Jedit v4.0	1.012	0.996	0.971	0.775	1.011	0.921	0.952	0.947	0.984	0.996
	Log4j v1.0	1.047	1.034	1.000	0.960	1.012	0.895	0.961	0.954	0.993	1.002
	Lucene v2.4	1.016	0.956	0.940	0.804	0.978	0.886	0.962	0.955	0.990	0.978
	POI v3.0	0.932	0.899	0.955	0.739	0.993	0.946	0.965	0.971	1.016	1.003
	Tomcat v6.0	1.014	1.016	0.947	0.875	0.997	0.925	0.985	0.990	1.045	1.026
	Xalan v2.6	0.861	0.912	0.987	0.698	0.992	0.970	0.993	0.985	1.012	0.998
	Xerces v1.3	1.032	1.022	0.969	0.817	1.019	1.017	1.034	1.033	1.005	1.012
NASA	CM1	1.004	0.979	0.939	0.984	0.974	0.995	0.949	0.940	1.028	0.981
	JM1	1.004	1.001	0.997	0.923	0.999	1.003	1.008	0.956	1.000	1.001
	KC3	0.910	0.944	0.901	0.923	1.010	0.924	0.905	0.874	1.025	0.985
	MC1	0.956	1.021	0.932	0.901	0.999	0.915	0.980	0.932	1.001	0.973
	MC2	1.019	1.014	0.955	0.947	1.037	1.014	0.976	0.972	0.973	0.968
	MW1	1.005	1.016	0.949	0.984	1.019	0.991	0.952	0.963	1.016	1.015
	PC1	0.906	0.830	0.936	0.972	1.013	0.976	0.971	0.925	1.004	1.021
	PC2	1.039	1.019	0.931	0.994	1.020	0.999	0.935	0.933	1.169	1.028
	PC3	0.944	0.971	0.985	0.921	0.997	0.988	1.001	0.968	1.026	0.995
	PC4	0.793	0.806	0.931	0.651	0.906	0.862	0.866	0.870	1.003	0.986
	PC5	0.999	0.982	0.994	0.798	0.990	0.977	0.989	0.986	0.986	0.997
AEEEM	Eclipse JDT Core	1.028	1.010	1.019	0.874	1.007	0.959	1.015	0.986	0.993	1.012
	Equinox	1.074	1.043	1.006	0.973	1.028	0.996	1.000	1.010	1.060	1.024
	Apache Lucene	1.112	1.079	1.090	1.045	1.009	1.063	1.030	1.039	1.021	1.035
	Mylyn	1.089	1.057	0.980	0.921	1.052	0.951	1.066	1.065	1.037	1.023
	Eclipse PDE UI	1.071	1.058	1.075	0.861	1.035	0.944	1.020	0.905	0.995	1.006
	Improved	17	15	5	2	14	4	8	5	17	15

(b) The unsupervised models

		RBM	AE	PCA	FM	FA	RP	TCA	TCA+	CFS	ConFS
PROMISE	Ant v1.7	1.005	1.000	1.001	0.739	0.896	0.695	0.918	0.748	1.002	1.006
	Camel v1.6	0.989	0.980	1.002	0.894	0.979	0.848	0.984	0.933	0.977	0.988
	Ivy v1.4	1.040	1.011	1.006	0.938	0.850	0.750	0.991	0.874	0.937	0.928
	Jedit v4.0	1.055	1.038	1.000	0.895	1.033	0.888	0.966	0.802	0.939	0.962
	Log4j v1.0	0.993	0.989	1.001	0.950	0.973	0.694	0.924	0.790	0.954	0.986
	Lucene v2.4	1.110	1.131	1.001	0.870	0.989	0.894	0.966	0.858	0.984	1.004
	POI v3.0	0.919	0.924	1.002	0.755	1.036	0.875	0.610	0.755	0.962	0.982
	Tomcat v6.0	1.041	1.007	0.997	0.762	0.956	0.652	1.007	0.763	1.010	0.968
	Xalan v2.6	1.070	1.065	0.997	0.885	0.936	0.965	1.084	0.848	0.964	0.974
	Xerces v1.3	1.244	1.210	1.008	0.885	1.259	0.893	1.201	0.939	1.106	1.017
NASA	CM1	1.008	0.996	0.997	0.967	0.979	1.049	0.983	0.844	1.030	0.949
	JM1	0.992	1.023	1.002	0.953	1.036	0.997	1.011	0.849	1.008	1.001
	KC3	0.970	0.987	1.000	0.974	1.014	0.997	0.998	0.850	0.968	0.955
	MC1	1.005	1.023	1.000	0.891	1.021	1.000	0.994	0.816	1.081	1.023
	MC2	1.073	1.043	1.000	0.981	1.018	0.985	0.995	0.862	0.993	0.991
	MW1	0.939	0.971	1.000	0.973	0.966	0.990	0.957	0.746	1.017	1.019
	PC1	0.990	0.996	0.996	0.927	1.027	0.935	0.993	0.855	1.102	1.041
	PC2	0.985	0.976	0.994	0.975	0.985	0.993	0.966	0.774	1.035	0.940
	PC3	1.074	1.090	1.000	0.940	0.954	0.758	1.041	0.831	1.213	1.123
	PC4	0.986	0.982	0.998	0.902	1.062	0.829	0.961	0.818	1.123	0.991
	PC5	0.971	0.975	0.999	0.866	0.971	0.987	0.932	0.857	1.030	0.997
AEEEM	Eclipse JDT Core	1.121	1.089	0.997	0.888	0.957	0.995	1.107	0.805	1.017	1.007
	Equinox	1.029	1.105	1.000	0.913	1.069	1.033	1.073	0.900	1.025	1.033
	Apache Lucene	1.031	1.053	1.000	0.863	0.989	1.006	1.024	0.740	0.956	0.981
	Mylyn	1.006	1.015	1.001	0.829	0.996	0.892	0.994	0.827	1.009	0.996
	Eclipse PDE UI	1.015	1.024	1.000	0.827	0.980	0.977	0.993	0.793	1.003	1.024
	Improved	16	15	9	0	10	3	8	0	16	11

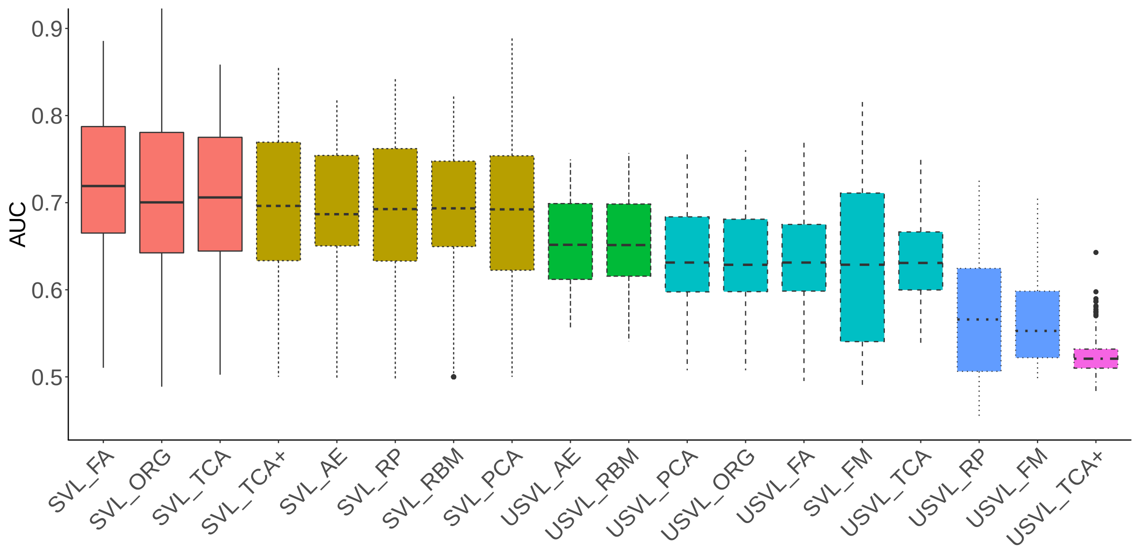
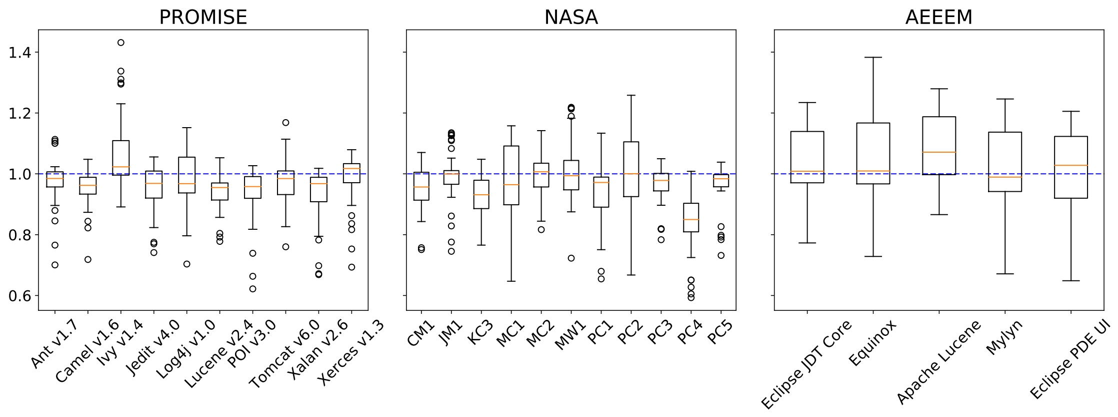


Figure 3.5: The Scott-Knott ESD test results for both the supervised and unsupervised models. Each color indicates a rank: models in different ranks have a significant difference in performance. Each boxplot has 130 median AUC values (5 defect prediction models times 26 projects). The x-axis refers to the feature reduction techniques; the y-axis refers to the AUC values. In the x-axis, the "SVL_"-prefix refers to the 5 supervised defect prediction models; the "USVL_"-prefix refers to the 5 unsupervised defect prediction models.

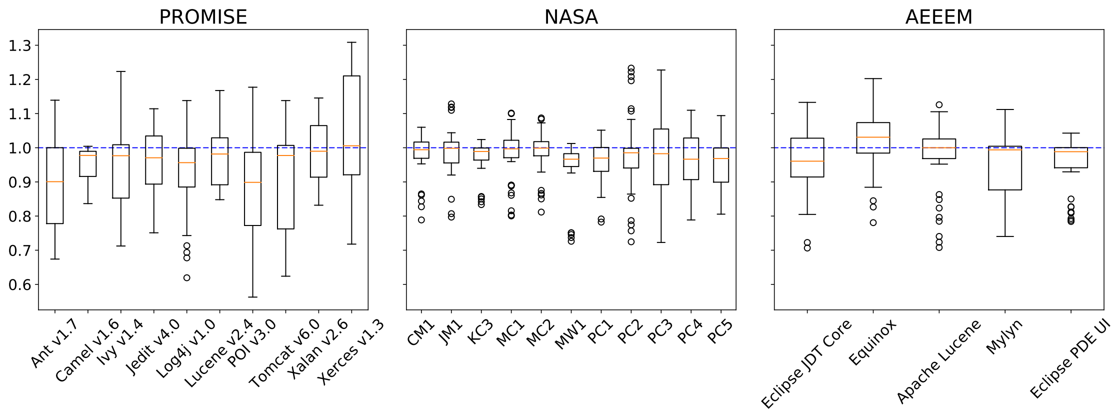
perform the unsupervised models with feature reduction techniques. Figure 3.5 shows the AUC after applying the feature reduction techniques to the supervised and unsupervised models. The supervised models significantly outperform the unsupervised models. Prior research [181] reported that spectral clustering (SC) is the only studied unsupervised defect prediction model that outperforms the supervised models.

The reason that the unsupervised models perform worse than the supervised models in Figure 3.5 is that we consider all the unsupervised models together, to be able to provide a more generic conclusion. However, as Figure 3.5 shows, some unsupervised defect prediction models perform better than others.

In the AEEEM dataset, the feature reduction techniques improve the predic-



(a) The supervised models



(b) The unsupervised models

Figure 3.6: The ratios of the AUCs of the supervised and unsupervised prediction models. Each boxplot contains 40 ratio values (5 prediction models * 8 feature reduction techniques). The dashed blue line indicates a ratio of 1.0. A ratio larger than 1.0 indicates that the feature reduction technique improved the AUC compared to the original models.

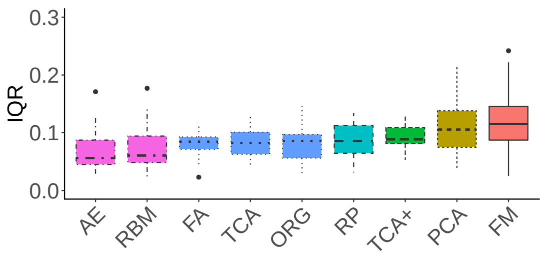
tion performance of the supervised models for most projects. We observe that the feature reduction techniques did not improve the prediction performance in many projects, as the median values of several boxplots in Figure 3.6 are lower than 1.0. However, the studied feature reduction techniques improved the prediction performance of the supervised models for many projects in the AEEEM dataset (Figure 3.6(a)). We further investigate this phenomenon in Section 3.4.3.

The neural network-based techniques improve the prediction performance of the supervised/unsupervised prediction models for most projects. Table 3.4 shows the median ratios for the feature reduction techniques. We observe that the neural network-based feature reduction techniques RBM and AE have the most gray cells for the supervised/unsupervised prediction models in combination with the feature reduction techniques.

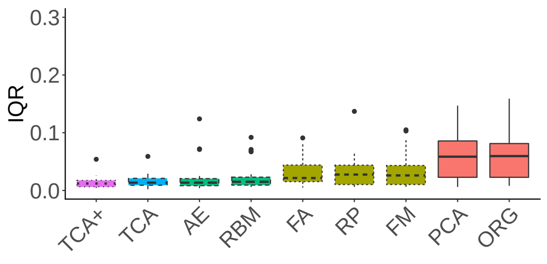
However, almost all feature reduction techniques did not improve the prediction performance in the NASA dataset except for FA with the unsupervised prediction models. FA combined with the unsupervised prediction models improved over half of the projects in the NASA dataset. We further investigate why the feature reduction techniques work well for the AEEEM dataset but not for the other datasets in Section 3.4.3.

3.4.2 RQ2: What is the impact of feature reduction techniques on the variance of the performance across defect prediction models?

Motivation: A challenge in applying defect prediction for practitioners is to select the best-performing model for their data from many possible defect prediction models [40,43]. In this RQ, we studied the variance in performance across defect prediction models of the studied feature reduction techniques for a particular dataset. If this variance is small, the practitioner does not need to worry about the choice at all, as the models perform similarly across datasets.



(a) The supervised models



(b) The unsupervised models

Figure 3.7: The Scott-Knott ESD test results for the IQR of the supervised and unsupervised models. Each color indicates a rank: feature reduction techniques in different ranks have a significant difference in variance (IQR). Each boxplot has 26 IQR values (one for each project). The x-axis refers to the feature reduction techniques; the y-axis refers to the IQR values.

Approach: We used the interquartile range (IQR) which captures the performance variance across the studied defect prediction models for a given project and feature reduction technique. We used the AUC values of all the studied supervised and unsupervised models for all bootstrap samples to conduct a new *bootstrap sampling* to calculate the IQR for each reduction technique and each project. We calculated an IQR value as follows:

1. Sample 100 AUC values at random from the 100 AUC values for each studied supervised/unsupervised model allowing for replacement.

2. Compute the median AUC value across the sampled 100 AUC values.

3. Repeat steps 1 and 2 100 times.

4. Compute the IQR value for the 500 sampled median AUC values (5 supervised/unsupervised prediction models * 100 median AUC values) for each feature reduction/selection technique for each studied project.

Where the IQR values are computed as follows:

$$\text{IQR} = Q_3 - Q_1$$

where Q_1 is the first quartile of the 500 sampled median AUC values, and Q_3 is the third quartile of the 500 sampled median AUC values. The first quartile is the median between the smallest and the median of the 500 AUC values, and the third quartile is the median between the median and the largest of the 500 AUC values. As we studied 26 projects, we have 26 IQR values for each feature reduction technique. We used the Scott-Knott ESD test to compare the distributions of IQRs for each feature reduction technique. Figure 3.7 shows the results of the Scott-Knott ESD test.

In addition, we compared the IQR values of the prediction models across the feature reduction techniques for each project. Table 3.5 shows the results of the IQR analysis at the project level. Each cell contains an IQR value that was computed from 500 bootstrapped median AUC values of the supervised and unsupervised prediction models.

Results: **The neural network-based feature reduction techniques, RBM and AE, generate features that result in less variance across the supervised models than the original features.** Figure 3.7(a) shows that the original features (ORG) are in the second rank, and RBM and AE belong to the first rank. Hence, RBM and AE significantly improve the variance of the performance across the supervised defect prediction models.

Almost all feature reduction techniques (except PCA) generate features that have a significantly smaller performance variance across the unsupervised models than the original features. Figure 3.7(b) shows that the unsupervised models that use features that were generated by PCA, or the original features are in the lowest rank. Hence, using the feature reduction techniques (except PCA) in combination with an unsupervised defect prediction model results in a small performance variance, which is helpful for practitioners.

The neural network-based feature reduction techniques improve the performance variance of the supervised models for the largest number of projects. Table 3.5(a) shows that the features that were generated by the neural network-

Table 3.5: The IQR ratio for the feature reduction techniques in the studied supervised and unsupervised prediction models. The gray cells refer to the ratios that are greater than 1.0. The "Improved" row indicates the number of gray cells in the column.

(a) The supervised models

		RBM	AE	PCA	FM	FA	RP	TCA	TCA+	CFS	ConFS
PROMISE	Ant v1.7	1.769	1.330	0.849	0.439	1.024	0.943	0.975	1.006	1.294	1.031
	Camel v1.6	3.885	3.027	0.842	0.809	0.774	0.777	1.312	1.357	1.637	1.466
	Ivy v1.4	1.841	2.964	1.978	2.197	2.001	1.151	3.641	1.233	1.340	2.072
	Jedit v4.0	1.018	1.180	0.562	0.501	0.771	0.595	0.688	0.718	0.767	0.683
	Log4j v1.0	1.335	2.643	1.030	5.212	1.388	1.898	1.252	1.189	1.361	1.377
	Lucene v2.4	1.737	0.661	0.554	0.473	0.635	1.141	0.666	0.647	0.800	0.658
	POI v3.0	0.973	0.586	0.628	0.241	0.959	1.642	0.894	0.977	1.268	1.157
	Tomcat v6.0	1.351	1.429	0.671	0.657	1.096	0.739	0.854	0.921	1.369	1.204
	Xalan v2.6	0.762	0.615	0.863	0.495	0.725	0.750	0.531	0.556	0.972	0.841
	Xerces v1.3	1.918	1.749	0.806	1.022	0.847	0.943	1.064	1.078	1.171	1.417
NASA	CM1	0.196	0.227	0.241	0.311	0.329	0.683	0.389	0.430	0.589	0.453
	JM1	2.502	2.579	1.886	0.604	0.804	0.972	1.036	0.758	0.892	0.986
	KC3	0.582	0.955	1.509	1.130	0.661	1.093	0.906	0.751	1.871	1.710
	MC1	0.639	0.670	0.605	0.734	0.980	0.810	1.227	0.967	0.831	2.011
	MC2	1.087	1.080	0.545	1.074	0.808	0.835	0.734	0.679	1.121	1.169
	MW1	1.199	1.451	0.722	4.598	2.933	1.166	2.113	1.697	1.792	1.274
	PC1	0.826	0.984	0.649	1.078	1.326	1.936	2.501	0.875	1.179	1.578
	PC2	0.541	0.565	0.454	0.788	1.067	1.317	0.937	0.898	0.796	0.389
	PC3	1.060	1.522	0.684	0.985	1.274	0.718	0.862	0.695	1.331	1.145
	PC4	0.885	1.504	0.548	0.397	0.852	1.055	0.681	0.750	5.539	1.138
	PC5	0.796	1.892	0.752	0.344	0.908	0.705	0.723	0.757	0.798	1.009
AEEEM	Eclipse JDT Core	1.332	1.433	1.246	0.434	1.556	0.998	1.373	0.970	1.072	1.393
	Equinox	1.989	2.927	1.543	0.821	1.076	0.822	1.075	1.129	1.438	1.337
	Apache Lucene	1.373	1.844	0.936	0.608	0.989	1.678	1.014	0.993	0.881	1.022
	Mylyn	5.981	4.024	0.940	0.946	5.923	1.155	1.289	1.282	1.992	2.406
	Eclipse PDE UI	1.574	1.208	1.096	0.224	0.392	0.374	0.434	0.384	0.475	0.452
	Improved	17	18	7	7	11	11	12	8	16	19

(b) The unsupervised models

		RBM	AE	PCA	FM	FA	RP	TCA	TCA+	CFS	ConFS
PROMISE	Ant v1.7	2.693	1.772	1.035	2.539	3.228	3.217	1.447	4.320	0.335	0.245
	Camel v1.6	1.224	1.715	1.331	0.961	1.659	0.778	0.503	0.680	0.209	0.301
	Ivy v1.4	2.881	4.073	0.924	2.096	3.690	3.628	3.317	3.076	2.809	2.153
	Jedit v4.0	3.221	3.229	1.012	1.704	4.387	2.213	3.346	4.496	0.671	0.790
	Log4j v1.0	3.400	3.445	1.221	1.346	0.542	0.977	1.562	0.783	0.762	0.793
	Lucene v2.4	0.460	0.982	0.653	0.796	0.577	1.865	0.945	0.560	0.176	0.163
	POI v3.0	0.764	0.582	1.152	7.759	1.425	7.589	4.332	2.842	0.415	0.927
	Tomcat v6.0	2.993	1.985	1.116	2.326	3.566	4.079	1.512	5.814	0.630	0.904
	Xalan v2.6	2.028	3.317	0.763	2.339	1.363	3.747	0.829	0.907	0.362	2.860
	Xerces v1.3	8.508	6.308	1.132	11.297	6.729	10.673	2.089	10.157	4.720	0.980
NASA	CM1	2.032	2.820	1.074	0.694	1.274	1.535	1.573	3.922	1.238	0.688
	JM1	1.206	5.044	0.988	0.983	1.360	1.304	39.761	154.852	1.210	1.009
	KC3	0.920	0.753	0.936	1.280	0.639	0.780	0.946	3.250	0.591	0.750
	MC1	5.284	3.674	1.035	0.576	3.833	1.261	5.611	3.635	1.132	0.971
	MC2	1.377	2.056	1.839	0.778	1.670	1.919	2.709	4.491	1.457	1.441
	MW1	2.059	1.207	0.952	1.043	0.897	1.008	1.325	1.520	0.814	0.816
	PC1	2.413	2.298	0.996	1.047	1.405	0.866	3.256	3.491	0.964	0.748
	PC2	2.058	1.900	1.014	2.441	5.610	1.035	5.055	7.867	2.231	1.436
	PC3	2.293	2.230	1.069	2.318	4.411	7.025	13.146	13.203	2.622	1.182
	PC4	4.032	3.431	0.917	4.188	1.266	3.476	9.617	10.161	1.653	1.177
	PC5	5.900	11.856	1.025	18.438	4.839	2.096	30.503	5.129	1.355	1.267
AEEEM	Eclipse JDT Core	15.497	15.262	0.973	1.727	1.290	3.088	20.680	8.202	1.104	1.505
	Equinox	3.504	7.741	0.684	3.573	1.155	1.420	6.460	9.422	0.676	0.601
	Apache Lucene	8.221	6.507	0.989	0.521	4.423	1.610	5.717	8.040	0.659	0.920
	Mylyn	10.940	9.597	1.038	8.177	1.099	1.036	17.600	11.010	0.980	0.889
	Eclipse PDE UI	3.237	2.806	1.068	0.690	1.235	0.465	5.627	6.956	0.389	0.366
	Improved	23	23	15	18	22	21	22	22	11	9

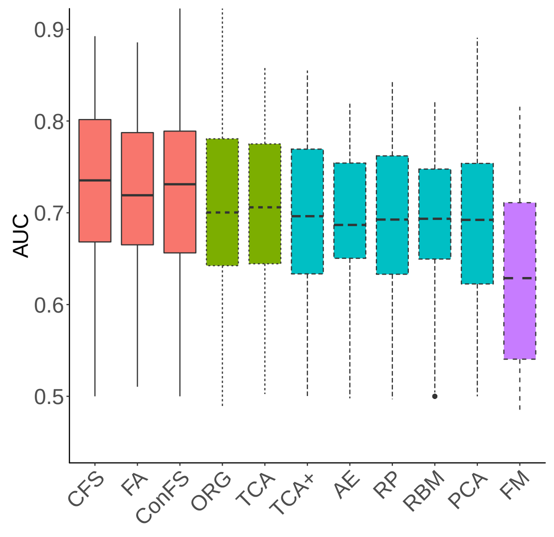
based feature reduction techniques (RBM and AE) improved the performance variance (IQR) across the studied supervised prediction models for the largest number of projects compared to the other feature reduction techniques, and the original models. RBM and AE also belong to the first rank of the overall performance variance result (Figure 3.7(a)).

The neural network-based feature reduction techniques improve the performance variance of the unsupervised models for the largest number of projects. Table 3.5(b) shows that the features that were generated by the neural network-based feature reduction techniques (RBM and AE) improved the performance variance across the studied unsupervised prediction models for the largest number of projects. Interestingly, in terms of overall performance variance, TCA and TCA+ belong to the first and the second rank (Figure 3.7(b)). However, the difference with the neural network-based feature reduction techniques is only small (TCA+) and negligible (TCA), according to the Cliff's delta effect size.

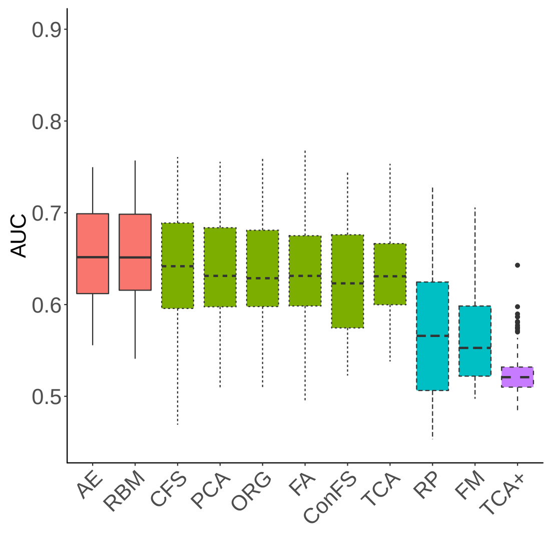
3.4.3 RQ3: How do feature selection techniques compare to feature reduction techniques when applied to defect prediction?

In this RQ, we compare feature reduction and selection techniques along two dimensions: the performance and the performance variance of the defect prediction models. We study the correlation-based (CFS) and consistency-based (ConFS) feature selection techniques, as they performed best according to prior studies [41, 175].

Motivation: In RQ1 and RQ2, we found that several feature reduction techniques (FA, RBM and AE) outperform the original features (ORG) in terms of performance or performance variance of the defect prediction models. Prior work [41, 175] showed that several feature selection techniques outperform the original models as well. In this RQ, we compare the performance (AUC) and



(a) The supervised models



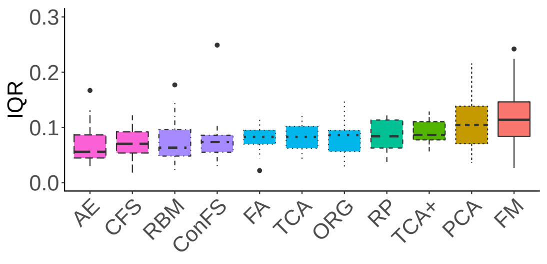
(b) The unsupervised models

Figure 3.8: The Scott-Knott ESD test results for the supervised (logistic regression, random forest, naive Bayes, decision tree, and logistic model tree) and unsupervised (spectral clustering, k-means, partition around medoids, fuzzy C-means, neural-gas) models. Each color indicates a rank: models in different ranks have a statistically significant difference in performance. Each boxplot has 130 median AUC values (5 defect prediction models times 26 projects). The x-axis refers to the feature reduction/selection techniques; the y-axis refers to the AUC values.

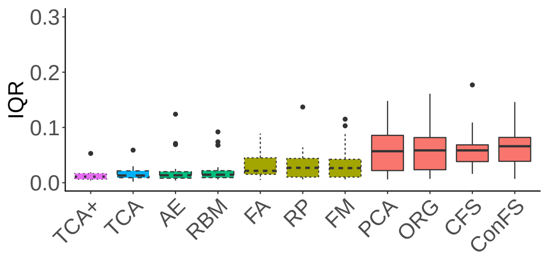
the performance variance (IQR) of the feature reduction and selection techniques when applied to defect prediction models.

Approach: The experimental procedure is the same as the procedures of RQ1 and RQ2 (only we use the two feature selection techniques CFS and ConFS instead of the feature reduction techniques).

Results: **The feature selection techniques (correlation-based feature selection (CFS) and consistency-based feature selection technique (ConFS)) significantly outperform the original features (ORG) in the supervised models, and perform as well as the feature agglomeration (FA) reduction technique.** Figure 3.8(a) shows the AUC values and the results of the Scott-Knott ESD test for



(a) The supervised models



(b) The unsupervised models

Figure 3.9: The Scott-Knott ESD test results for IQR in the supervised and unsupervised models. Each color indicates a rank: feature reduction/selection techniques in different ranks have a significant difference in variance (IQR). Each boxplot has 26 IQR values (one for each project). The x-axis refers to the feature reduction/selection techniques; the y-axis refers to the IQR values.

the supervised models after applying the studied feature reduction and selection techniques.[†] Each boxplot shows the median AUC values for the projects using a certain feature reduction/selection technique. CFS, ConFS and FA are in the highest rank by themselves, which indicates that the subsets of features that were selected by CFS or ConFS perform as well as the feature sets that were generated by FA for the supervised models.

The neural network-based feature reduction techniques (RBM and AE) significantly outperform the feature selection techniques (CFS and ConFS) for the unsupervised defect prediction models. The highest rank contains only the two studied neural network-based feature reduction techniques (Figure 3.8(b)). The studied feature selection techniques (CFS and ConFS) belong to the second rank, together with the original models (ORG). Hence, the studied feature selection techniques have a worse performance than the neural network-based feature reduction techniques for the unsupervised defect prediction models.

[†]Note that the ranks are slightly different from Figure 3.4 due to the fact that Scott-Knott ESD is a clustering algorithm, and hence affected by the total set of input distributions. For more information see `https://github.com/klainfo/ScottKnottESD`.

In the supervised models, applying the neural network-based feature reduction techniques, RBM and AE, or the feature selection techniques, CFS and ConFS, significantly outperforms the original models in terms of performance variance. The original models (ORG) belong to the third rank (Figure 3.9(a)). The neural network-based feature reduction techniques and the feature selection techniques belong to the first or second rank, hence they have a smaller performance variance than the original models.

In the unsupervised models, all feature reduction techniques (except PCA) significantly outperform the feature selection techniques in terms of performance variance. The feature selection techniques belong to the worst rank together with the original models (ORG) and PCA (Figure 3.9(b)). Hence, the studied feature selection had a larger performance variance than almost all the studied feature reduction techniques for the unsupervised defect prediction models.

Our above findings for RQ3 are confirmed by our project-level analysis. Table 3.4 shows the median ratios of the performance of each feature reduction/selection technique compared to the original models. We calculated this ratio as follows:

$$\text{ratio} = \frac{AUC_{FRS}}{AUC_{ORG}}$$

Where AUC_{ORG} is the AUC (the median AUC across all bootstrap samples) of a prediction model using the original features, and AUC_{FRS} is the AUC of a prediction model using the features that were generated/selected by a particular feature reduction or selection technique. We computed the median ratio across the five studied supervised and unsupervised prediction models. Table 3.4 confirms our above findings about the performance of the studied feature selection techniques compared to that of the feature reduction techniques.

Table 3.5 shows the IQR ratio values of each feature reduction/selection technique. We define this ratio as follows:

$$\text{ratio} = \frac{IQR_{FRS}}{IQR_{ORG}}$$

Where IQR_{ORG} is the IQR (the median IQR across all bootstrap samples) of a prediction model using the original features, and IQR_{FRS} is the IQR of a prediction model using the features that were generated/selected by a particular feature reduction or selection technique. We calculated the median IQR value for the supervised models using bootstrap samples as follows:

1. Sample 100 values following the distribution of the 100 AUC values for each studied supervised model while allowing for replacement.
2. Compute the median AUC value across the sampled 100 values.
3. Repeat steps 1 and 2 100 times.
4. Compute the IQR value for the 500 sampled median AUC values (5 supervised prediction models * 100 median AUC values) for each feature reduction/selection technique for each studied project.

We repeated the above procedure for the unsupervised models. Table 3.5(a) shows that the project-level results confirm our findings above, as the RBM and AE feature reduction techniques and the CFS and ConFS feature selection techniques improve the performance variance of most projects compared to the other techniques. In addition, Table 3.5(b) shows that all feature reduction techniques improve the performance variance of more projects than the CFS and ConFS feature selection techniques.

Why do feature reduction techniques work well in the AEEEM dataset?

Motivation: We observed that the feature reduction techniques work better for the projects in the AEEEM dataset than for the projects in the other datasets. Ghotra et al. [41] applied PCA to the data of each project to capture its richness. We use the same analysis to investigate whether the dataset richness is an explanation of why feature reduction techniques work better for the AEEEM dataset.

Approach: The idea behind Ghotra et al.'s analysis [41] is to generate features from a dataset using PCA that (together) retain at least 95% of the variance of the original dataset. Ghotra et al. reason that a larger number of generated features indicates a richer dataset. Likewise, they interpret that a small number of generated features indicates redundancy in the original dataset. We applied PCA to each project and counted the number of generated features.

Results: **The PROMISE, NASA, and AEEEM datasets have different data richness characteristics, however; the characteristics of the projects within each dataset are consistent.** Table 3.6 shows the number of generated features. While the number of generated features is approximately the same for the PROMISE and NASA projects, the proportion of generated features compared to the number of original features is different. In addition, this proportion is even lower for the AEEEM projects. Hence, we conclude that the datasets have different characteristics in terms of data richness. However, within each dataset, the projects have approximately the same richness characteristics.

The original features of the projects in the AEEEM dataset are more diverse than the projects of the other datasets. We observe that 36 principal components are needed to cover 95% of the variance of the Eclipse JDT Core project in Figure 3.10, compared to 12 components for the Ant project and 11 for the CM1 project. Hence, the original features of the AEEEM dataset are much more diverse than those of the PROMISE and the NASA datasets. The diversity of the AEEEM dataset could be a reason why feature reduction techniques improve the performance of this dataset.

Comparing feature selection and reduction techniques along the dimensions of understandability and execution time

The understandability of the features in a defect prediction model is important, as understandable features make the model, and its predictions, easier to explain [148]. Feature reduction techniques combine the original features into one

Table 3.6: The number of generated features (principal components) that are needed to account for 95% of the data variance.

Studied Dataset	**Studied Project**	**# of Studied Features**	**# of Generated Features**	**% of Generated Features**
PROMISE	Ant v1.7	20	12	60.0
	Camel v1.6	20	12	60.0
	Ivy v1.4	20	10	50.0
	Jedit v4.0	20	12	60.0
	Log4j v1.0	20	12	60.0
	Lucene v2.4	20	12	60.0
	POI v3.0	20	12	60.0
	Tomcat v6.0	20	12	60.0
	Xalan v2.6	20	12	60.0
	Xerces v1.3	20	12	60.0
NASA	CM1	37	11	29.7
	JM1	21	8	38.1
	KC3	39	10	25.6
	MC1	38	15	39.5
	MC2	39	11	28.2
	MW1	37	11	29.7
	PC1	37	12	32.4
	PC2	36	10	27.8
	PC3	37	13	35.1
	PC4	37	14	37.8
	PC5	38	15	39.5
AEEEM	Eclipse JDT Core	212	36	17.0
	Equinox	212	31	14.6
	Apache Lucene	212	33	15.6
	Mylyn	212	46	21.7
	Eclipse PDE UI	212	38	17.9

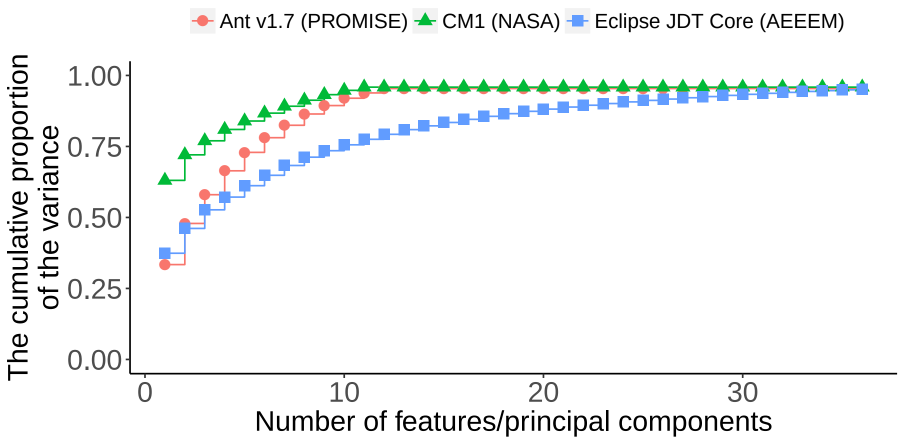


Figure 3.10: The number of principal components (features that were generated by PCA) that are needed to account for the original data variance for the Ant, CM1 and Eclipse projects in the PROMISE, NASA and AEEEM datasets. The x-axis indicates the number of principal components. The y-axis indicates the cumulative proportion of the variance. The other projects of the datasets showed a similar pattern.

or more newly-generated features. Hence, these newly-generated features are by definition harder to understand than the features that are a subset (i.e., they were *selected*) of the original features. We inspected the feature sets that were generated during our experiments, and we observed that almost all generated feature sets consist of features that are a complex combination of all available original features. Hence, defect prediction models that are generated using feature reduction techniques are harder to understand than those that use feature selection.

In addition, the execution time of a feature reduction/selection technique and a defect prediction model is important – models that take too long to build or execute are not very useful in practice. To conduct our experiments in a timely manner, we ran them on a cluster of servers in parallel. Hence, it is difficult to compare the execution time of the experiments. In general, the execution time of our feature reduction/selection techniques and defect prediction models was short (i.e., in the range of minutes). Therefore, execution time is not a very problematic metric for practitioners who wish to apply feature reduction or selection to their defect prediction models.

3.5 Discussion: Which features are generated by the feature reduction techniques?

In RQ1, RQ2, and RQ3, we observed that some feature reduction/selection techniques generate/select features that perform defect prediction better and less variance than the features that were generated/selected by other feature reduction/selection techniques. In particular, we found that RBM and AE outperform the other studied feature reduction/selection techniques for the unsupervised models. However, RBM and AE are less-performing feature reduction techniques than the original models (ORG) for the supervised models. In this discussion, we take a closer look at the generated features to investigate why neural network-based feature reduction techniques perform well for the unsupervised defect prediction models, but not for the supervised models. In this section, we discuss possible explanations for the differences in AUC and variance of the performance.

Approach: We focused our discussion on the RBM and AE feature reduction techniques, as these techniques generate new features by assigning (combinations of) weights to the original features. For example, a newly generated feature may be generated by 0.5 times original feature 1 and 0.5 times original feature 2. These weight sets allow us to study how the new features are related to the original features, and to the features that were generated by other feature reduction techniques, and extract possible explanations for the improved and small variance of the performance. RP and PCA also generate new features by assigning weights to the original features, however, the weights of RP are randomly generated, and PCA generates a different number of features for each project, which makes them difficult to compare. Hence, we focused our discussion on RBM and AE. As we generated 100 new feature sets of 10 features using

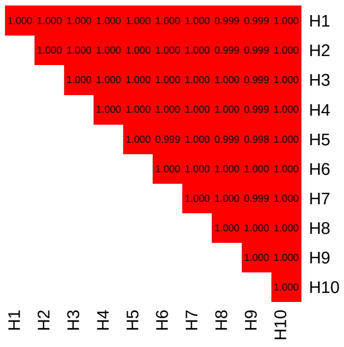
RBM and AE, we generated 1,000 weight sets using these two feature reduction techniques for each project. For each feature reduction technique, we randomly selected 10 (out of the 100) generated feature sets for our investigation.

We conducted correlation analysis and clustering analysis on the studied feature sets to study their similarity within and across projects. The correlation analysis shows how independent the features that are generated for a project are. Highly correlated features can negatively impact the performance of regression models [37] and this effect can affect our supervised models as well. We first calculated the Spearman rank correlation [188] between the generated features in a feature set within a project. Each generated feature was normalized using the *z*-score. We chose Spearman rank correlation because it is non-parametric, and therefore requires no assumption about the distribution of the studied data.

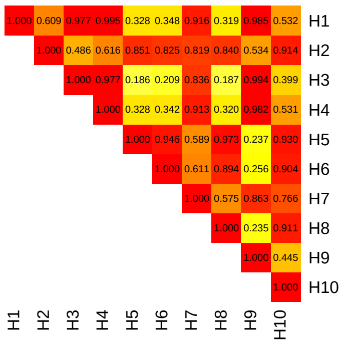
To study the similarity of the generated feature sets across projects within a dataset, we compared the weight sets of the generated features. First, we normalized all weights using *z*-score normalization. Second, we used *k*-means to cluster the weight sets of the features, and then we used t-distributed stochastic neighbour embedding (t-SNE) [168] to visualize the clustering results. t-SNE is commonly used for visualizing high dimensional features in scatter plots [167]. In particular, t-SNE models high-dimensional objects (i.e., feature sets) by two- or three-dimensional points such that similar objects are close, and dissimilar objects are further away from each other.

The goal of our clustering analysis is to find out how similar the generated features are across projects within a dataset. Hence, we configured *k*-means to search for 10 clusters (as we are generating 10 new features). We visualized the clustering results using the default settings of t-SNE.

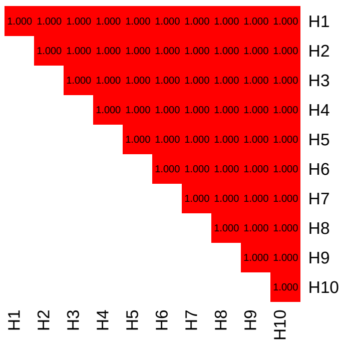
Results: **RBM generates feature sets in which all features are strongly correlated with each other.** Figure 3.11(a) shows the Spearman rank correlation of one set of RBM-generated features for the Ant project in the PROMISE dataset using a heatmap.



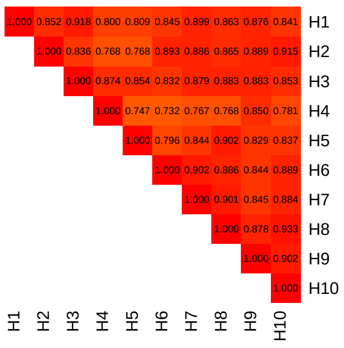
(a) RBM-generated features in the Ant project in the PROMISE dataset.



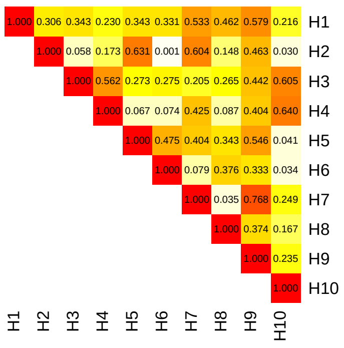
(b) RBM-generated features in the Eclipse project in the AEEEM dataset.



(c) RBM-generated features in the Eclipse project using change features in the AEEEM dataset.



(d) AE-generated features in the Log4j project in the PROMISE dataset.



(e) AE-generated features in the Camel project in the PROMISE dataset.

Figure 3.11: The Spearman rank correlation of generated features in the studied datasets. The darker colours indicate a strong absolute correlation (close or equal to one). The lighter colours indicate a weak absolute correlation (close or equal to zero).

We observe that all correlations are close to 1 (dark red), which means that all RBM-generated features in the feature set are strongly correlated (and hence similar) to each other. We observe similar correlations for the other studied feature sets for the PROMISE and NASA datasets. RBM generates weakly correlated features for several projects in the AEEEM dataset (e.g., for the Eclipse project: Figure 3.11(b)). However, if we use a smaller set of original features from that dataset, such as only the change features, RBM generates strongly correlated features for these projects as well (e.g., for the Eclipse project: Figure 3.11(c)).

We observe similar correlations in feature sets that were generated by AE. However, the correlation within the AE-generated feature sets appeared to be linked to the specific project. For example, AE generates sets of features that are strongly correlated to each other for the Log4j project (see Figure 3.11(d)), but features that are not as strongly correlated for the Camel project (see Figure 3.11(e)). Hence, a possible explanation of the reason for the small variance of the performance of features that were generated by the neural network-based feature reduction techniques (i.e., RBM and AE) could be the strong correlation within the generated feature sets.

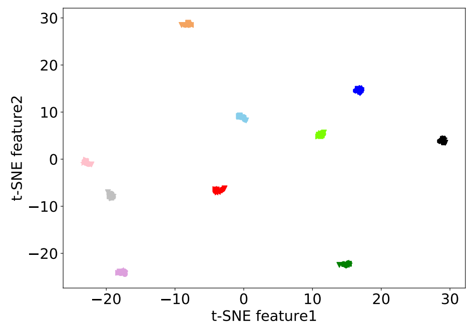
RBM mostly generates the same feature sets across projects. Figure 3.12(a) shows the *k*-means clustering result for one bootstrap sample of the RBM-generated weight sets in the PROMISE dataset using t-SNE [167]. Each shape refers to a project, and each color refers to a cluster that was identified by *k*-means clustering. The x-axis and y-axis refer to the t-SNE features that were generated from the 260 RBM-generated weight sets (10 weight sets on each project) by t-SNE. Hence, if there are 10 clearly identifiable groups of different shapes with the same colour in the t-SNE plot, we can conclude that the generated weight sets (and hence the features) are the same across projects. We observe that each colored cluster contains all shapes, which indicates that each cluster contains all projects. Hence, in this bootstrap sample, RBM generated the same feature sets across projects in the PROMISE dataset. We noticed a similar pattern for the other projects in the

PROMISE dataset for the other bootstrap samples.

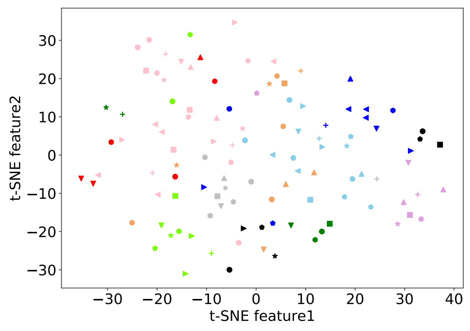
Figure 3.12(b) shows the result for the RBM-generated weight sets in the NASA dataset. The figure shows that RBM generated different feature sets across projects in the NASA dataset. A possible reason is that the original features in the NASA dataset are different in each project (Table 3.1).

Figure 3.12(c) shows the result for the RBM-generated weight sets in the AEEEM dataset. The figure shows that RBM generated different feature sets across projects in the AEEEM dataset. The explanation is similar to our observation during the correlation analysis. If we use one type of features, such as only the change features, Figure 3.12(d) shows that RBM generates the same feature sets across projects. However, if we use complexity code change features, we observe that RBM generates different feature sets across projects (Figure 3.12(e)). AE generated different features across projects (Figure 3.12(f), 3.12(g) and 3.12(h)).

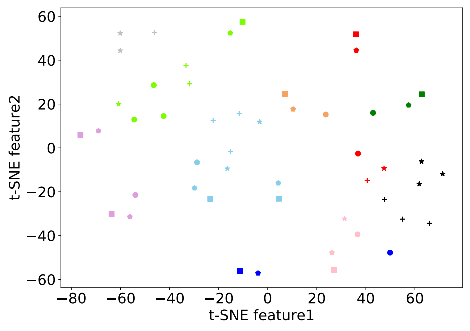
From the discussion results, we can extract several possible explanations for the fact that RBM and AE significantly improve the variance of the performance across the unsupervised models, but not across the supervised models, and for why these feature reduction techniques perform well for the unsupervised models. As the studied neural network-based feature reduction techniques appear to generate strongly correlated features for a project, these feature sets suffer from multicollinearity [37], which is known to negatively affect the performance of the supervised models. However, as the unsupervised defect prediction models do not need to be trained, these models are not be affected by the multicollinearity problem. In addition, because RBM generates strongly correlated features for a project, the unsupervised models become much simpler, which seems to improve the variance of the performance across the unsupervised defect prediction models.



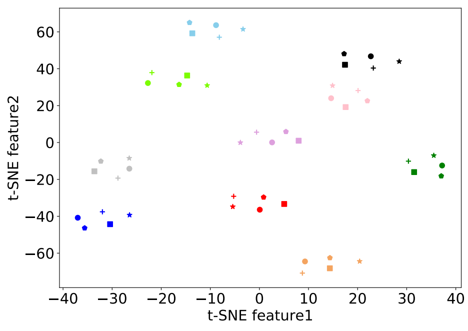
(a) RBM-generated weight sets in the PROMISE dataset.



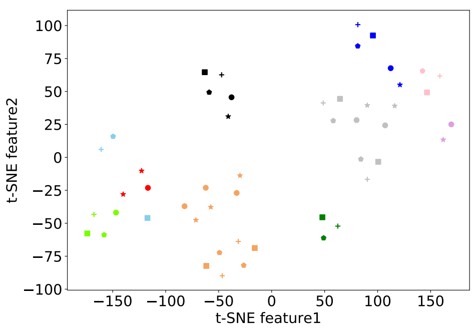
(b) RBM-generated weight sets in the NASA dataset.



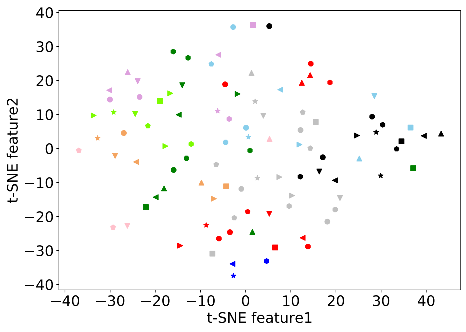
(c) RBM-generated weight sets in the AEEEM dataset.



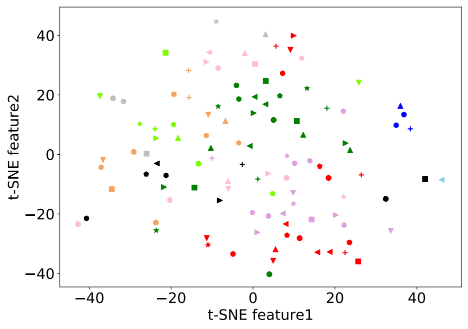
(d) RBM-generated weight sets using Change features in the AEEEM dataset.



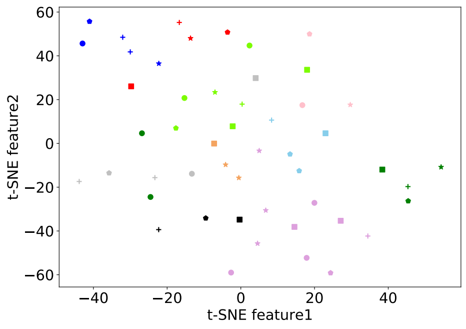
(e) RBM-generated weight sets using complexity code change features in the AEEEM dataset.



(f) AE-generated weight sets in the PROMISE dataset.



(g) AE-generated weight sets in the NASA dataset.



(h) AE-generated weight sets using Change features in the AEEEM dataset.

Figure 3.12: *k*-means clustering results with t-SNE for generated weight sets in the studied datasets. Each shape represents a project, and each color represents a cluster.

3.6 Threats to validity

3.6.1 External validity

With regards to the generalizability of our results, we applied our experiments to three publicly available datasets. These studied datasets (PROMISE, NASA and AEEEM) were all used in many prior defect prediction studies. The projects in these studied datasets span different domains, include both open source and industrial projects and contain different features. Future studies are necessary to investigate whether our results generalize to other projects.

In addition, we studied only a subset of the many existing feature reduction and selection techniques and defect prediction models. We carefully selected techniques and models that have been used before for defect prediction, and that have an implementation readily available. Without such an implementation, it is difficult and time-consuming to ensure that the implementation matches the one used in prior studies. Future studies are necessary to investigate whether our results apply to other feature reduction/selection techniques and defect prediction models.

3.6.2 Internal validity

In our experiments, we used AUC as a performance measure. AUC is a popular performance measure for defect prediction, as it does not require a threshold. However, different software project teams may have different objectives. Hence, future studies should investigate the impact of feature reduction techniques on other performance measures, while keeping in mind the possible pitfalls of studying threshold-dependent performance measures [157].

When using the out-of-sample bootstrap sampling, we encountered computational errors in two bootstrap samples. The errors occurred because two bootstrap samples violated requirements of the NB and SC models (e.g., a generated feature had a variance of zero which violates a requirement of the NB model). To mitigate this threat, we discarded these bootstrap samples and generated a new sample instead.

We provide all experimental scripts that we used in our study.[‡] This replication package allows researchers and practitioners to replicate our experiments and confirm our results.

3.7 Chapter summary

In defect prediction, reducing the number of features is an important step when building defect prediction models [10, 37, 146, 149]. Prior studies indicated that reducing the number of features avoids the problem of multicollinearity [37] and the curse of dimensionality [10]. Feature selection and reduction techniques help to reduce the number of features in a model. Feature selection techniques reduce the number of features in a model by selecting the most important ones, while feature reduction techniques reduce the number of features by creating new, combined features from the original features.

Prior work [41, 175] studied the impact of feature *selection* techniques on defect prediction models. Our work is the first large-scale study on the impact of feature *reduction* techniques on defect prediction models. In particular, we studied the impact of eight feature reduction techniques on five supervised and five unsupervised defect prediction models. In addition, we compared the impact of feature reduction techniques on defect prediction with the impact of the two best-performing feature selection techniques (according to prior work).

We studied the impact of feature reduction/selection techniques on defect prediction models along two dimensions:

1. The defect prediction performance (AUC) of the features that are generated by the feature reduction/selection techniques, to study whether feature

[‡]`https://sailhome.cs.queensu.ca/replication/featred-vs-featsel-defectpred/`

reduction/selection techniques can improve the performance of defect prediction models.

2. The variance of the AUC across defect prediction models that use the features that are generated by feature reduction or selected by feature selection techniques. It is difficult to select the best performing model for each project, since the best model may change per project [40, 43]. Hence, we studied whether feature reduction or selection techniques can relieve the burden for practitioners of having to choose the best performing defect prediction model for their data.

Below, we summarize the main recommendations that follow from our work.

Recommendation 1: For the supervised defect prediction models, use the correlation-based (CFS) or consistency-based (ConFS) feature selection techniques. Our experiments in RQ3 show that, for the supervised models, CFS and ConFS outperform the feature reduction techniques (except feature agglomeration (FA)) and the original models. While FA has a similar performance, CFS and ConFS have a smaller performance variance. Hence, using CFS or ConFS in combination with a supervised defect prediction model allows practitioners to improve the performance of their defect prediction models, while making the choice for a particular defect prediction model easier as well.

Recommendation 2: For the unsupervised defect prediction models, use a neural network-based technique (Restricted Boltzmann Machine (RBM) or autoencoder (AE)). Our experiments in RQs 1 and 3 show that the RBM and AE feature reduction technique can significantly improve the performance of the unsupervised defect prediction models compared to the other feature reduction/selection techniques and the original models. In addition, we observed in RQs 2 and 3 that RBM and AE significantly improve the performance variance across the unsupervised models (except compared to the transfer component analyses (TCA and TCA+)). While the transfer component analyses (TCA and TCA+) have the smallest performance variance, they have a worse performance than RBM and AE. The effect size (Cliff's Delta) between the transfer component analyses and the neural network-based feature reduction techniques is negligible or small for the performance variance in favour of the transfer component analyses, but small (TCA) or large (TCA+) for the performance in favour of the neural network-based techniques. Hence, using a neural network-based feature reduction technique to preprocess the data of the unsupervised defect prediction models can improve both their performance and relieve the burden for practitioners of having to select the best-performing unsupervised defect prediction model for their project.

Recommendation 3: If a project has diverse data, the neural network-based techniques (Restricted Boltzmann Machine (RBM) or autoencoder (AE)) are likely to improve its defect prediction performance and performance variance. Our experiments showed that RBM and AE consistently improve the AUC and IQR of projects in the AEEEM dataset, for both supervised and unsupervised models. Practitioners should run PCA on their data to identify the diversity of their project's data (similar to what we did in Section 3.4.3). If the data turns out to be rich, RBM and AE are good options to improve the defect prediction performance and variance.

Chapter 4

The Impact of Context Features on Just-In-Time Defect Prediction

An earlier version of this chapter is published in the Empirical Software Engineering Journal (EMSE) [83].

4.1 Introduction

Software developers have limited resources to verify and test their source code. If developers can identify defective components (e.g., files or commits) they would be able to focus their effort on these components. *Defect prediction* supports this activity, and prior work has reported that defect prediction can reduce development cost for developers [162].

There exists plenty of work aimed at predicting defective components [9,28, 54,77,110]. In particular, several prior research work has focused on predicting defective changes called *change-level defect prediction*—also called *just-in-time defect prediction* [38,71,76,106]. Just-in-time defect prediction has the advantage that it can determine if a commit is likely to be defective when the commit is being performed [55] and providing faster feedback than other defect prediction methods [71]. Previous research has used features based on measuring the code changes (e.g., churn–*changed lines*) in just-in-time defect prediction [71,76,106].

To the best of our knowledge, no studies have considered using the information in the lines that surround the changed lines of a commit, which we call *context lines*. Our main hypothesis is that information in the context lines has an impact on the likelihood that the change is defective. In this chapter, we evaluate the use this information in just-in-time defect prediction. The dictionary defines context as "the parts of something written or spoken that immediately precede and follow a word or passage and clarify its meaning" [153]. In this chapter, we define the *context lines of a chunk of changed lines* as the *n-lines* ($n = 1, 2, \cdots$) *that precede the chunk and the n-lines that follow the chunk.*

This chapter proposes several *context features*. The different features vary around three different axis: a) how many context lines around each change to use (the size of the context, n), b) whether to use all context lines, or only those of added or removed lines (the type of the change), and c) counting the number of words or counting the number of keywords (as defined by the programming language) in the context. We consider these axes as the parameters of context features. We refer to a context feature which uses a set of the parameters as a variant of context features. We empirically study the best-performing variant in terms of defect prediction performance. We also compare the context features that are the best-performing variants with traditional *code churn features* (*change features* [71,76,106] and *indentation features* [59]), *extended context features* and combination features that use two extended context features. Indentation features use the total number of white spaces in front of changed lines, and the total number of pairs of braces that surrounded changed lines; we handle indentation features as code churn features, since they are computed on changed lines. In order to improve the predicting power of the context features in defect prediction, we also define extended context features. *Extended context features* count the number of words/keywords in both, the context lines and the changed lines. Hence, extended context features are hybrids of the context features and traditional code churn features. In addition, we use *combination features* that use two extended context features that count (1) number of words and (2) number of a certain keyword (e.g., "goto") at a prediction model in order to improve the predicting power of the extended context features in defect prediction.

Using six large open source software projects (from different domains) we empirically evaluate the defect prediction power of context features and compare them against traditional change features. This comparison is done using logistic regression models and random forest models.

Specifically, we address the following three research questions:

RQ1: What is the impact of the different variants of context features on defect prediction?

RQ2: Do context lines improve the performance of defect prediction?

RQ3: What is the impact of combination features of context features on defect prediction?

The main findings of this chapter are as follows:

```
int calculate(double value1, double value2){
      ...
      cons = 10;
 +    sum = value1*value2 + cons;

      ...
}
```

(a) Simple context lines.

```
int calculate(double value1, double value2){
      ...
      if (sum > 10) {
 +          sum = value1*value2 + cons;
      } else if (sum==10) { sum = cons;}
      ...
}
```

(b) Complex context lines.

Figure 4.1: An example of two changed functions each of which has one changed line (in this case, an added line, in bold). We call the lines that precede or follow the changed lines *context lines* (in italic with an underline). Other lines except the context lines are same in both functions.

- The best performing context features are the ones that measure the context of added-lines only.

- The prediction power of context features varies when different sizes of the context (number of lines around the change) are used. The optimal size of the context for the feature that uses number of words is smaller than the optimal size for the feature that uses keywords.

- The number of "goto" statements in context lines and changed lines is a good indicator of defective commits.

- Our proposed combination features of extended context features significantly outperform all the features that are used in this chapter, and achieve the best-performing features in all of the studied projects in terms of 2 of the 3 evaluation measures used (area under the receiver operation characteristic curve, and Matthews correlation coefficient).

4.2 Motivating example

Let us start from a simple example to illustrate the use of context lines to measure the complexity of changes. Figure 4.1 shows an example of two changed functions. The context lines are lines that precede or follow the changed lines. In this example, the underlined text represents the context lines and the bold lines are the changed lines. The function shown in Figure 4.1(a) has simple context lines: there is one assignment before the changed line and one empty line after the changed line. The changed function in Figure 4.1(b) has more complex context lines: the "if" and "else" statements. If we use only the changed lines as an input to compute the complexity of the changes these two changes have the same complexity. In contrast, if we use the context lines as a measure of complexity, these two functions have a different complexity.

To the best of our knowledge, there exists no research work that studies the context lines in defect prediction. In this chapter, we introduce two types of new features that use the context lines: context features and extended context features, and evaluate their performance in defect prediction.

There are complexity features, such as Halstead's complexity features [50] and McCabe's Cyclomatic complexity features [98], that can capture the complexity of the function being changed and take into consideration the context; however, (1) to compute these features we need all the lines of the functions, (2) these features are limited because they require a parser, and (3) complexity features are not optimized for code churn. In contrast, context features provide several advantages; first, they are easy to compute (they only require the "diff" and—in the case of number of keywords—a list of keywords of the programming language as input) and they measure only the complexity that surrounds the change instead of the entire function.

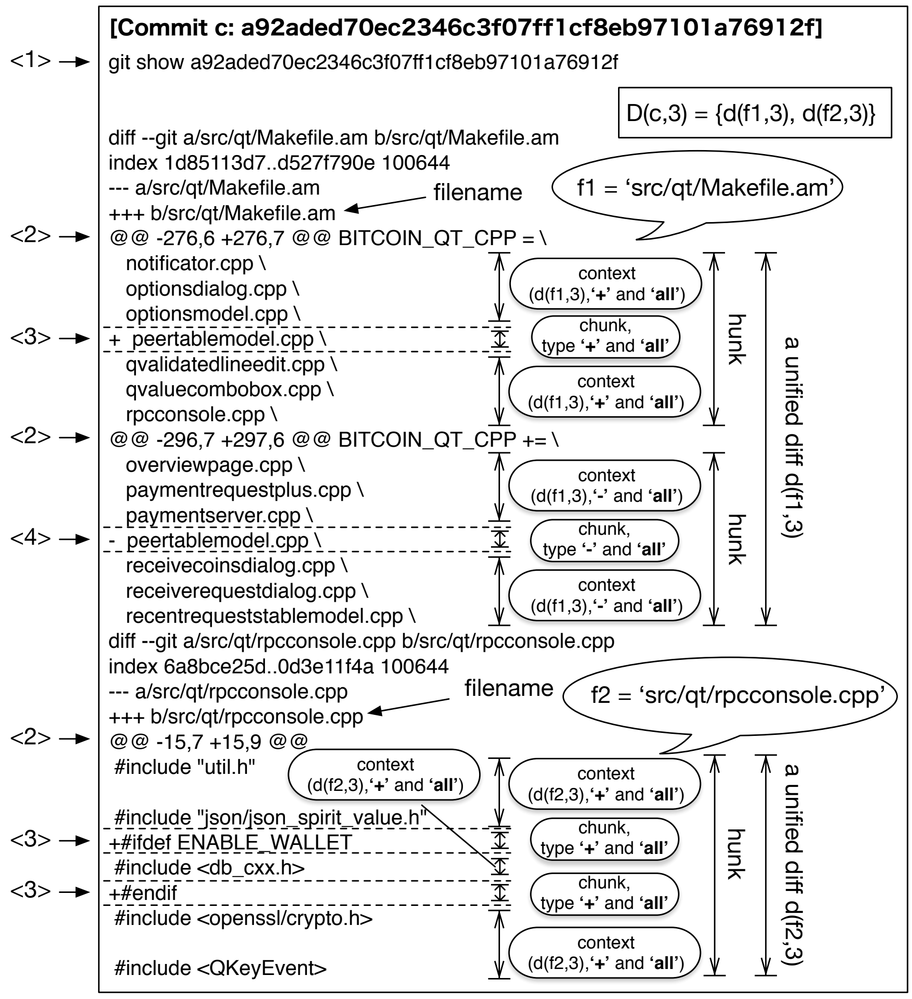
4.3 Context features

In this section, we describe the implementation of the proposed context features. As described in the previous sections, context information might be useful for defect prediction since it provides a new perspective of changes. In addition, it is easy to obtain context information (e.g., using the diff command in the version control system). For example, for the changed function in Figure 4.1(b), we consider only the lines in italic with an underline for context information.

Any modifications to a file can be described in terms of a *unified diff*. A unified diff is a sequence of *hunks*; each hunk is composed of one or more sequences of contiguously changed lines. Each of these sequences is composed of '+' lines (lines added to the file) or '-' lines (lines removed from the file). For the sake of simplicity, we refer to these sequences of changed lines as *chunks*. We consider two types of chunks: '+' chunks (which contain at least one '+' line), '-' chunks (which contain at least one '-' line). Finally, we will refer to any chunks (including both '+' and '-' chunks) as 'all' chunks. Figure 4.2 shows an example of two unified diffs (a part of output by `git show`). The above unified diff is a sequence of two hunks that are divided by the lines prefixed with @@, <2>. Each hunk has a chunk <3> and <4>, respectively. The above chunk, <3>, is of type '+' and 'all'. The below chunk, <4>, is of type '-', and 'all'. The below unified diff has a hunk. This hunk includes two chunks that are type '+' and 'all' [*].

Each chunk is surrounded by its context lines (the lines above and below the chunk that indicate where the chunk is to be applied—prefixed with ' ' in the hunk). We refer to these context lines as the context of the chunk. We also consider as a part of the context the full filename of the file being changed. This is because we consider that the directories where the file is located can contribute to the complexity of the context; i.e., more directories in the filename indicate a more complex context than no-directories. We evaluated the use or the

[*]Note that a chunk is able to be of type '+', '-' and 'all' at once. In this case, a chunk includes at least two lines that consist of at least one '+' and '-' line.



```
[Commit c: a92aded70ec2346c3f07ff1cf8eb97101a76912f]
git show a92aded70ec2346c3f07ff1cf8eb97101a76912f

diff --git a/src/qt/Makefile.am b/src/qt/Makefile.am
index 1d85113d7..d527f790e 100644
--- a/src/qt/Makefile.am
+++ b/src/qt/Makefile.am
@@ -276,6 +276,7 @@ BITCOIN_QT_CPP = \
   notificator.cpp \
   optionsdialog.cpp \
   optionsmodel.cpp \
+  peertablemodel.cpp \
   qvalidatedlineedit.cpp \
   qvaluecombobox.cpp \
   rpcconsole.cpp \
@@ -296,7 +297,6 @@ BITCOIN_QT_CPP += \
   overviewpage.cpp \
   paymentrequestplus.cpp \
   paymentserver.cpp \
-  peertablemodel.cpp \
   receivecoinsdialog.cpp \
   receiverequestdialog.cpp \
   recentrequeststablemodel.cpp \
diff --git a/src/qt/rpcconsole.cpp b/src/qt/rpcconsole.cpp
index 6a8bce25d..0d3e11f4a 100644
--- a/src/qt/rpcconsole.cpp
+++ b/src/qt/rpcconsole.cpp
@@ -15,7 +15,9 @@
 #include "util.h"

 #include "json/json_spirit_value.h"
+#ifdef ENABLE_WALLET
 #include <db_cxx.h>
+#endif
 #include <openssl/crypto.h>

 #include <QKeyEvent>
```

<1>
<2>
<3>
<2>
<4>
<2>
<3>
<3>
D(c,3) = {d(f1,3), d(f2,3)}
filename
f1 = ‘src/qt/Makefile.am’
context (d(f1,3),‘+’ and ‘all’)
chunk, type ‘+’ and ‘all’
context (d(f1,3),‘+’ and ‘all’)
hunk
a unified diff d(f1,3)
context (d(f1,3),‘-’ and ‘all’)
chunk, type ‘-’ and ‘all’
context (d(f1,3),‘-’ and ‘all’)
hunk
filename
f2 = ‘src/qt/rpcconsole.cpp’
context (d(f2,3),‘+’ and ‘all’)
context (d(f2,3),‘+’ and ‘all’)
chunk, type ‘+’ and ‘all’
chunk, type ‘+’ and ‘all’
context (d(f2,3),‘+’ and ‘all’)
hunk
a unified diff d(f2,3)

Figure 4.2: An example of unified diffs of a commit with context size equal to three produced by `git show` (<1>) in Bitcoin project; due to the space limitation, we remove the metadata of this commit (the commit comment and the author information). This commit consists of two source code file diffs. The above diff has two hunks (divided by the lines prefixed with @@, <2>). Each of both hunks consists of only one chunk (sequence of changed lines). The first chunk is of type ‘+’ and ‘all’. The second one is of type ‘-’, and ‘all’. The below diff has a hunk. This hunk consists of two chunks. Each of both chunks is of type ‘+’ and ‘all’. The context lines of each chunk are the above and below the corresponding chunk (above and below of <3> and <4>). The filename is prefixed with ‘+++ b/’.

filename/directories in the context features for their prediction power and found that when used, the performance of the context features improved.

For explaining context features, we define the following terminology:

- c: a commit.
- n: a *context size* that is the maximum number of lines that can precede or follow a chunk we consider. (This is also a parameter of the diff command in the version control system.)
- $d(f,n)$: a unified diff of a changed file f with *context size* n.
- $D(c,n)$: a set of $d(f,n)$ for all the changed files in commit c.

For a given unified diff $d(f,n)$, we define the three types of contexts, based on the three chunk types, with the following notation (refer to Table 4.1):

- $context(d(f,n),t)$: the concatenation of the full filename of f and the context of all chunks of chunk type t in diff $d(f,n)$.

For a unified diff $d(f,n)$, we define the following two notations:

1. $ncw(d(f,n),t)$: the number of words in $context(d(f,n),t)$.
2. $nckw(d(f,n),t)$: the number of programming language keywords (Table 4.2 shows all studied keywords)[†] in $context(d(f,n),t)$.

Given a commit c, a context size (the number of context lines) n, and the chunk type t, we define the following two kinds of context features:

$$NCW(c,n,t) = \sum_{d(f,n)\in D(c,n)} ncw(d(f,n),t),$$

[†]The keywords refer to reserved words (statements) in C++ that are shown by Microsoft Visual Studio [103]. Because the reserved words of C++ and Java are almost the same, we use the keywords for the projects in Java. We separate the reserved words that include underscores. For instance, we convert "__if_exists" into "if" and "exists".

Table 4.1: Types of contexts. The context of chunk type t of a unified diff $d(f,n)$ is the concatenation of the full filename of f and the contexts of the chunk type t in the diff $d(f,n)$.

Types of contexts	**Definition**
$context(d(f,n)$, all)	Context of all chunks in diff $d(f,n)$
$context(d(f,n)$, +)	Context of all chunks in diff $d(f,n)$ that contain at least one '+' line
$context(d(f,n)$, -)	Context of all chunks in diff $d(f,n)$ that contain at least one '-' line

Table 4.2: Studied programming language keywords.

break	case	catch	continue	default
do	else	except	for	goto
finally	if	exists	not	leave
return	switch	throw	try	while

$$NCKW(c,n,t) = \sum_{d(f,n)\in D(c,n)} nckw(d(f,n),t).$$

The defined context features are described in Table 4.3. To compute the context features of a commit $m(c,n,t)$ —where m is either NCW or $NCKW$, c is a commit id, n is the number of context lines, and t is the chunk type—we use the following algorithm:

1. Compute the diffs $D(c,n)$ of the source code files[‡] of commit c with the given number of lines of context, n, using the following command:
   `git show --unified=n c`
2. For each diff $d(f,n)$ of a source code file, compute $ncw(d(f,n),t)$ or $nckw(d(f,n),t)$:
   (a) Remove all chunks that are not of chunk type t, including their contexts.
   (b) Remove comments.

[‡]Here, a source file is a file with the name ending in java, c, h, cpp, hpp, cxx, or hxx, since we analyze both C++ and Java.

Table 4.3: Different context features. "Keywords" refers to the keywords defined in the programming language of the source code. c denotes a commit id, n denotes the context size (size of the context of the diff), and t is either of 'all', '+' or '-'.

Features	**Description**
$NCW(c, n, t)$	Sum of the number of words in the contexts of all chunks of chunk type t.
$NCKW(c, n, t)$	Sum of the number of programming language keywords in the contexts of all chunks of chunk type t.

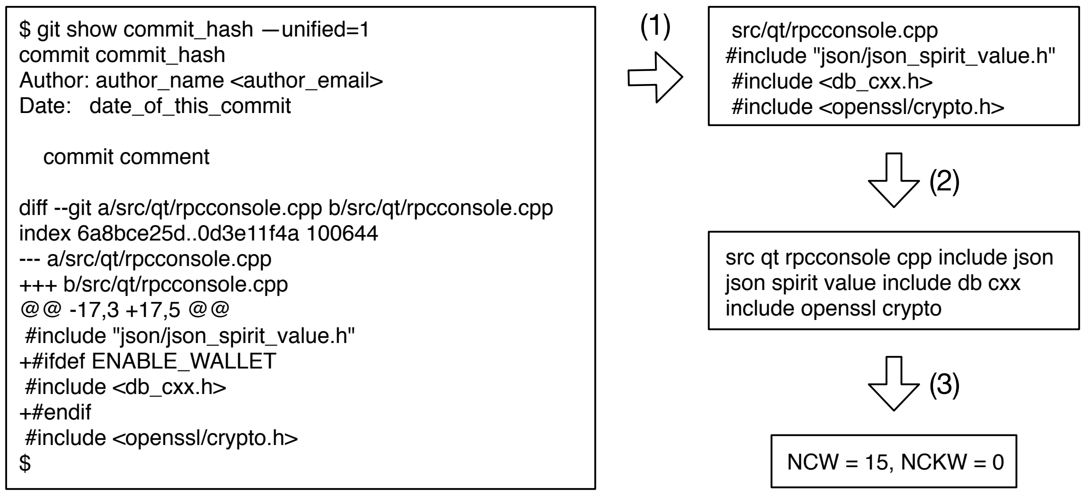
(c) Create a string *st* with the concatenation of
  - the full filename of the diff $d(f, n)$, and
  - the contexts around the identified chunks.

(d) Use `lscp`[§] [163] to convert *st* into a sequence of words. For *ncw*, count the number of words in this sequence; for *nckw*, count the number of programming language keywords in *st*.

3. Finally, the context feature *NCW*/*NCKW* of the commit is calculated as the sum of values of *ncw*/*nckw* for all diffs of the source code files in the commit.

Figure 4.3 depicts an example showing how the context features are computed from a unified diff. The left square corresponds to the first step in our algorithm. (1) and (2) are corresponding to the second step; we have removed unrelated code in (1), and convert the string into a sequence of words by `lscp` in (2). (3) is corresponding to the step three; we compute the context features.

[§]https://github.com/doofuslarge/lscp. `lscp` separates complex identifiers into its component words —e.g., converts *GetBoolArg* into *Get*, *Bool*, *Arg*).



```
$ git show commit_hash —unified=1
commit commit_hash
Author: author_name <author_email>
Date:   date_of_this_commit

    commit comment

diff --git a/src/qt/rpcconsole.cpp b/src/qt/rpcconsole.cpp
index 6a8bce25d..0d3e11f4a 100644
--- a/src/qt/rpcconsole.cpp
+++ b/src/qt/rpcconsole.cpp
@@ -17,3 +17,5 @@
 #include "json/json_spirit_value.h"
+#ifdef ENABLE_WALLET
 #include <db_cxx.h>
+#endif
 #include <openssl/crypto.h>
$
```

```
 src/qt/rpcconsole.cpp
#include "json/json_spirit_value.h"
 #include <db_cxx.h>
 #include <openssl/crypto.h>
```

```
src qt rpcconsole cpp include json
json spirit value include db cxx
include openssl crypto
```

```
NCW = 15, NCKW = 0
```

Figure 4.3: Example showing how NCW and NCKW are computed from a unified diff. The unified diff corresponds to the change from Figure 4.2; due to the space limitation, we remove several hunks, the commit comment, the author information, and the commit hash from the unified diff. The number of context lines n is 1. The chunk type t is '+'. The commit hash c is 'commit_hash.' The changed file f is 'src/qt/rpcconsole.cpp.' The left square corresponds to the first step in our algorithm. (1) and (2) are corresponding to the second step; we remove unrelated code in (1), and convert the string into a sequence of words by `lscp` in (2). (3) is corresponding to the step three; we compute the context features.

The intuition behind counting words or keywords:

Our definition of context features involves counting words or keywords in the context of a change. We consider that a context with more words is likely to be more complex than a context that has less words. Hence, we consider that counting the number of words in the context of a change is a proxy of the complexity of such change.

The main intuition behind using the number of keywords is that the number of keywords in the context might indicate how deeply nested change is. Therefore, a change with a larger number of keywords is likely to more complex that a change that has fewer (or no) keywords around it.

Finally, counting number of words/keywords is easy to compute in practice.

4.4 Case study design

In this section, we discuss our selection criteria for the studied indentation features, data, validation technique, preprocessing, projects, resampling approach, evaluation measures, and prediction models.

4.4.1 Indentation features

We compare context features with indentation features. We study two indentations features: *Added Spaces (AS)*, defined by Hindle et.al [59]; AS is the sum of the number of white spaces on all the '+' lines in a commit.

We additionally define a new indentation feature *Added Braces (AB)*. We consider the number of braces as a logical indentation because the number of braces in C++ and Java expresses how embedded one block of code is inside others. We first count the number of left-braces B_{left} and right-braces B_{right} from the head of a function to each '+' line, respectively. Second, we compute the difference B_{diff} between B_{left} and B_{right} on each '+' line. Finally, we sum B_{diff} for all '+' lines in a commit.

The intuition of using the indentation features as way to predict defects:

The indentation features have been used as a proxy to measure complexity of source code [59]. However, they have not been used in defect prediction. The rationale behind their use in defect prediction is that modifications in more indented code are likely to be more complex that modifications that happen in less indented code because the person doing the changes not only has to be concerned with what the code does, but also with the code that surrounds it. The code with the larger indentation is likely to be inside more control blocks–e.g., while, for, and if statements–than the code with the less indentation; we hypothesize that more control blocks might create more brittle code. Hence, all things equal, we

Table 4.4: Change features.

Dim.	Name	Definition
Diffusion	NS	Number of modified subsystems
	ND	Number of modified directories
	NF	Number of modified files
	Entropy	Distribution of modified code across each file
Size	LA	Lines of code added
	LD	Lines of code deleted
	LT	Lines of code in a file before the change
Purpose	FIX	Whether or not the change is a defect fix
History	NDEV	The number of developers that changed the modified files
	AGE	The average time interval between the last and the current change
	NUC	The number of unique changes to the modified files
Experience	EXP	Developer experience
	REXP	Recent developer experience
	SEXP	Developer experience on a subsystem

expect that changes to code that has more indentation might result in more defects that changes to code that has less indentation.

4.4.2 Preparing data using Commit Guru

The availability and openness of experimental data is a real challenge to evaluate defect prediction approaches. Therefore, we use data provided by Commit Guru, which Rosen et al. [141] provide publicly. Commit Guru is a web application, which identifies and predicts defective commits for Git repositories and calculates the change features (Table 4.4) that are often used for just-in-time defect prediction [71].

In this chapter, we use Commit Guru to calculate the change features [71]. We use the change features in RQ2 to compare with the context features in order to study what is the impact of the context features on defect prediction. Then, we use the change features, and their subsets (each of the change features) as studied features.

We refer to each feature in the change features as a subset of the change features. When using a subset of the change features, we pick up a feature from the change features, and use that feature for defect prediction. This is because each of the change features is also a churn feature. However, several features do not strongly relate to code churn. For example, Purpose feature (i.e., FIX, described in Table 4.4) is not affected by code churn. Hence, we remove three types of features from all the change features when considering their subsets that are Purpose feature (i.e., FIX), History features (i.e., NDEV, AGE, and NUC), and Experience features (i.e., EXP, REXP and SEXP). Hence we use each of NS, ND, NF, Entropy, LA, LD, and LT as a subset of the change features. We apply z-score to each of the subsets to normalized to a mean of 0 and a variance of 1.

When using the change features, to avoid using several strongly correlated features in the prediction, we apply the following preprocessing proposed and described in [71]:

- Exclude ND and REXP since they are strongly correlated with NF and EXP.
- LA and LD are divided by LT to normalize LA and LD.
- LT and NUC are divided by NF to normalize LT and NUC.

Finally, we apply z-score [181] to the changed features to normalized to a mean of 0 and a variance of 1.

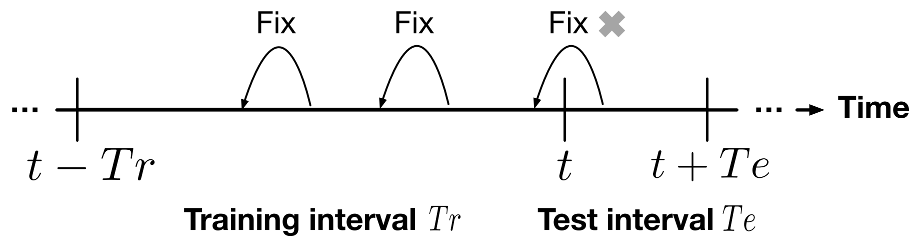


Figure 4.4: An example of the time sensitive change classification. The cross in gray indicates the information of fixing a commit is not used in the training interval.

4.4.3 Time sensitive change classification

Because we could use future commits to predict past commits, using 10-fold cross validation has a risk to make the artificially good results such as high precision and recall while studying just-in-time defect prediction [155]. In addition, while using 10-fold cross validation, we label the commits in training data as defective or not using all the commits information. However, this procedure also risks to use future information for prediction. To address these two issues and validate our experiments, we use *time sensitive change classification* [155].

Time sensitive change classification uses only past commits to label past commits and build prediction models for future commits. Figure 4.4 shows an example of the time sensitive change classification that uses the *training interval* between $t - Tr$ and t as training data and the *test interval* between t and $t + Te$ as test data. In this example, we use the commits in the training data to label its commits and build prediction models for predicting commits in the test data.

However, Tan et al. [155] reported three challenges. First, because defective commits are typically detected and fixed in 100–300 days [75], many undetected defective commits in the training interval would be labeled clean. Second, this validation is sensitive to the interval. For example, if the training interval is before the release day, features in the test interval would be different with the training interval. Third, if we take a long time gap between the training interval and

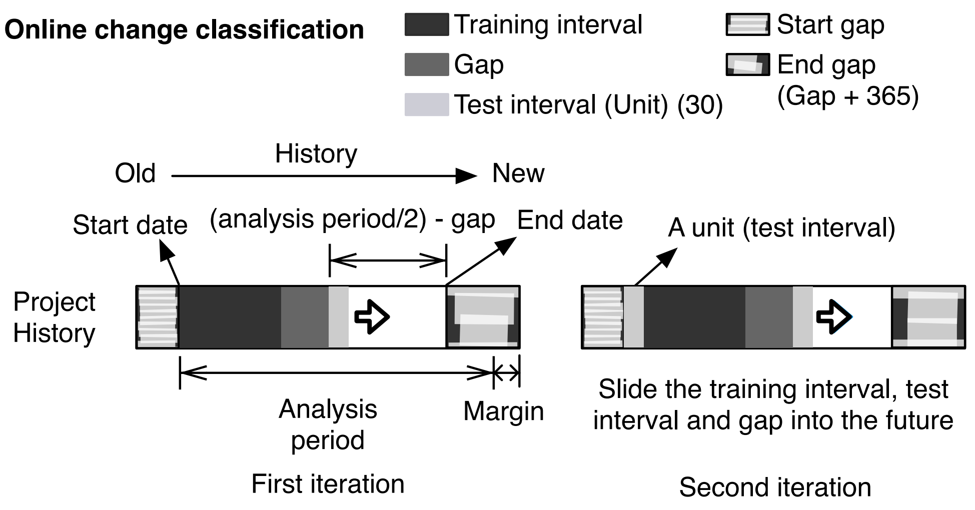


Figure 4.5: An overview of the online change classification. We show two iterations as an example. The part of the rectangle in black is the training data (training interval) labeled using the commits in the training interval and gap (in dark gray). The part of the rectangle in light gray is the test data (test interval) labeled using all of the commits in the project history including the end gap. Details of the terms in this figure are described in Section 4.4.4.

the test interval, features such as developers and programming styles might have changed between the training interval and the test interval. To address these three challenges, Tan et al. [155] recommended to use *online change classification*.

4.4.4 Online change classification

Online change classification is a validation technique. We describe the online change classification, and how this validation technique addresses these three challenges. To address the first challenge, a *gap* is used between the *training interval* and the *test interval* (Figure 4.5). The *gap* is used only during the labeling of the commits in the *training interval*. This additional interval allows more time to detect defective commits in the *training interval* and make labeling result more precise. Typically, the *gap* is the average or medium time between a defect inducing commit and a defect fixing commit; in our experiments, we use median

Table 4.5: Parameter values of the online change classification for each project (days).

Project	Start gap	End gap	Gap	Unit (test interval)	Training interval	Iteration step size
Hadoop	925	526	151	30	510	17
Camel	743	416	40	30	1,110	37
Gerrit	375	523	137	30	900	30
Osmand	1,011	413	17	30	420	14
Bitcoin	789	459	77	30	600	20
Gimp	2,004	687	281	30	2,100	70

time for each project from our pre-experiment (Table 4.5).

To address the second and third challenges, the *time sensitive change classification* is executed multiple times while updating the *training interval*, *test interval* and *gap*. The multiple execution minimizes the bias from a certain *test interval*. The *training interval*, *test interval* and *gap* slide into the future by a certain interval (Figure 4.5). This certain interval is called *unit*. A *unit* is 30 days (one month) in our experiments. The *test interval* is 30 days as well. Note that the unit and the test interval are parameters, hence; different parameter values might have the impact to the result of our experiments. We studied this point in Section 4.8. The result shows that these parameters have little impact for the results of our experiments.

We also use *start gap* and *end gap* [155] that are intervals that we do not use as *training interval* and *test interval*. The beginning of a software project history may be inconsistent and unstable. The end of a software project history would be labeled clean because defective commits would not be detected. Hence, the *start gap* and *end gap* would support building better prediction models and improving the quality of the analysis.

Table 4.5 shows the actual parameters for each project. We manually look at the number of commits and decide on the *start date* at a point after the number of committed commits increases and decreases moderately (reach a peak). The *start gap* is the interval between the first commit date and the *start date*. The reason why we use this process is that after the number of committed commits increases and decreases moderately, the project would have been released and would be in a stable state.

To decide the *end gap*, we need to compute the *analysis period*, *iteration step size* and *training interval*. In the following, *analysis period* is the maximum studied days. We define the *analysis period*, *iteration step size* and the *training interval* as follows:

$$\textit{analysis period} = (\textit{CommDate}_{\textit{latest}} - \textit{start date}) - \textit{margin},$$

$$\textit{iteration step size} = (\textit{analysis period}/2 - \textit{gap})/\textit{unit},$$

$$\textit{Tr} = \textit{iteration step size} \cdot \textit{unit},$$

where (and hereafter)

- $\textit{CommDate}_{\textit{latest}}$ is the latest commit date,
- *margin* is a margin to remove defective commits that may not be detected yet, and
- *Tr* is the training interval.

We first compute the interval between the *start date* and the date that is *margin* days before the latest commit date. This process removes the defective commits that are not detected. We use 365 as the *margin* to compute the *end gap*. Hence, the *end gap* is always 365 and over. Because we use *unit* as a *test interval* as well, *iteration step size* shows that the rest of iterations that we can slide the *training interval*, *test interval* and *gap* into the future as avoiding to use the commits that are committed in the latest *margin* days. In addition, we use *gap* to compute *iteration step size*. This additional *gap* avoids the commits that are in the latest *margin* days plus the *gap* days and ensure that we consider enough commits to label the commits in the *test interval*. The *training interval* is decided by *iteration step size* and *unit*. Finally, we define the *end date* and the *end gap* as follows:

$$end\ date = start\ date + (Tr + gap + (iteration\ step\ size \cdot unit)),$$

$$end\ gap = CommDate_{latest} - end\ date.$$

For labeling commits either defective or clean, we follow the labeling process used by Commit Guru:

1. Collect commits c_{fix} whose messages contain specific keywords (as described by Rosen et al. [141]), such as "bug" or "fix". Identify the modified lines l in the commits c_{fix}.

2. Find out previous commits c_{bad} on which the lines l were added or modified previously to the corresponding change in c_{fix}. Label each commit c_{bad} as defective.

We conduct this procedure using the *training interval* and the *gap* for labeling training data, and using all of the commits for labeling test data.

4.4.5 Preprocessing by *z*-score

z-score is a popular normalization approach in defect prediction [181]. *z*-score normalizes the input data to mean 0 and variance 1. The equation of *z*-score is:

$$\mathbf{X}_{z-score} = \frac{\mathbf{X}_{org} - \mu}{\sigma} \tag{4.1}$$

where μ is the mean of the values of a feature for commits. σ is the variance of the values of a feature for commits. $\mathbf{X}_{org}$ is a vector of all values (all commits) of a feature. $\mathbf{X}_{z-score}$ is a vector of all values (all commits) of a normalized feature.

4.4.6 Studied projects

For our experiments, we use six open source projects: Hadoop, Camel, Gerrit, Osmand, Bitcoin and Gimp. Table 4.6 shows details of the projects. The studied

Table 4.6: Details of the studied projects. Defective rate refer to the commits labeled using all commits.

Project	Language	Total Number of Commits	Defective Rate
Hadoop	Java	13,920	24.8 %
Camel	Java	24,740	23.2 %
Gerrit	Java	18,794	20.1 %
Osmand	Java	31,366	14.0 %
Bitcoin	C++	11,093	14.4 %
Gimp	C++	37,149	22.5 %

projects include software for various fields, such as a server or an application, and are written in two popular programming languages (C++ and Java). We calculate the context features and the indentation features for each commit of these projects. For more precise analysis, we study all the commits that have changed at least one line in the source code.

4.4.7 Resampling approach

While learning the defect prediction model, the learning performance is affected by imbalanced data [155]. In our case, Table 4.6 shows that "clean" commits outnumber "defective" commits. Hence, if we use this data directly as training data, the learning performance could decrease. General resampling approaches remedy this problem, as shown by prior studies [71, 155, 176].

For our experiment, we use random under-sampling. Random under-sampling reduces the majority class at random to make the size of the majority class equal to the size of the minority class. Because we must evaluate our approach on real data, we apply resampling only to training data, not to test data.

4.4.8 Evaluation measures

To measure the impact of the context features for defect prediction, we use three evaluation measures: the area under the receiver operation characteristic curve (AUC), the Matthews correlation coefficient (MCC), and Brier score (Brier)[¶]. Precision and Recall are frequently used in defect prediction as evaluation measures. However, several researchers warned that these measures show biased results [16, 23, 157].

AUC and Brier score are threshold-independent measures. Tantithamthavorn et al. [157] suggested to use threshold-independent measures to address pitfalls in defect prediction research. Although MCC is a threshold-dependent measure, MCC is not affected by the skewness of defect data [15, 181] and we want to better understand the predicting power of the features [70]. Therefore, we also use MCC in this chapter. The threshold of MCC is 0.5.

We use the Scott-Knott ESD test [160] (using 95% significance level) that was used in Chapter 3 to compare the context features and the traditional code churn features. We also apply the Scott-Knott ESD test to the ranks that are computed by the Scott-Knott ESD test.

The reason why we apply the Scott-Knott ESD test twice is to avoid the variances of the values of the evaluation measures across the studied projects. If there exist the variances across the studied projects, it would be difficult to compare the studied features over all the studied projects instead of each studied project. This idea was proposed by Ghotra et al. [40]. They applied Scott-Knott test twice to ensure that they recognized techniques that perform well across the studied projects. They showed the following example: if a prediction model has an AUC of 0.9 on project A, and 0.5 on project B, we would get worse result while using Scott-Knott test once for all projects. However, if an AUC of 0.5 is the best

[¶]Note that while higher values of AUC and MCC are better than lower values, lower values of Brier score are better than higher values. This is because Brier score is the sum of the mean squared differences between predicted probabilities and actual binary labels.

AUC value in the project B, and 0.9 is also the best value in the project A, then this classification technique should be the best-performing technique. The first Scott-Knott test computes the rank within a project. And the second Scott-Knott test computes the rank across the projects without the variance of the values of the evaluation measures due to using the rank. We use the Scott-Knott ESD test instead of the Scott-Knott test in order to consider the effect size. We call this procedure as *double Scott-Knott ESD test*.

The results of the Scott-Knott ESD test and the double Scott-Knott ESD test are a rank (number) for each feature. The smallest rank, 1, indicates the best rank. The largest rank indicates the worst rank. A rank can contains multiple features at once. We interpret features which have many smallest/smaller ranks as the best features since it indicates that the features significantly outperform many others. Hence, for the Scott-Knott ESD test, we used the top-3 ranks to evaluate the features across the studied projects. We report features which have the most top-3 ranks across the studied projects as the best features in the Scott-Knott ESD test.

For the double Scott-Knott ESD test, we used boxplots to show the ranks of the studied features for each evaluation measure. Each boxplot contains six ranks by the Scott-Knott ESD test for all the studied projects. The double Scott-Knott ESD test classifies these boxplots by the Scott-Knott ESD test. This analysis avoids the variances of the actual performance differences across the studied projects due to using the rank. Our interpretation is that features which have the smallest rank as the best features since it indicates that the features significantly outperform many others.

4.4.9 Prediction models

We use two defect prediction models, logistic regression model (LR) [99] and random forest model (RF) [61] that were used in Chapter 3 as well.

Prior work [49,159] showed that the parameter optimization of the prediction models crucially affects the prediction performance. For example, Tantithamthavorn et al. [159] showed that a simple automated parameter optimization can dramatically improve the AUC performance of defect prediction models (the best case is about 40 percentage points of AUC). Hence, considering the parameter optimization is also an important aspect in our experiment.

For LR, we consider a parameter: *C*.

- *C:* C is a parameter which indicates the regularization strength. For example, if we have many features but not much data, LR would optimize its parameter for the training data excessively. Hence, LR provides worse performance for the test data. To address this challenge, the regularization strength C is used when optimizing the parameter. We study the C of 0.00001, 0.0001, 0.001, 0.01, 0.1, 1, 10, and 100 when using the change features and COMB. For the other features, we do not use the C since the number of features is 1 at a prediction model.

In addition, we need to consider the correlation between the studied features. If the studied features are correlated, LR would get *multicollinearity problem* [37]. When using the change features, we need to consider the correlation. To avoid the correlated features, prior work [71] proposed a preprocessing. We follow the same preprocessing of prior work [71] that was described in Section 4.4.2. COMB has two features. However, they are not correlated (see Table 4.21). Hence, we do not need to deal with the correlation in COMB.

For RF, we keep using the normalized change features for LR. In addition, we consider two parameters: *mtry* and *number of trees* that are specific parameters in RF.

- *mtry:* mtry is a parameter which indicates the number of features randomly selected for each node in a tree. For example, if we set mtry=2, RF selects 2 features from the studied features to generate a node in a tree for splitting the studied commits. We study the mtry of 1, 2, 5, 10, and 12 when using the normalized change features, 1 and 2 when using COMB, and 1 when using other features since the number of normalized change features is 12, the number of features in COMB is 2, and the number of other features is 1 at a prediction model.

- *number of trees:* Number of trees is a parameter which indicates the number of trees which RF generates. RF merges all the outputs of the trees for computing the final result. We study the number of trees of 2, 5, 10, 50, 100, 500, 1,000.

We optimize these parameters for each iteration. We split the training data to 80% of the training data and 20% of validation data. We use the training data to train the model based on a parameter setting, and evaluate that parameter setting on the validation data. We use the best parameter setting on the test data.

4.5 Research questions and methodology

4.5.1 Research questions

Our proposed context features have three parameters: commit c, context size n and chunk type t. Hence, we first study which configurations of these parameters are the best for predicting defective commits. Because c is a parameter that cannot be optimized, we study n and t to design the best context features. To do this, we formulate the following research question: (RQ1) *What is the impact of the different variants of context features on defect prediction?*

RQ1 does not confirm what is the impact of the context features for defect prediction compared to the traditional code churn features. Hence, we also study the prediction performance of the context features compared to the traditional code churn features that are the change features, their subsets and the indentation features in order to confirm whether the context features are effective or not. We additionally study the performance of extended context features, which are combinations between the context features and the traditional code churn features for defect prediction in order to improve the predicting power of the context features. The extended context features count (1) number of words and (2) number of keywords in the context lines and the changed lines. To do this, we formulate the following research question: (RQ2) *Do context lines improve the performance of defect prediction?*

RQ2 compares the prediction performance across the context features, the extended context features, and the traditional code churn features. However, we do not study combination features between the context features; we use a context feature alone on a prediction model in RQ1 and RQ2. Hence, in this RQ, we study the impact of combination features that use two extended context features that count (1) number of words and (2) number of a certain keyword (e.g., "goto") at a prediction model. To do this, we formulate the following research question: (RQ3) *What is the impact of combination features of context features on defect prediction?*

4.5.2 Methodology

We explain our experimental methodology.

RQ1. What is the impact of the different variants of context features on defect prediction?

We conduct two experiments in order to study the impact of chunk types and context sizes for just-in-time defect prediction. We first study the impact of chunk types. Second, we study the impact of context size based on a fixed chunk type. In each experiment, we build the studied defect prediction models and predict defective commits in the studied project histories.

We consider two supervised learning models as defect prediction models that are LR and RF. Prior research showed inconsistent results that prediction models provide significant difference [40] and no significant difference [90, 102, 143]. The main point in this chapter is to evaluate the impact of the context features for defect prediction, not the impact of the prediction models. Hence, we use only two models and do not consider the difference between the prediction models.

We split the set of commits into training data and test data using the online change classification [155]. 10-fold cross validation is a frequently used validation technique in defect prediction, however; cross validation has risks such as making artificially good results due to mixing past and future commits. The online change classification addresses the challenges of the cross validation and improves the quality of the analysis in just-in-time defect prediction [155]. We described details in Section 4.4.4.

We compute the context features for each chunk type for each commit. We apply preprocessing to the context features in the training and test data. We use z-score; the mean and the variance of z-score are decided from the training data. We use the context feature as an input of the studied models. The models are trained using training data, and compute prediction results using test data. When training the model, we optimize the parameters of the prediction models. We described details in Section 4.4.9.

Finally, we evaluate the results using three evaluation measures: AUC, MCC, and Brier score. Each measure has multiple values that come from the number of the iteration step sizes of the online change classification. We show the number of iteration step sizes in Table 4.5. For example, it is 17 that is the number of iteration step size of the Hadoop project. Hence, we get 17 values for each of three evaluation measures. For each measure, we summarize the multiple values with its median value. We conduct the above procedure for each studied project. Therefore, each context feature has 12 median values in the online change classification (for six projects times two prediction models).

We conduct this procedure for each chunk type. Then, we compare the context features of the different chunk types w. r. t. the three evaluation measures. We apply the Scott-Knott ESD test [160] to the context features for each evaluation measure for each project. Each context feature has two values (results by LR and RF models) for each project. Then, we evaluate statistically significant differences and effect sizes between the context features for each evaluation measure for each project. The result is shown as a rank. For example, if a certain context feature A has the best value on a certain evaluation measure, this context feature A achieves the rank 1. If another context feature B has no significant difference to the context feature A that achieves the rank 1, this context feature B also achieves the rank 1. If another context feature C has significant difference to the context feature A and B, this context feature C achieves rank 2.

Although we would get the rank from the first Scott-Knott ESD execution, the rank is computed for each project. Hence, we would get different ranks for each project on a context feature. To avoid the variances of the ranks across the studied projects, we additionally apply the Scott-Knott ESD test to the ranks instead of the actual values of the evaluation measures, the double Scott-Knott ESD test. Each context feature has six ranks (results by all the studied projects) for each evaluation measure. The additional Scott-Knott ESD test compares the studied context features in terms of the rank. Then, we evaluate statistically significant differences and effect sizes between the context features for each evaluation measure.

We conduct the same procedures on different context sizes instead of different chunk types before we apply the Scott-Knott ESD test. In this comparison, we then compare the values of evaluation measures for each iteration step between different context sizes. We count the iteration steps for each context size that provide the best prediction performance value. We make histograms of the number of iteration steps that provide the best prediction performance for each context size for each evaluation measure and context feature. From these histograms, we conclude the impact of different context sizes for the performance of defect prediction. For example, let us suppose we conducted an experiment with that iteration steps are 100, context sizes are 1, 2, and 3; the context size 1 has 50 iteration steps where the context size 1 has the best performance, the context size 2 has 20 iteration steps where the context size 2 has the best performance, and the context size 3 has 30 iteration steps where the context size 3 has the best performance. In

Table 4.7: The extended context features.

Features	Description
$NCCW(c, n, t)$	Extend $NCW(c, n, t)$ using not only context lines but also changed lines.
$NCCKW(c, n, t)$	Extend $NCKW(c, n, t)$ using not only context lines but also changed lines.

this example, we would get histograms in which the context size 1 has 50, 2 has 20, and 3 has 30; hence, we would conclude that the context size 1 is the best.

From the results, we investigate the impact of the context features variants (different chunk types and context sizes). The goal of this RQ1 is to find the best context features variant for just-in-time defect prediction. The best context features variant is considered as the context features in RQ2.

RQ2. Do context lines improve the performance of defect prediction?

To answer this RQ, we compare the best variant of the context features NCW and NCKW (as determined in RQ1) with the change features and their subsets (both described in Section 4.4.2), the indentation features (described in Section 4.4.1) and the extended context features. We build the defect prediction models to evaluate the features. The prediction procedure is similar to the procedure for RQ1; however, the preprocessing has differences (the details are described later in this section).

In order to improve the performance of defect prediction, we define two new features based on NCW and NCKW that measure both the context and the changed lines called *extended context features*. These features are *NCCW* (number of words in the context and the changed lines) and *NCCKW* (number of keywords in the context and the changed lines) in Table 4.7. NCCW and NCCKW use only added-lines as the changed lines. This is because it is known that a change feature, "added-lines", is one of the best indicator of change risk [145, 147]. These features will show the results of the combination between the context features and the traditional code churn features. From the results of RQ1, we choose the appropriate chunk type from '+', '-' and 'all', and the context size from one to ten for NCCW and NCCKW.

We apply the preprocessing to the change features and their subsets that was described in Section 4.4.9. For the context features, we apply z-score to normalize to a mean of 0 and a variance of 1 since the subsets of the change features are also normalized by z-score.

RQ3. What is the impact of combination features of context features on defect prediction?

To answer this RQ, we use our new combination features that use both NCCW and NCCKW. This is because, according to the results of RQ2, NCCW and NCCKW have better prediction performance than NCW and NCKW alone. NCCW and NCCKW are strongly correlated with each other (see Section 4.7.3). Hence, we need to remove the correlation in order to address the multicollinearity problem [37] for using them on a prediction model.

We, hence, modify NCCKW into counting only each specific keyword instead of counting all keywords (Table 4.2, # of keywords: 20). Hence, we get 20 variants of NCCKW. For example, a variant of NCCKW measures the number of "goto" statements (in both the context and the changed lines). We call each of these features as a modified NCCKW. There are 20 modified NCCKW. This modification removes the strong correlation between NCCW and NCCKW. NCCW and each modified NCCKW are rarely correlated.

We use NCCW and each of the modified NCCKW on a prediction model as two explanation variables, and study the performance of each of the modified NCCKW. From this result, we conclude the best combination features for NCCW and a modified NCCKW. We call the combination features as *COMB*. We compare COMB with the other features following the same procedures of the procedure for RQ2.

4.6 Case study results

4.6.1 RQ1. What is the impact of the different variants of context features on defect prediction?

For the context features, the best chunk type is '+'.

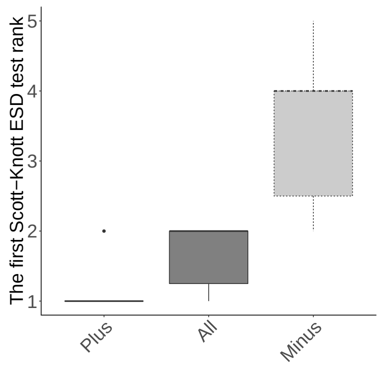
Table 4.8 shows the ranks of the Scott-Knott ESD test results for each evaluation measure for each context feature variant. Each cell shows the rank of a context feature variant in an evaluation measure and a project. Note that we compared variants with different chunk types with the same context size ($n = 3$, the default context size of the diff command `git show`). The rank is computed across context feature variants for each project and evaluation measure. For example, the gray cells in Table 4.8 are a set where the Scott-Knott ESD test is conducted. We summarize the number of projects that are the top three ranks for each context feature variant (row) in columns of *#R1*, *#R2*, and *#R3*. Hence, the sum of numbers between *#R1* to *#R3* in a row is 6 or less. The column of *Sum* is the sum of *#R1*, *#R2*, and *#R3*. Due to space limitation, we shorten the project names in the table: Bitcoin is B., Camel is C., Gerrit is Ge., Gimp is Gi., Hadoop is H., and Osmand is O.

Regarding AUC, using only the '+' chunk on NCW yields the best results and statistically outperforms the other features except the Osmand project, i.e., the rank is one in 5 of 6 projects. Regarding MCC, we find that the rank is one in 3 of 6 projects, and the rank is one, two or three in all projects when using '+' chunk on NCW or NCKW. Regarding Brier score, using the '+' or 'all' chunk on NCKW yields the best results and statistically outperforms the other features for 3 of 6 studied projects.

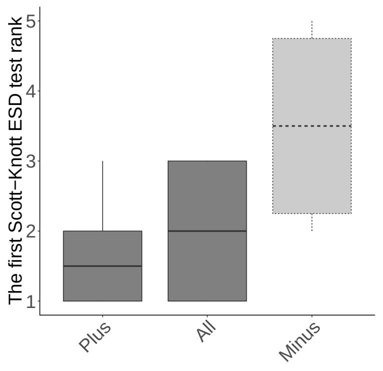
Figure 4.6 shows the results of the double Scott-Knott ESD test on the results for each context feature in all projects; each boxplot contains six ranks of the first Scott-Knott ESD test execution for the studied projects on a chunk type. The x-axis indicates a chunk type; Plus, Minus, and All correspond to '+', '-', and 'all';

Table 4.8: The ranks of the Scott-Knott ESD test results for each context feature variant and studied project on three evaluation measures. Please see text for a full explanation. The actual values of each evaluation measure by RF and LR models are shown in Appendix (Table 4.15, 4.16, and 4.17).

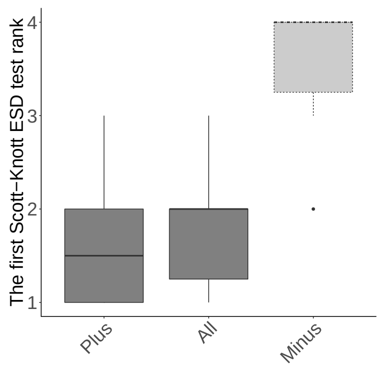
Evaluation Measures	**Features**	**Chunk Types**	**Projects**						**Numbers of Ranks**			
			B.	**C.**	**Ge.**	**Gi.**	**H.**	**O.**	*#R1*	*#R2*	*#R3*	*Sum*
AUC	NCW	+	1	1	1	1	1	2	5	1	0	6
		-	2	5	4	4	2	4	0	2	0	2
		all	1	2	2	2	1	2	2	4	0	6
	NCKW	+	1	2	2	3	1	1	3	2	1	6
		-	2	4	5	5	2	3	0	2	1	3
		all	1	3	3	3	1	1	3	0	3	6
MCC	NCW	+	2	1	2	1	3	1	3	2	1	6
		-	3	4	4	4	4	2	0	1	1	2
		all	2	1	2	2	3	1	2	3	1	6
	NCKW	+	1	2	2	3	1	1	3	2	1	6
		-	2	3	3	4	3	1	1	1	3	5
		all	2	2	1	2	2	1	2	4	0	6
Brier	NCW	+	3	3	1	1	2	2	2	2	2	6
		-	3	3	2	1	2	2	1	3	2	6
		all	3	4	1	1	2	2	2	2	1	5
	NCKW	+	1	2	1	3	2	1	3	2	1	6
		-	2	1	2	2	1	2	2	4	0	6
		all	2	2	1	2	1	1	3	3	0	6



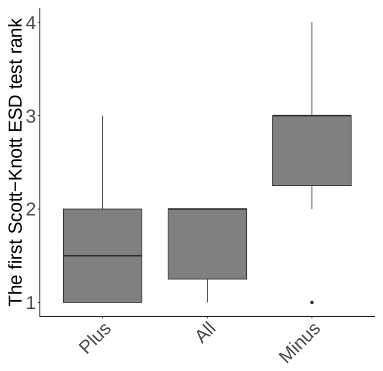
(a) NCW AUC



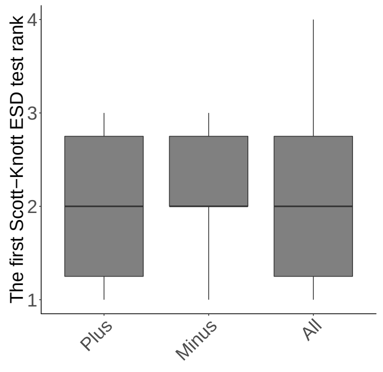
(b) NCKW AUC



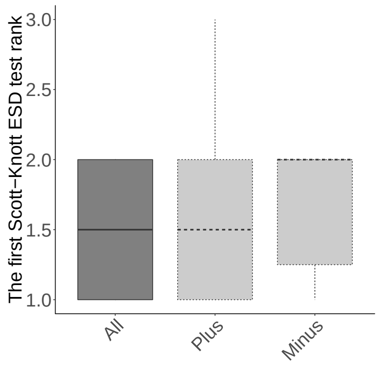
(c) NCW MCC



(d) NCKW MCC

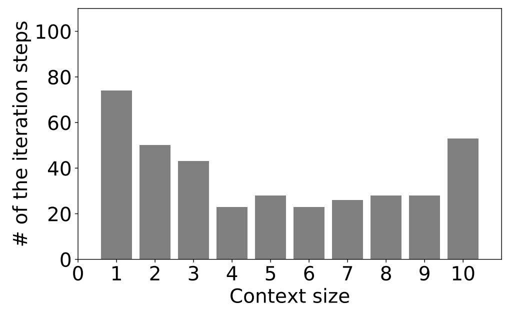


(e) NCW Brier

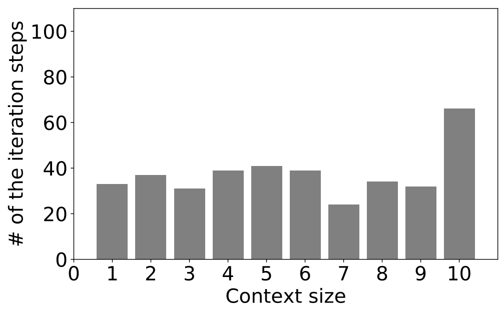


(f) NCKW Brier

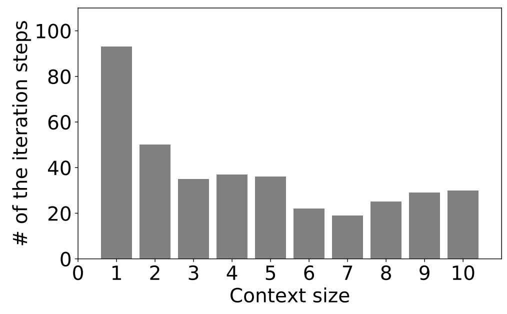
Figure 4.6: The results of the double Scott-Knott test on the results for each context feature in all projects. Please see text for a full explanation.



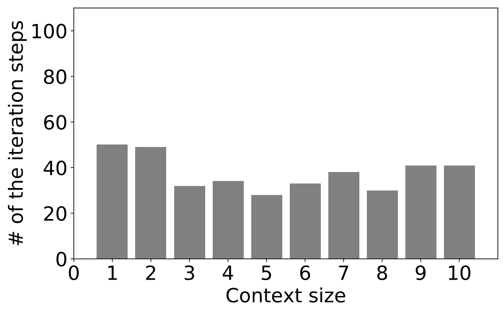
(a) NCW AUC



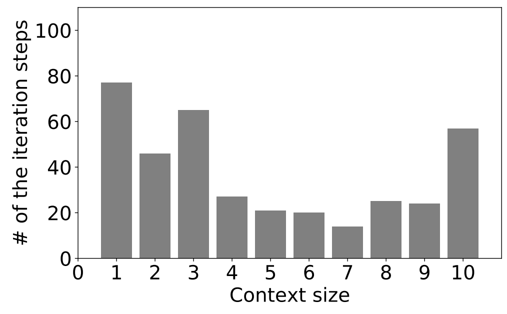
(b) NCKW AUC



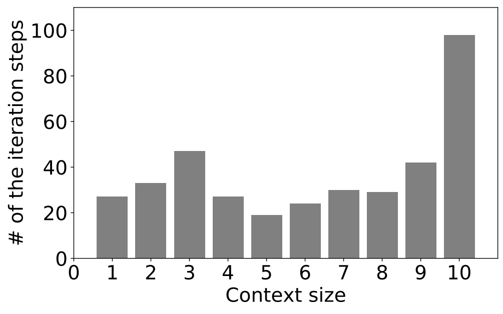
(c) NCW MCC



(d) NCKW MCC

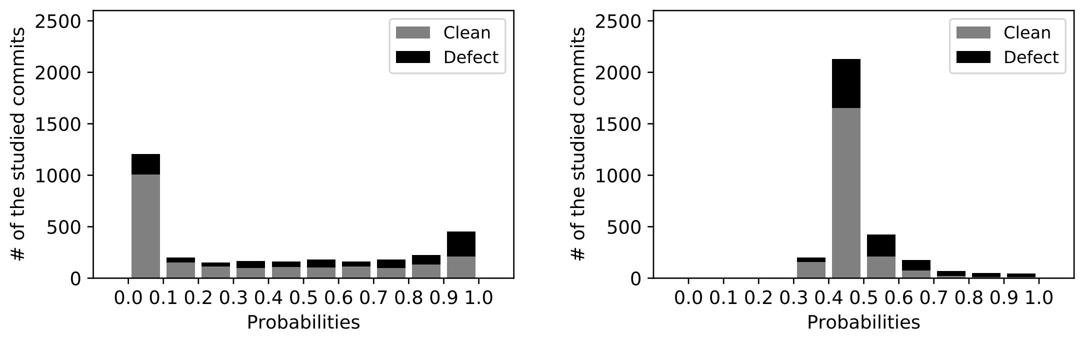


(e) NCW Brier



(f) NCKW Brier

Figure 4.7: The numbers of iteration steps that provide the best prediction performance for each context size. We use all iteration steps of all studied projects on two prediction models (LR and RF). The sum of all iteration steps is 188 (17 + 37 + 30 + 14 + 20 + 70 from Table 4.5). Hence, the sum of all values is 376 (188 iteration steps * 2 models). For example, the sum of the y-axis values in Figure 4.7(a) between 1 to 10 is 376.



(a) NCKW RF Hadoop (context size=10) (b) NCKW LR Hadoop (context size=10)

Figure 4.8: The numbers of studied commits in Hadoop project when the context size is 10. The x-axis refers to the predicted probabilities using NCKW that were computed by either RF (left) or LR (right) models.

the y-axis indicates the rank for each studied project in the first Scott-Knott ESD test execution. We use two gray colors (dark gray and light gray) and two lines (solid line and dashed line) indicate a rank according to the double Scott-Knott ESD test. The different rank indicates a statistical significant difference with small effect size or over. We observe that '+' achieves the best median rank for all the evaluation measures and the context features.

With one exception, '+' consistently performed better than other types of chuck types. This exception is shown in Figure 4.6(f) shows that 'all' chunk statistically outperforms '+' chunk on NCKW on Brier score; however, the median, and 25 and 75 percentiles are same. Hence, we choose '+' chunk as the best chunk type for our context features.

A context size of 1, provides better prediction performance for NCW, while a context size of 10, provides better prediction performance for NCKW.

Figure 4.7 shows the numbers of iteration steps that provide the best prediction performance on different context sizes. The left column of Figure 4.7 (Figure 4.7(a), 4.7(c) and 4.7(e)) shows the results for NCW with chunk type '+'. The right column of Figure 4.7 (Figure 4.7(b), 4.7(d) and 4.7(f)) shows the results for NCKW with chunk type '+'.

We can observe opposite results between the NCW and NCKW. On the NCW, the context size of 1 has the highest histogram. This result indicates that the context size of 1 provides the most best prediction performance in all the iteration steps comparing to other context sizes. However, on the NCKW, the context size of 10 has the highest histogram on AUC and Brier score. The context size of 1 in MCC is slightly higher than the other context sizes. This result implies that the threshold, 0.5, is not suitable for NCKW. Figure 4.8 shows the numbers of studied commits with predicted probabilities that were computed by the prediction models in Hadoop project when the context size is 10. The numbers of commits in Figure 4.8(b) are gathered more closely around 0.5 and many defective commits (orange) are lower than 0.5 (by LR), however, the numbers of commits in Figure 4.8(a) are not gathered around 0.5 (by RF). Because the threshold 0.5 provides many defective commits that are identified as clean in Figure 4.8(b), this distribution affects the results on MCC when using NCKW. Hence, the results are best when the context size is 10 in AUC and Brier score, however; the result is not best when the context size is 10 in MCC. We can observe the same tendency on different studied projects.

From these results, as the appropriate context size, we use 1 for NCW, and 10 for NCKW. Hereafter, we refer to $NCW(c, 1, +)$ and $NCKW(c, 10, +)$ as NCW and NCKW, respectively. In addition, we refer to $NCCW(c, 1, +)$ and $NCCKW(c, 10, +)$ as NCCW and NCCKW, respectively.

4.6.2 RQ2. Do context lines improve the performance of defect prediction?

The extended context feature NCCW, the indentation features, and lines added (LA) provide many top three rank performance on just-in-time defect prediction.

Table 4.9 shows the ranks according to the Scott-Knott ESD test results of the

Table 4.9: The ranks of the Scott-Knott ESD test results for studied features. *#R1* (*#R2*, or *#R3*) is the sum of the numbers of cases where the rank is one (two, or three); *Sum* = *#R1* + *#R2* + *#R3*. The actual values that were computed by RF and LR are shown in Appendix.

(a) AUC

Feature Types	Features	Projects						Numbers of Ranks			
		B.	**C.**	**Ge.**	**Gi.**	**H.**	**O.**	*#R1*	*#R2*	*#R3*	*Sum*
Context	NCW(c,1,+)	6	8	5	4	4	2	0	1	0	1
	NCKW(c,10,+)	7	8	7	4	3	2	0	1	1	2
	NCCW(c,1,+)	1	3	3	1	2	2	2	2	2	6
	NCCKW(c,10,+)	4	7	4	2	1	1	2	1	0	3
Indentation	AS	1	1	1	2	2	2	3	3	0	6
	AB	3	2	2	3	2	2	0	4	2	6
Traditional	Changes	5	3	6	7	1	4	1	0	1	2
	NS	12	10	10	9	7	7	0	0	0	0
	ND	10	5	8	8	3	5	0	0	1	1
	NF	8	4	5	6	2	3	0	1	1	2
	Entropy	9	6	6	6	5	3	0	0	1	1
	LA	2	1	3	1	1	2	3	2	1	6
	LD	11	9	9	5	5	6	0	0	0	0
	LT	13	11	11	10	6	8	0	0	0	0

(b) MCC

Feature Types	**Features**	**Projects**						**Numbers of Ranks**			
		B.	**C.**	**Ge.**	**Gi.**	**H.**	**O.**	*#R1*	*#R2*	*#R3*	*Sum*
Context	NCW(c,1,+)	5	7	5	5	4	1	1	0	0	1
	NCKW(c,10,+)	4	7	6	3	4	2	0	1	1	2
	NCCW(c,1,+)	3	1	2	2	3	2	1	3	2	6
	NCCKW(c,10,+)	3	5	3	1	2	1	2	1	2	5
Indentation	AS	1	2	1	4	3	3	2	1	2	5
	AB	2	3	2	2	4	3	0	3	2	5
Traditional	Changes	5	5	5	8	1	7	1	0	0	1
	NS	10	8	8	10	7	8	0	0	0	0
	ND	8	4	5	9	4	4	0	0	0	0
	NF	6	2	4	4	2	3	0	2	1	3
	Entropy	7	6	5	7	5	5	0	0	0	0
	LA	3	1	3	1	3	2	2	1	3	6
	LD	8	8	7	6	5	6	0	0	0	0
	LT	9	9	9	11	6	9	0	0	0	0

(c) Brier Score

Feature Types	**Features**	**Projects**						**Numbers of Ranks**			
		B.	**C.**	**Ge.**	**Gi.**	**H.**	**O.**	*#R1*	*#R2*	*#R3*	*Sum*
Context	NCW(c,1,+)	5	5	5	9	2	3	0	1	1	2
	NCKW(c,10,+)	5	4	5	8	2	2	0	2	0	2
	NCCW(c,1,+)	3	3	3	6	3	2	0	1	4	5
	NCCKW(c,10,+)	3	3	4	6	2	2	0	2	2	4
Indentation	AS	2	2	2	9	3	2	0	4	1	5
	AB	3	2	3	11	3	2	0	2	3	5
Traditional	Changes	1	1	1	1	1	1	6	0	0	6
	NS	7	4	7	2	4	2	0	2	0	2
	ND	7	4	6	5	3	4	0	0	1	1
	NF	8	5	6	7	2	4	0	1	0	1
	Entropy	6	3	5	3	3	4	0	0	3	3
	LA	4	2	4	4	3	2	0	2	1	3
	LD	8	5	6	10	3	4	0	0	1	1
	LT	9	6	8	12	4	5	0	0	0	0

three evaluation measures for each studied feature. Each cell includes the rank. The rank is computed across the studied features for each project. For example, the gray cells in Table 4.9(a) is a set where the Scott-Knott ESD test is computed. The actual values of the three evaluation measures that are used in the Scott-Knott ESD test are shown in Appendix as Table 4.18, 4.19 and 4.20. We summarize the number of projects that are the top three ranks for each studied feature (row) in columns *#R1* to *#R3*, and the column *Sum* is the sum of *#R1*, *#R2*, and *#R3*. The maximum value of *Sum* is six that is the number of the studied projects. Note that "Changes" in the table (also in other tables and figures of this chapter) indicates the change features.

NCCW ($NCCW(c, 1, +)$) provides the top three rank prediction performance in all projects on AUC and MCC, and 5 of 6 projects on Brier score. NCCW does not provide the top one rank prediction performance on Brier score. However, this is not to be a challenge for just-in-time defect prediction. Brier score is the sum of the mean squared differences between predicted probabilities, i.e., the outputs computed by RF and LR models, and actual binary labels, i.e., clean or defect in the studied commits. From this point, this result implies that the probabilities that were computed by NCCW might be close to 0 or 1 (clean or defect) than other studied features. The probabilities that are closer to 0 or 1 indicate that the probabilities clearly indicate either clean or defect even if predicted results are incorrect. However, the results on AUC and MCC are good. Hence, even if incorrect results are far from correct results, NCCW still has strong predicting power because of its MCC results and NCCW might provide better performance at other thresholds on average because of its AUC results. This result indicates that the extended context feature NCCW has strong predicting power for just-in-time defect prediction in the studied churn features.

Added spaces (AS), added braces (AB) and lines of code added (LA) also provide many top three rank prediction performance on AUC and MCC. For AS and AB, all projects on AUC and 5 of 6 projects on MCC, for LA, all projects on

AUC and MCC. This result also shows that the indentation features and a churn feature LA have strong predicting power. All of the features do not provide the top one rank prediction performance on Brier score as well. From the same reason of the results of the extended context features, we conclude that AS, AB and LA have strong predicting power.

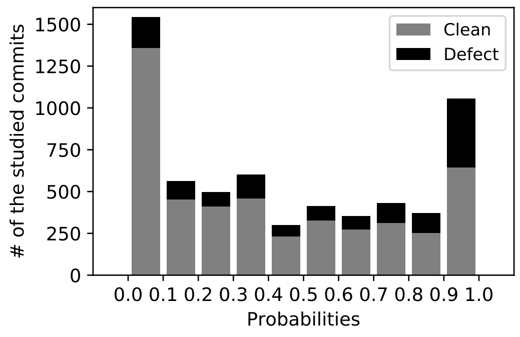
The change features that use all of the churn features provide that all projects are in the top three ranks on Brier score, while rarely providing the top three rank performance on AUC and MCC. This result implies that the probabilities that were computed by the change features might be close to 0.5 or the correct label than probabilities given by the other studied features. The probabilities that are close to 0.5 indicate that the probabilities are close to the correct label in incorrect results. Figure 4.9 shows the number of studied commits with predicted probabilities that were computed by the prediction models in the Camel project using NCCW and the change features. We can observe that when using the RF model, the probabilities that were computed by the change features are close to 0.5 than the NCCW.

When using the LR model, the probabilities that were computed by the NCCW is close to 0.5 than the change features. However, the mean squared differences (Brier score) of the results of the change features are smaller than NCCW in the half of the projects (Table 4.20 in Appendix). To show this result in a simpler manner, we define a difference between the probabilities and the actual labels in LR model. In the following, *Diff* is the difference on a feature in a project, C is a set of all of the studied commits c, *abs* is a function that computes absolute value, p_c is the predicted probabilities of a commit c and $label_c$ is the actual label of a commit c where defective commits are 1 and clean commits are 0. Based on these parameters, we define the *Diff* as follows:

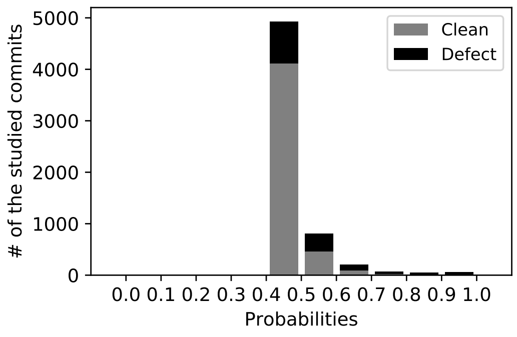
$$Diff = \sum_{C} \mathrm{abs}(p_c - label_c).$$

This is a simple variant of the Brier score.

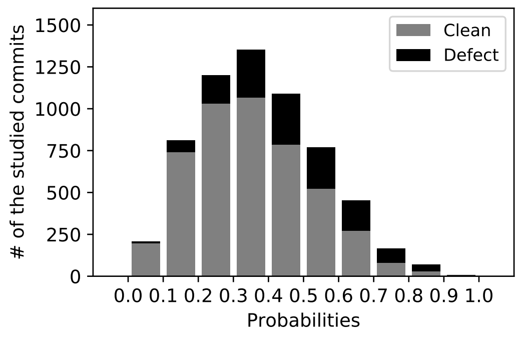
Table 4.10 shows the values of the difference of the LR model. The gray cells



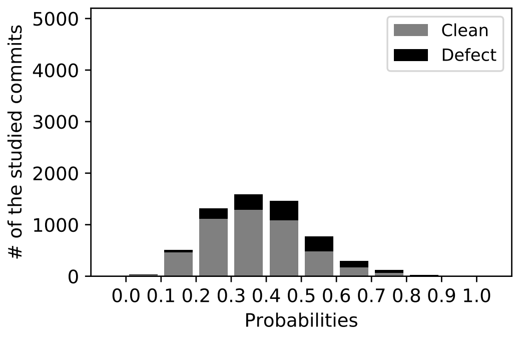
(a) NCCW RF



(b) NCCW LR



(c) Changes RF



(d) Changes LR

Figure 4.9: The number of studied commits in Camel project. The x-axis refers to the probabilities using each feature on either RF (left column) or LR (right column) models.

indicate the smallest values of the difference between the features in a project. We can observe that the change features achieve gray cells in the majority (5 of 6) of projects. This result implies that although probabilities that were computed by the NCCW are close to 0.5 than the change features, the difference of the results of the change features is smaller than NCCW. Hence, the probabilities are close to the correct label than NCCW. This is the reason why the change features provide that all projects are in the top three rank on Brier score.

The indentation feature, AS, is the best-performing feature on AUC and MCC according to the double Scott-Knott ESD test.

Table 4.10: The values of our proposed difference of the LR model. The gray cells refer to the smallest difference values by the features within each project.

Feature Types	Features	Projects					
		Bitcoin	Camel	Gerrit	Gimp	Hadoop	Osmand
Context	NCW(c,1,+)	747	2,981	1,883	3,568	1,403	1,188
	NCKW(c,10,+)	744	2,989	1,907	3,577	1,426	1,199
	NCCW(c,1,+)	700	2,859	1,819	3,437	1,373	1,197
	NCCKW(c,10,+)	700	2,913	1,856	3,529	1,402	1,202
Indentation	AS	706	2,882	1,843	3,514	1,410	1,204
	AB	758	2,885	1,849	3,688	1,403	1,204
Traditional	Changes	675	2,509	1,749	2,898	1,212	1,203
	NS	836	3,001	1,972	3,281	1,534	1,233
	ND	818	2,921	1,937	3,411	1,431	1,222
	NF	790	2,935	1,955	3,590	1,458	1,219
	Entropy	782	2,815	1,847	3,395	1,388	1,194
	LA	825	2,905	1,908	3,589	1,492	1,212
	LD	830	3,032	2,022	3,673	1,526	1,220
	LT	834	3,031	2,060	3,780	1,294	1,238

Figure 4.10 shows the results of the double Scott-Knott ESD test on the results for each studied feature in all projects; each boxplot contains six ranks of the first Scott-Knott ESD test execution for the studied projects on a studied feature. We use two gray colors (dark and light gray) and two lines (solid and dashed lines) to represent the ranks according to the double Scott-Knott ESD test; the adjacent boxplots with the same gray color and line indicate the same rank. Otherwise, the rank is changed at that point. The different rank indicates a statistical significant difference with small effect size or over according to the double Scott-Knott ESD test. We observe that AS is the best-performing feature on both AUC and MCC. The change features are the best-performing features on Brier score, and AS is the second best-performing feature. This result provides that AS is a top rank feature across the studied projects on AUC and MCC, and the change features are the top

rank features across the studied projects on Brier score.

The extended context feature, NCCW, and the churn feature, LA, are also better features according to the double Scott-Knott ESD test.

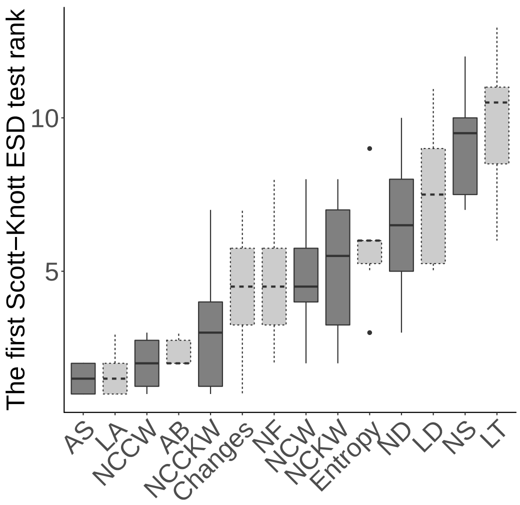
LA provides the second-rank performance in AUC and Brier score, and the first rank performance in MCC as well. The extended context feature NCCW provides the third rank performance in AUC, the second rank performance in Brier score, and the first rank performance in MCC as well. This result provides that NCCW and LA are also better features across the studied projects on AUC and MCC.

In this RQ, we study the features in terms of the prediction performance. However, we ignore other aspects such as detected defective commits. We closely look at the detected defective commits, pair-wise relation across the studied features, and the basic predicting power of the studied features in Section 4.7 (discussion).

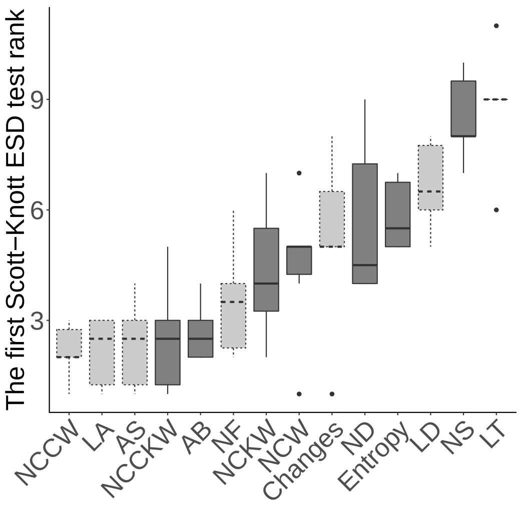
4.6.3 RQ3. What is the impact of combination features of context features on defect prediction?

"goto" statement is the best keyword for the modified NCCKW.

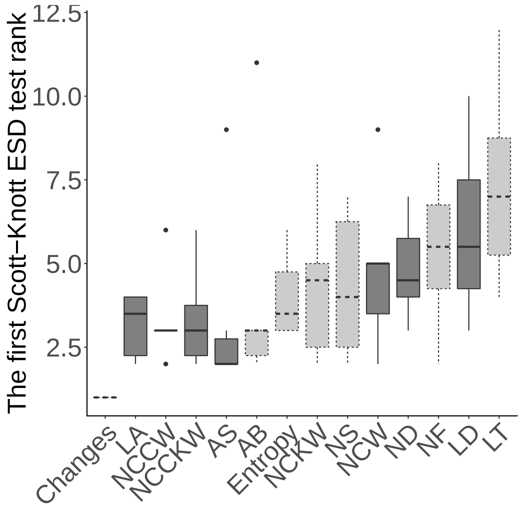
Figure 4.11 shows the results of the double Scott-Knott ESD test on the results for each modified NCCKW in all projects. Each boxplot contains six ranks of the first Scott-Knott ESD test execution within a studied project for all projects using a studied keyword as the modified NCCKW. The x-axis indicates a keyword which is used on the modified NCCKW; the y-axis indicates the rank for each studied project in the first Scott-Knott ESD test execution. We use two gray colors (dark and light gray) and two lines (solid and dashed lines) to represent the ranks according to the double Scott-Knott ESD test; the adjacent boxplots with the same gray color and line indicate the same rank. Otherwise, the rank is changed at that point. The different rank indicates a statistical significant difference with small effect size or over. The first Scott-Knott ESD test is applied to the values of the evaluation measures that were computed by the results of the studied prediction



(a) AUC

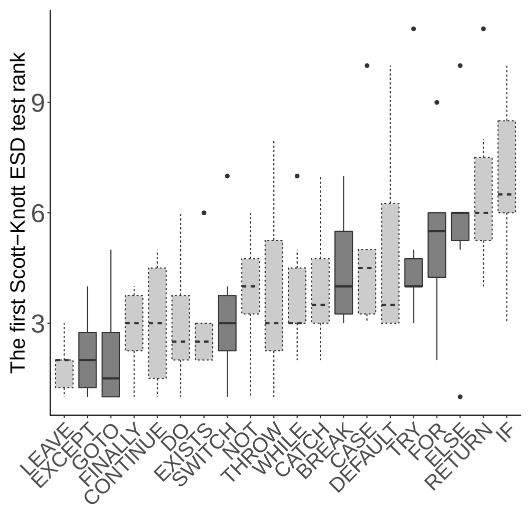


(b) MCC

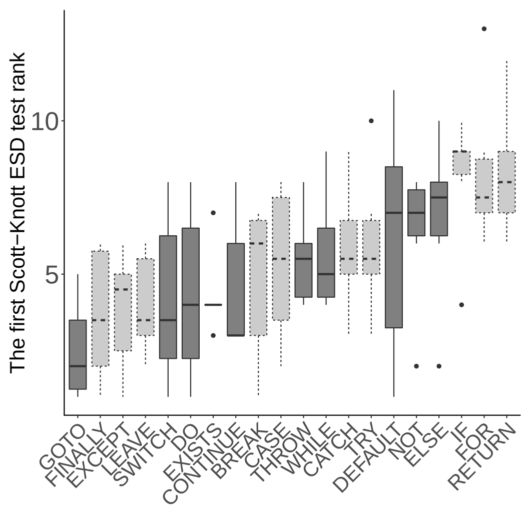


(c) Brier score

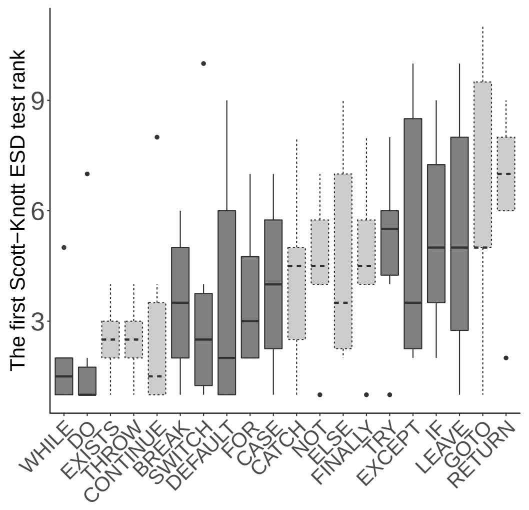
Figure 4.10: The double Scott-Knott ESD test results for each studied feature in all projects. Please see text for a full explanation.



(a) AUC



(b) MCC



(c) Brier score

Figure 4.11: The results of the double Scott-Knott ESD test on the results for each modified NCCKW in all projects. Please see text for a full explanation.

models that use NCCW and a certain modified NCCKW which uses a certain keyword (e.g., “goto”) as the explanation variables.

We observe that the number of “goto” statement in the context and changed lines achieves the top-1 or 2 rank in AUC and MCC. In addition, the median rank value is the best in AUC and MCC. The number of “goto” statement achieves the worst rank in Brier score. From the same reason of RQ2 results in Brier score, we conclude that the modified NCCKW which counts the number of “goto” statements is the strongest feature on the combination with NCCW. In addition, the modified NCCKW is not strongly correlated with NCCW (see Table 4.21). Hereafter, we refer to this variant (using the number of “goto” statement) of the modified NCCKW as *gotoNCCKW*. We use NCCW and gotoNCCKW for a prediction model in order to improve the prediction performance. We refer to the combination features as *COMB*.

COMB provides the top-one rank prediction performance for all the studied projects in AUC and MCC.

Table 4.11 shows the ranks according to the Scott-Knott ESD test results of the three evaluation measures for each studied feature. We observe that COMB provides the top-one rank prediction performance for all the studied projects in AUC and MCC. In addition, except AS in MCC, there exists no other studied features that achieve the top-one rank prediction performance. This result indicates that COMB are the best prediction features in all the studied features. COMB achieves at least the top-three rank prediction performance for all studied projects in Brier score.

COMB statistically outperforms the other studied features.

Figure 4.12 shows the results of the double Scott-Knott ESD test on the results for each studied feature in all projects; each boxplot contains six ranks of the first Scott-Knott ESD test execution for the studied projects on a studied feature. The x-axis indicates a feature; the y-axis indicates the rank for each studied project in the first Scott-Knott ESD test execution. We use two gray colors (dark and light

Table 4.11: The ranks of the Scott-Knott ESD test results for studied features. *#R1* (*#R2*, or *#R3*) is the sum of the numbers of cases where the rank is one (two, or three); *Sum* = *#R1* + *#R2* + *#R3*. The actual values that were computed by RF and LR are shown in Appendix.

(a) AUC

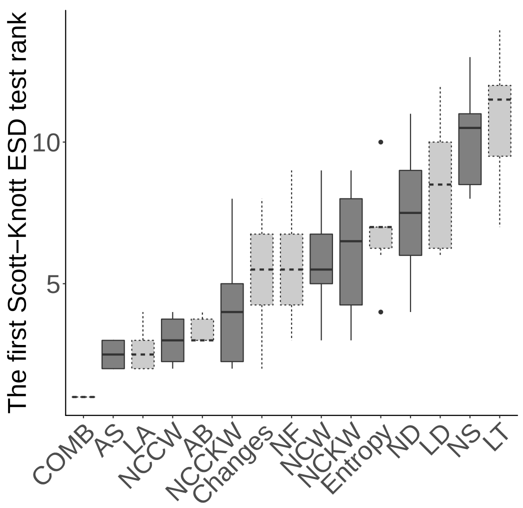
Feature Types	**Features**	**Projects**						**Numbers of Ranks**			
		B.	**C.**	**Ge.**	**Gi.**	**H.**	**O.**	*#R1*	*#R2*	*#R3*	*Sum*
Context	NCW(c,1,+)	7	9	6	5	5	3	0	0	1	1
	NCKW(c,10,+)	8	9	8	5	4	3	0	0	1	1
	NCCW(c,1,+)	2	4	4	2	3	3	0	2	2	4
	NCCKW(c,10,+)	5	8	5	3	2	2	0	2	1	3
	COMB	1	1	1	1	1	1	6	0	0	6
Indentation	AS	2	2	2	3	3	3	0	3	3	6
	AB	4	3	3	4	3	3	0	0	4	4
Traditional	Changes	6	4	7	8	2	5	0	1	0	1
	NS	13	11	11	10	8	8	0	0	0	0
	ND	11	6	9	9	4	6	0	0	0	0
	NF	9	5	6	7	3	4	0	0	1	1
	Entropy	10	7	7	7	6	4	0	0	0	0
	LA	3	2	4	2	2	3	0	3	2	5
	LD	12	10	10	6	6	7	0	0	0	0
	LT	14	12	12	11	7	9	0	0	0	0

(b) MCC

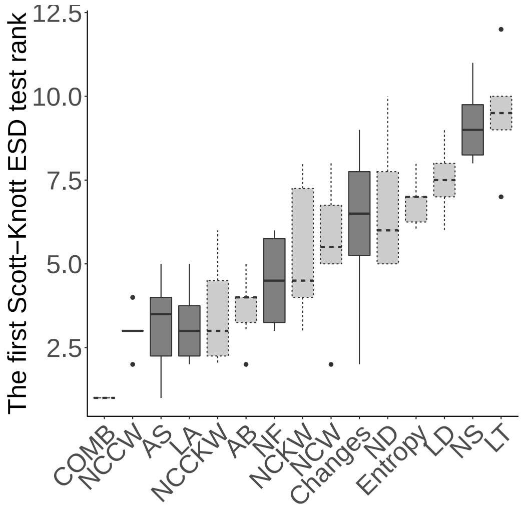
Feature Types	**Features**	**Projects**						**Numbers of Ranks**			
		B.	**C.**	**Ge.**	**Gi.**	**H.**	**O.**	*#R1*	*#R2*	*#R3*	*Sum*
Context	NCW(c,1,+)	5	8	7	6	5	2	0	1	0	1
	NCKW(c,10,+)	4	8	8	4	5	3	0	0	1	1
	NCCW(c,1,+)	3	2	3	3	4	3	0	1	4	5
	NCCKW(c,10,+)	3	6	5	2	3	2	0	2	2	4
	COMB	1	1	1	1	1	1	6	0	0	6
Indentation	AS	1	3	2	5	4	4	1	1	1	3
	AB	2	4	4	3	5	4	0	1	1	2
Traditional	Changes	5	6	7	9	2	8	0	1	0	1
	NS	10	9	8	11	8	9	0	0	0	0
	ND	8	5	7	10	5	5	0	0	0	0
	NF	6	3	6	5	3	4	0	0	2	2
	Entropy	7	7	7	8	6	6	0	0	0	0
	LA	3	2	5	2	4	3	0	2	2	4
	LD	8	9	8	7	6	7	0	0	0	0
	LT	9	10	9	12	7	10	0	0	0	0

(c) Brier Score

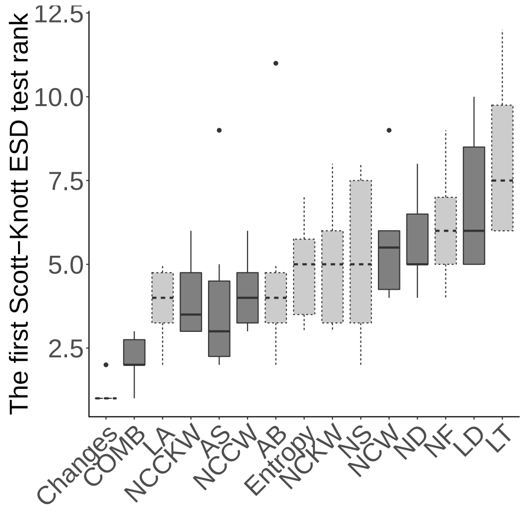
Feature Types	**Features**	**Projects**						**Numbers of Ranks**			
		B.	**C.**	**Ge.**	**Gi.**	**H.**	**O.**	*#R1*	*#R2*	*#R3*	*Sum*
Context	NCW(c,1,+)	6	5	6	9	4	4	0	0	0	0
	NCKW(c,10,+)	6	4	6	8	3	3	0	0	2	2
	NCCW(c,1,+)	4	3	4	6	5	3	0	0	2	2
	NCCKW(c,10,+)	4	3	5	6	3	3	0	0	3	3
	COMB	3	3	2	2	2	1	1	3	2	6
Indentation	AS	2	2	3	9	5	3	0	2	2	4
	AB	4	2	4	11	5	3	0	1	1	2
Traditional	Changes	1	1	1	1	1	2	5	1	0	6
	NS	8	4	8	2	6	3	0	1	1	2
	ND	8	4	7	5	5	5	0	0	0	0
	NF	9	5	7	7	4	5	0	0	0	0
	Entropy	7	3	6	3	5	5	0	0	2	2
	LA	5	2	4	4	5	3	0	1	1	2
	LD	9	5	7	10	5	5	0	0	0	0
	LT	10	6	9	12	6	6	0	0	0	0



(a) AUC



(b) MCC



(c) Brier score

Figure 4.12: The results of the double Scott-Knott ESD test on the results for each studied feature in all projects. Please see text for a full explanation.

gray) and two lines (solid and dashed lines) to represent the ranks according to the double Scott-Knott ESD test; the adjacent boxplots with the same gray color and line indicate the same rank. Otherwise, the rank is changed at that point. The different rank indicates a statistical significant difference with small effect size or over.

We observe that COMB are the best-performing features on both AUC and MCC. This result provides that COMB are the top rank features across the studied projects on AUC and MCC. Even on Brier score, COMB are the second rank features. The best-performing features on Brier score is still the change features.

4.7 Discussion

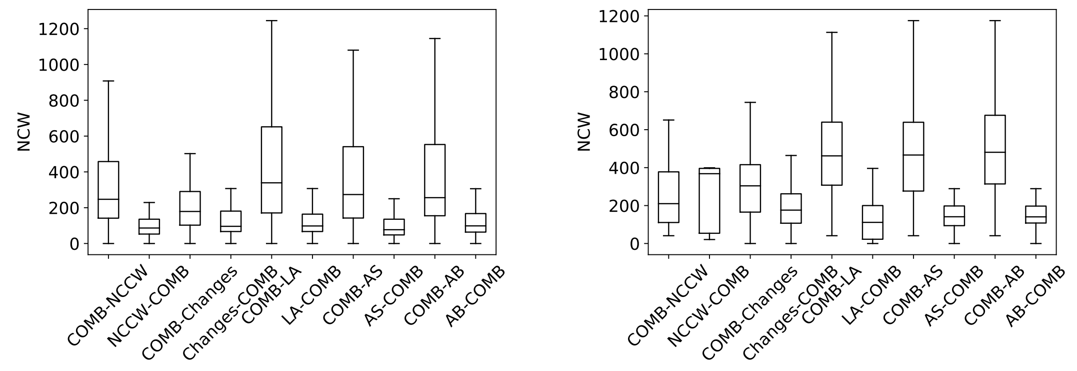
4.7.1 Are the commits identified by the context features different than the ones identified by the traditional churn features?

The proposed context features COMB identify some defective commits that other churn features cannot; these commits tend to have large context lines.

We define *unique* defective commits as the commits that are only identified by our proposed features (and not by other features). The existence of these defective commits contributes to defect prediction since they cannot be identified using traditional churn features. Hence, we study the commits identified as defective by COMB.

Figure 4.13 shows the values of the context feature NCW for the commits identified as defective in Hadoop project. We can observe that COMB identifies the commits that have higher NCW values as defective compared to the other features. For example, the median NCW value of COMB-Changes is higher than the median NCW value of Changes-COMB (Figure 4.13(a) and 4.13(b)). The results for the other projects show the same tendency except NCCW; NCCW has higher NCW values in 4 of 6 projects since NCCW is also a context feature.

Because we use NCW values to show unique defective commits, this result



(a) RF

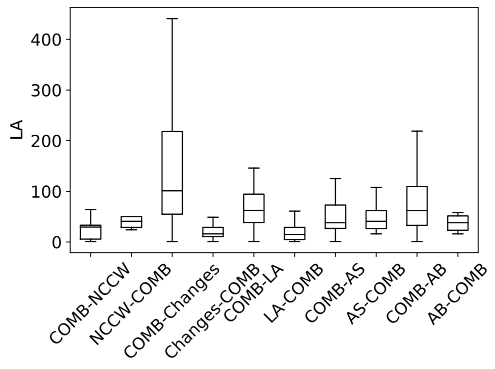
(b) LR

Figure 4.13: The values of the context feature NCW for the commits identified as defective in Hadoop project. The boxplots show the cases where COMB identified the commits differently with the context feature NCCW, the change features, LA and the indentation features on RF and LR models. For instance, COMB-AB refers to the cases where commits are identified as defective by COMB but are identified as clean by AB. The x-axis shows the features that are compared; the y-axis shows the value of NCW.

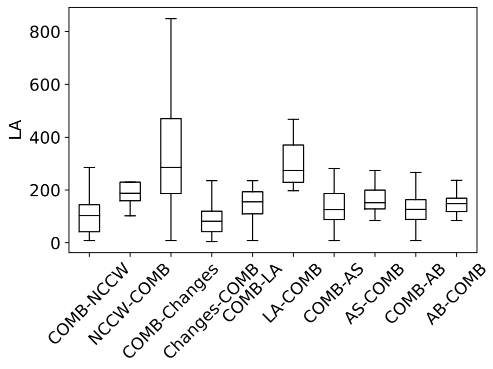
may seem obvious. However, even if we use LA value to show unique defective commits, the median LA value of COMB-LA is higher than the median LA value of LA-COMB in several projects. Figure 4.14 shows the values of LA for the commits identified as defective by LR model in Bitcoin project and Hadoop project. In Bitcoin project, the median LA value of COMB-LA is higher than the median LA value of LA-COMB, while LA-COMB has higher median LA value in Hadoop project. This result implies that the result in Figure 4.13(a) and 4.13(b) indicates that COMB can uniquely identify some defective commits.

The proposed context features NCW and NCKW, and the extended context features NCCW and NCCKW can uniquely identify defective commits; and these commits tend to have larger context lines than other churn features on the LR model.

We observe the same tendency for the other context features on LR model,



(a) Bitcoin



(b) Hadoop

Figure 4.14: The values of LA for the commits identified as defective in Bitcoin project and Hadoop project. The boxplots show the cases where COMB identified the commits differently with the context feature NCCW, the change features, LA and the indentation features on LR model. The y-axis shows the value of LA.

but not RF model. This result may be from the difference between RF and LR models. To study the difference between the prediction models lies beyond the scope of this chapter. In addition, there exist commits that the traditional code churn features can identify that the context features cannot. Future studies are necessary to investigate these points.

4.7.2 How much do the indentation features improve the defect prediction performance?

Indentation features AS and AB have the potential to improve defect prediction performance.

Our study is the first applying the indentation features to the defect prediction problem. From our results, the indentation features are one of the best features on defect prediction performance, and significantly outperform other studied features without COMB. Hence, we observe that the indentation features have the potential of predicting power for just-in-time defect prediction.

4.7.3 How redundant are the context features compared to the traditional features?

Motivation:

To our knowledge, prior work in defect prediction disregards information around the changed lines, context lines. Hence, we propose the context features, and study the impact of them in the defect prediction performance. However, we did not study the redundancy of our context features compared to the traditional features.

We present an in-depth analysis to understand the relation between our context features and the traditional features. This result produces insights of why our context features are not inducing redundancy, and why the context features can uniquely identify defective commits compared to the traditional features. Finally, we show the basic predicting power using *information gain* [140].

Approach:

We first study five context features (i.e., NCW, NCKW, NCCW, NCCKW, and gotoNCCKW[‖]), two indentation features and 14 traditional change features based on a correlation analysis [188] and the principal component analysis (PCA) [28] to identify correlated features and find features that are important to represent the variance of the original features. Second, we compute information gain [140] for all the studied features in order to clarify the basic predicting power of the studied features.

We first conduct a correlation analysis on the features. When we use strongly correlated features as explanation variables for a prediction model, we get the problem of multicollinearity [37]. In addition, these features are redundant. We use Spearman rank correlation [188] to measure the correlation between the fea-

[‖]COMB are two context features NCCW and gotoNCCKW. Hence, we study NCCW and gotoNCCKW instead of COMB.

tures. Spearman rank correlation is a non-parametric correlation. We apply Spearman rank correlation to all commits on each studied project. We compute the average values of the correlation coefficients between the projects.

Second, we conduct the PCA in order to identify features which represent the highest variance of all the studied features. The PCA result shows which features can represent the variance of all the studied features. The PCA reduces the number of input features and makes new features. Then, the PCA shows the coefficient[**] for every new feature to convert the input features into the new feature. We use the coefficient of the most important new feature called the first principal component[††] to identify which features represent the highest variance[‡‡]. We apply the PCA to all commits on each studied project. We suppose that features which represent the highest variance are important features in the studied features.

Finally, we compute information gain [140] in order to clarify the basic predicting power of the studied features. In our case, information gain measures the basic predicting power of each of the features. For example, if an original feature perfectly separates defective commits and clean commits, the value of information gain would be maximum. However, if an original feature separates all the commits to 50% defective commits and 50% clean commits, the value of information gain would be minimum because this prediction is the same as random classification. The formula of information gain [140] is as follows:

$$InfoGain(feature) = H(Defect) + H(feature) - H(Defect, feature),$$

where *feature* is a certain studied feature, *InfoGain*$(\cdot)$ is the information gain of $\cdot$ (*feature*), $H(\cdot)$ is Shannon entropy [142] of $\cdot$ where the base of the logarithm is 2, $H(\cdot, \cdot')$ is Shannon entropy of $\cdot$ after classifying by $\cdot'$, *Defect* is the set of all commits

[**]Here, the coefficient means the left-singular vector. We conduct the PCA using singular vector decomposition.

[††]The first principal component means the input features set that can very retain the original features variance.

[‡‡]Features which represent higher variance of the studied features have higher coefficient in the first principal component.

with prediction results (defective or clean).

We compute the ratio of the information gain between NCCW, and the indentation features and the churn features. Since NCCW is our proposed feature, we use NCCW as a base. The formulation is as follows:

$$Ratio = InfoGain(NCCW)/InfoGain(\cdot),$$

If the ratio is over 1.0 when using a certain original feature, NCCW has high potential to classify the commits in defect prediction rather than the certain feature.

Results:

The context features NCCW and NCCKW, the indentation features AI and AS, and the change feature LA are strongly correlated.

Table 4.12 shows the Spearman rank correlation between all the studied features (including the context, the indentation and the change features) in all studied projects; each cell in the table shows the average correlation in the studied projects (the median is very similar). A gray cell refers to the case of the strong correlation whose coefficient is 0.7 and over. We observe that the correlations between NCCW, NCCKW, AI, AS, and LA are strong (over 0.7). This is because the context features and the indentation features include changed lines information.

The context features NCW and NCKW, however, are moderately correlated to the indentation features and the change feature LA.

NCCW and NCCKW are extended features of NCW and NCKW. NCW and NCKW are moderately correlated to AI, AS, and LA (less than 0.7). Hence, although the context information have a similar concept with the indentation features and changed lines, the context information is not redundant.

The context features NCCW and NCCKW are the features that represent the highest variance of all the original features.

Table 4.13 shows the coefficient of the first principal component for each project in the PCA. A gray cell refers to the case with the absolute coefficient 0.3 and over.

Table 4.12: Spearman rank correlation between the context features, the indentation features, and the change features in the studied projects. GNCCKW indicates gotoNCCKW. We average correlations in the studied projects. Each cell shows the average correlation. "*" refers to that at least one non statistical significant correlation in the studied projects. Due to the space limitation, we omit the History and Experience of the change features in this chapter. These features are not strongly correlated (0.7 and over) with the other features (except for the correlation between EXP and SEXP).

	NCW	NCKW	NCCW	NCCKW	GNCCKW	AI	AS	FIX	NS	ND	NF	Entropy	LA	LD	LT
NCW	1.00	0.75	0.80	0.72	0.06*	0.59	0.64	-0.08	0.24	0.50	0.65	0.61	0.64	0.60	-0.01*
NCKW		1.00	0.65	0.88	0.06*	0.58	0.61	-0.06*	0.20	0.39	0.51	0.47	0.54	0.53	0.10
NCCW			1.00	0.81	0.08*	0.84	0.90	-0.14	0.27	0.54	0.70	0.62	0.91	0.60	-0.07*
NCCKW				1.00	0.08*	0.76	0.79	-0.11	0.24	0.46	0.60	0.54	0.75	0.55	0.03*
GNCCKW					1.00	0.06*	0.07*	-0.01*	0.04*	0.05*	0.07*	0.06*	0.08*	0.06*	-0.03*
AI						1.00	0.92	-0.10	0.22	0.42	0.52	0.44	0.82	0.48	-0.02*
AS							1.00	-0.12	0.22	0.43	0.56	0.47	0.87	0.52	-0.02*
FIX								1.00	-0.05	-0.10*	-0.14	-0.12	-0.16	-0.09	0.05*
NS									1.00	0.60	0.46	0.44	0.33	0.24	0.06*
ND										1.00	0.81	0.77	0.58	0.45	-0.04
NF											1.00	0.96	0.71	0.58	-0.09
Entropy												1.00	0.62	0.52	-0.08
LA													1.00	0.58	-0.08
LD														1.00	-0.00*
LT															1.00

Table 4.13: The coefficient of the first principal component for each project in the PCA. GNCCKW indicates gotoNCCKW. Please see text for a full explanation.

	Hadoop	Camel	Gerrit	Osmand	CMake	Bitcoin	Gimp
NCW	-0.300	-0.264	-0.312	-0.237	-0.206	-0.297	-0.189
NCKW	-0.295	-0.259	-0.289	-0.228	-0.201	-0.294	-0.180
NCCW	-0.341	-0.345	-0.347	-0.411	-0.370	-0.366	-0.379
NCCKW	-0.376	-0.346	-0.330	-0.410	-0.374	-0.376	-0.394
GNCCKW	-0.040	0.002	-0.009	-0.199	-0.313	-0.039	-0.212
AI	-0.282	-0.294	-0.266	-0.369	-0.357	-0.299	-0.263
AS	-0.285	-0.294	-0.266	-0.381	-0.359	-0.309	-0.363
FIX	0.055	0.052	0.024	0.028	0.024	0.025	0.014
NS	-0.135	-0.170	-0.232	-0.075	-0.100	-0.192	-0.167
ND	-0.342	-0.277	-0.318	-0.129	-0.179	-0.249	-0.291
NF	-0.299	-0.334	-0.301	-0.175	-0.282	-0.330	-0.268
Entropy	-0.289	-0.277	-0.272	-0.117	-0.153	-0.279	-0.194
LA	-0.210	-0.292	-0.136	-0.378	-0.317	-0.206	-0.359
LD	-0.174	-0.175	-0.215	-0.094	-0.162	-0.078	-0.109
LT	0.114	0.021	0.031	-0.098	0.025	0.053	0.016
NDEV	0.009	-0.005	-0.006	-0.014	-0.025	0.006	0.095
AGE	-0.018	-0.003	-0.007	-0.034	-0.037	-0.004	0.005
NUC	-0.024	-0.194	-0.250	-0.052	-0.099	-0.161	-0.045
EXP	-0.042	0.008	-0.011	-0.033	-0.003	-0.019	0.068
REXP	0.002	0.010	0.016	0.009	0.003	0.014	0.006
SEXP	-0.039	0.027	-0.002	-0.022	0.021	-0.019	0.085

We observe that NCCW and NCCKW have over 0.3 absolute coefficient in all the studied projects. If the first principal component has a certain feature which has high coefficient in all the projects, this feature is likely to represent the highest variance of all studied features in all the projects.

NCCW and NCCKW include the context information and have the strong correlation to the indentation features and LA due to using changed line information. Hence, NCCW and NCCKW can add the context information while having the information of the indentation features and LA. Hence, NCCW and NCCKW represent the highest variance.

In summary, the context features NCW and NCKW are not redundant features, and add the context information to the defect prediction model. While NCCW and NCCKW have strong correlations to the indentation features and LA, NCCW and NCCKW also add information from the context of the change.

Except for LT, NCCW has the strongest basic predicting power regarding the information gain compared to other studied features.

Figure 4.15 shows the ratio of the information gain. We observe that all the median values are grater than 1.0 except LT. Hence, almost all cases, the information gain of NCCW is better than the other studied features. LT has better value of the information gain. However, the prediction performance such as AUC is not good. In summary, except for LT, NCCW has the strongest basic predicting power in the studied features.

4.7.4 Does the context size changes the complexity of change?

We argued that more words/keywords in a context, more complex a change is. Although number of words/keywords are determined by the context size, we were concerned about that the complexity is changed by the context size. In this discussion, we explain that changing context size does not affect the complexity of change.

From our experiments, given a fixed size of context, the number of words/keywords

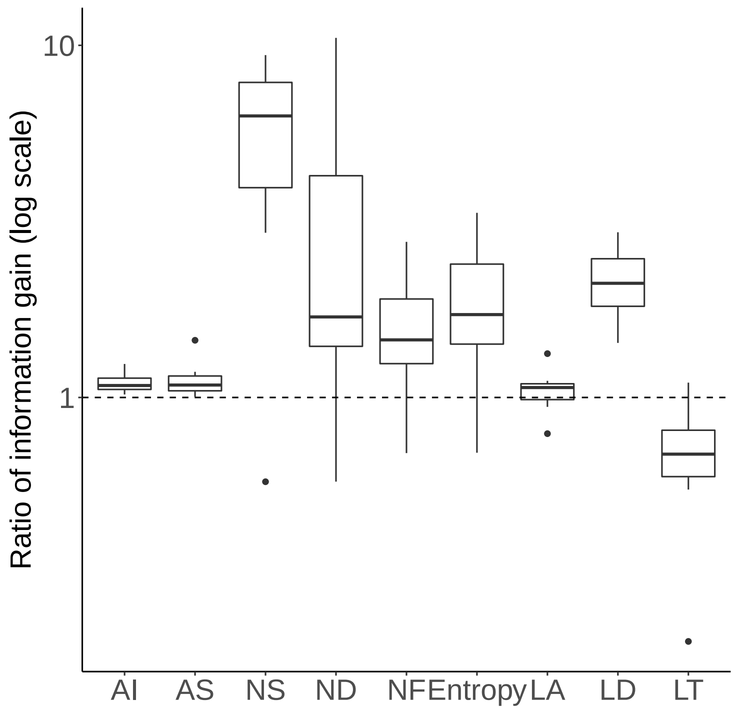


Figure 4.15: The ratio of the information gain between NCCW and other features. The x-axis indicates the features that are used to compute the ratio; the y-axis indicates the ratio. The dashed line indicates that the ratio is 1.0.

in such context is a good indicator of the complexity of the change (RQ1). This is because as the context size increases, the number of context words/keywords also increases; however, the distance of some words/keywords to the hunk will also increase, making them less effective as an indicator of complexity. Hence, a balance is required: too small a context might not have enough information to capture the context of the change, however a context that is too large will dilute the important context information around a hunk.

4.7.5 How are the actual AUC and MCC values of the context features?

We study the ranks that were computed by the Scott-Knott ESD test across the studied features to determine which are the best prediction features to use in defect prediction. However, practitioners would concern about the actual AUC and MCC values since practitioners need accurate prediction model.

We show the actual AUC and MCC values in Appendix (Table 4.18 and 4.19). From the AUC result (Table 4.18), COMB provides at least 0.737. This value corresponds to the strong effect size according to prior work [135]. From the MCC result (Table 4.19), COMB provides at least 0.3 except RF in the Camel project. This value corresponds to the moderate correlation. Hence, we conclude that COMB can be used in practice since they have acceptable prediction performance in the actual values as well.

4.7.6 Practical guides (recommendations) for the parameters of the context features

The context features have two tunable parameters: the context size, and the churn type. We made our practical guides (recommendations) of optimizing the parameters of the context features as applicable as possible to practitioners.

Recommendation 1: If practitioners have both, training data and validation data, we recommend to optimize the context size and the churn type following our experiments in RQ1. The most important parameters to determine are how many context lines to use (we call this the context size) and what type of context lines to use (we call this the churn type). In our study, we calculated the context size and the churn type that yield the best results; we recommend that, if practitioners have training data and validation data, they optimize the context size and the churn type following our experiments in RQ1. Our experiments in RQ3 show that COMB which are the combination features of the extended context features that are number of words and number of "goto" keyword significantly outperform the other studied features. Hence, if practitioners want to use our prediction model, we recommend to use COMB. Practitioners do not need to decide using either number of keywords or words as a parameter of the context features. COMB includes both of them. The details of how to use COMB can be found in Section 4.7.7.

Recommendation 2: If practitioners do not have enough validation data, we recommend to use the same parameters that we found perform best. Our experiments in RQ1 show optimal values for the parameters for the projects we studied. The studied projects cover multiple domains of software, and two popular programming languages, C++ and Java. We believe this diversity of studied projects is likely to make these parameters useful in general.

4.7.7 Practical guides (recommendation) for practitioners who want to use a defect prediction model

We proposed the context features. We present recommendations of using them for defect prediction according to the experimental results.

Recommendation 1: Use the indentation feature AS instead of the traditional size features in the change features. Our experiments in RQ2 show that AS significantly outperforms the other studied features including traditional size features (LA, LD and LT). In addition, AS is strongly correlated with the traditional size feature LA which has the highest performance in the change (code churn) features. Hence, using AS instead of the traditional size features allows practitioners to improve the performance of their defect prediction models.

Recommendation 2: For the case where practitioners want to improve the prediction performance using a simple prediction model, use the context features COMB on the logistic regression model. Our experiments in RQ3 show that COMB are the best-performing features in AUC and MCC. In addition, our discussion shows that: (1) a context feature used in COMB, NCCW, is one of the feature that represents the highest variance of all the original features, and (2) the basic defect predicting power of NCCW is strong. For the interpretation of the prediction model, COMB contains only two features (NCCW and gotoNCCKW), and therefore, we can easily interpret the prediction results. Finally, the effect size of the actual prediction values in AUC is strong. Hence, for the case where practitioners want to improve the prediction performance using a simple prediction model, using COMB might allow practitioners to get good prediction performance with a simple prediction model.

4.8 Threats to validity

4.8.1 Construct validity

We follow the labeling process in Commit Guru [141] in order to label each commit either defective or clean. SZZ algorithm is also a popular approach to identify defective commits [150]; however, it has no open source implementation available. In contrast, Commit Guru is a publicly available open source project. Hence, we follow the labeling process in Commit Guru for its repeatability and openness.

We use the online change classification [155] to validate the performance of defect prediction. This validation technique addresses the challenges of the cross validation technique. Hence, we believe this validation technique is acceptable.

The online change classification has parameters. In particular, the unit (test interval) is the most important parameter. Below, we studied the impact of the unit for the performance in defect prediction. If the unit has strong impact for the performance in defect prediction, we would need to consider the parameter in our experiments.

Approach: We build defect prediction models for NCCW, NCCKW, COMB, AS, LA, and the change features. The prediction procedure is the almost same as RQ2. The only difference is that we change the unit value between 10 to 100 by 10. Finally, we report the evaluation measures by (1) plotting a line plot for each project, prediction model, and studied feature, and (2) computing the median and 75 percentile IQR values of different unit values in all projects, prediction models, and studied features.

Results: Figure 4.16 shows the values of the evaluation measures for different unit values. We observe that all evaluation measures are stable for different unit

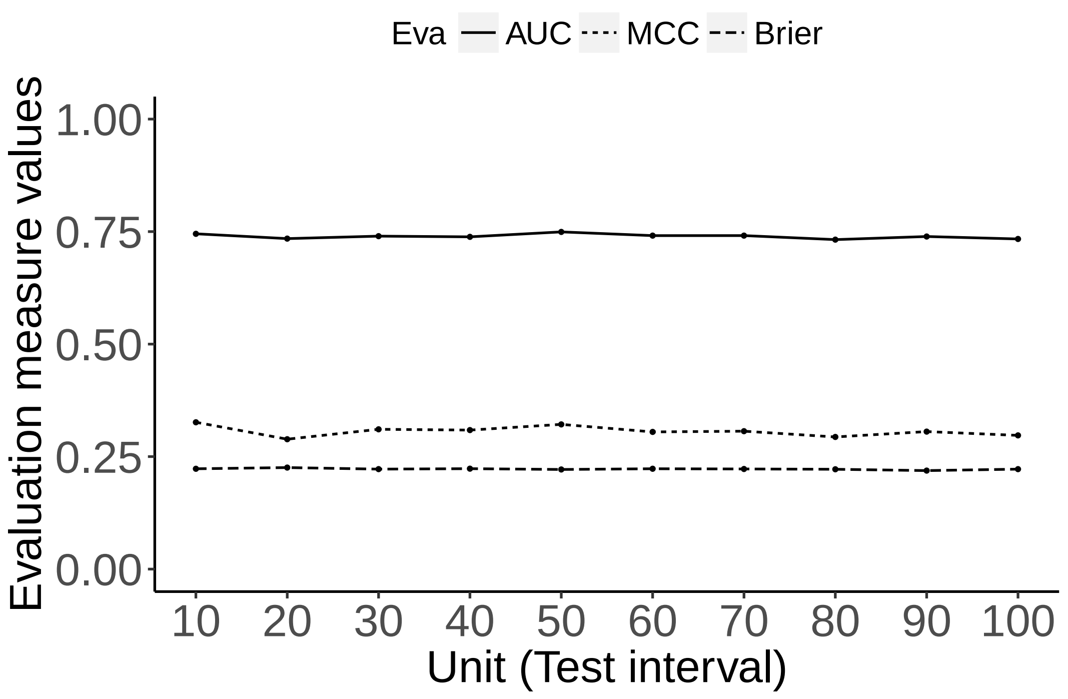


Figure 4.16: The values of the evaluation measures for each unit (test interval) using the NCCW feature on LR model in the Camel project. Eva indicates evaluation measures. The x-axis indicates the unit value between 10 to 100; the y-axis indicates the values of the evaluation measures.

values. In addition, we observe the same tendency for other projects, prediction models, and features.

Table 4.14 shows the median and 75 percentile (3Q) IQR values for different unit values in all projects, prediction models, and studied features. We observe that even if we check 3Q values, they are less than 0.05 IQR value in all cases. Hence, the unit (test interval) has little impact for the results. The training interval is decided by the unit. Hence, the training interval also has little impact for the results.

Table 4.14: The median and 75 percentile (3Q) IQR values of the performance for the context features, the indentation feature and the change features. An IQR value is computed across all unit values for each prediction model for each project for each feature. The median/3Q IQR values are computed for each feature. Hence, the median/3Q for all prediction models and projects.

	NCCW		**NCCKW**		**COMB**		**AS**		**LA**		**Changes**	
	Median	3Q	Median	3Q	Median	3Q	Median	3Q	Median	3Q	Median	3Q
AUC	0.011	0.016	0.012	0.015	0.008	0.014	0.010	0.013	0.009	0.013	0.015	0.017
MCC	0.015	0.026	0.019	0.022	0.024	0.028	0.019	0.021	0.022	0.030	0.032	0.037
Brier	0.004	0.007	0.004	0.009	0.009	0.010	0.004	0.008	0.005	0.007	0.005	0.006

4.8.2 External validity

As the studied projects, we use six large open source software. These software are written in the popular programming languages C++ and Java; and one of various types of software, such as server and web application. These systems we study are open source but not commercial software. In the future, we need to study the context features, extended context features, and combination context features on commercial projects for verifying our findings.

4.8.3 Internal validity

We remove comments from the hunks. However, if all lines in a hunk are comments and use "/**/", we do not identify whether the hunks are comments.

We use three evaluation measures, AUC, MCC and Brier score, which are not affected by skewed data [15,181] and address the pitfalls in defect prediction [160]. Hence, we believe these measures are acceptable.

4.9 Chapter summary

In this chapter, we propose context features based on the context lines, the extended context features based on both the context lines and changed lines as code churn features, and COMB based on the extended context features. We study the impact of considering the context lines for defect prediction. We compare the context features, the extended context features, and COMB with the traditional code churn features in six open source software. The main findings of this chapter are as follows:

- The chunk type '+' is the best parameter for context features for defect prediction. This chunk type achieves the best median rank according to the three evaluation measures, AUC, MCC and Brier score on the Scott-Knott ESD test.
- The small context size is suitable when considering the number of words, while the large context size is suitable when considering the number of keywords in context lines for defect prediction.
- "goto" statement in the context lines and the changed lines is the best keyword to detect defective commits in the modified NCCKW.
- Our proposed combination features, COMB, significantly outperform all the features, and achieve the best-performing features in all of the studied projects in terms of AUC and MCC.

4.10 Appendix

In this appendix, we show the actual values of the three evaluation measures corresponding to the results of the rank in RQ1, RQ2, and RQ3. In addition, we show the correlation across the modified NCCKW.

Table 4.15: The median values of AUC for each context feature variant and studied project. Each cell indicates the values of AUC by RF (left) and LR (right) models, respectively, when the context size is three. The Scott-Knott ESD test is conducted for each project (column).

Features	**Chunk**	**Projects**					
	Types	**Bitcoin**	**Camel**	**Gerrit**	**Gimp**	**Hadoop**	**Osmand**
NCW	+	0.619/0.727	0.579/0.610	0.642/0.722	0.692/0.722	0.624/0.738	0.585/0.757
	-	0.580/0.663	0.548/0.561	0.594/0.655	0.619/0.672	0.636/0.690	0.567/0.695
	all	0.631/0.707	0.577/0.598	0.624/0.707	0.647/0.715	0.638/0.737	0.587/0.749
NCKW	+	0.650/0.694	0.580/0.592	0.650/0.679	0.640/0.663	0.660/0.698	0.652/0.729
	-	0.594/0.640	0.549/0.567	0.598/0.620	0.618/0.649	0.639/0.677	0.619/0.682
	all	0.646/0.678	0.567/0.595	0.641/0.667	0.637/0.663	0.661/0.705	0.640/0.720

Table 4.16: The median values of MCC for each context feature variant and studied project. Each cell indicates the values of MCC by RF (left) and LR (right) models, respectively, when the context size is three. The Scott-Knott ESD test is conducted for each project (column).

Features	**Chunk**	**Projects**					
	Types	**Bitcoin**	**Camel**	**Gerrit**	**Gimp**	**Hadoop**	**Osmand**
NCW	+	0.130/0.269	0.090/0.178	0.170/0.316	0.203/0.268	0.156/0.361	0.118/0.415
	-	0.079/0.237	0.060/0.135	0.108/0.258	0.163/0.221	0.136/0.315	0.101/0.365
	all	0.152/0.271	0.084/0.186	0.160/0.309	0.172/0.251	0.171/0.344	0.129/0.418
NCKW	+	0.228/0.238	0.106/0.135	0.216/0.268	0.191/0.211	0.260/0.311	0.238/0.346
	-	0.178/0.232	0.093/0.135	0.168/0.234	0.185/0.199	0.201/0.297	0.227/0.309
	all	0.199/0.213	0.118/0.128	0.233/0.270	0.190/0.233	0.261/0.296	0.239/0.343

Table 4.17: The median values of Brier score for each context feature variant and studied project. Each cell indicates the values of Brier score by RF (left) and LR (right) models, respectively, when the context size is three. The Scott-Knott ESD test is conducted for each project (column).

Features	**Chunk**	**Projects**					
	Types	**Bitcoin**	**Camel**	**Gerrit**	**Gimp**	**Hadoop**	**Osmand**
NCW	+	0.313/0.214	0.310/0.238	0.289/0.221	0.237/0.230	0.293/0.216	0.340/0.238
	-	0.307/0.229	0.316/0.243	0.321/0.239	0.237/0.236	0.295/0.223	0.331/0.241
	all	0.311/0.215	0.334/0.239	0.302/0.223	0.242/0.231	0.293/0.217	0.332/0.238
NCKW	+	0.265/0.218	0.271/0.241	0.290/0.228	0.262/0.233	0.276/0.231	0.305/0.241
	-	0.270/0.227	0.261/0.244	0.329/0.236	0.247/0.237	0.266/0.233	0.332/0.243
	all	0.286/0.218	0.274/0.241	0.289/0.228	0.252/0.234	0.265/0.230	0.319/0.242

Table 4.18: The median values of AUC for each context feature, each extended context feature, COMB, each indentation feature, the change features and each of the change features. Each cell indicates the values of AUC by RF (left) and LR (right) models, respectively.

Feature	**Features**	**Projects**					
Types		**Bitcoin**	**Camel**	**Gerrit**	**Gimp**	**Hadoop**	**Osmand**
Context	NCW(c,1,+)	0.653/0.709	0.573/0.597	0.660/0.719	0.662/0.706	0.642/0.736	0.628/0.758
	NCKW(c,10,+)	0.648/0.695	0.576/0.599	0.643/0.679	0.675/0.689	0.684/0.716	0.628/0.744
	NCCW(c,1,+)	0.706/0.798	0.640/0.738	0.699/0.786	0.699/0.762	0.660/0.780	0.625/0.751
	NCCKW(c,10,+)	0.689/0.754	0.628/0.682	0.677/0.735	0.699/0.723	0.691/0.769	0.660/0.758
	COMB	0.750/0.798	0.741/0.738	0.771/0.786	0.743/0.768	0.765/0.780	0.737/0.751
Indentation	AS	0.735/0.780	0.675/0.739	0.742/0.785	0.693/0.737	0.682/0.763	0.636/0.749
	AB	0.734/0.739	0.667/0.733	0.720/0.775	0.692/0.689	0.693/0.754	0.614/0.750
Traditional	Changes	0.706/0.670	0.708/0.668	0.694/0.666	0.677/0.574	0.733/0.732	0.645/0.634
	NS	0.525/0.526	0.546/0.548	0.597/0.599	0.539/0.540	0.525/0.529	0.519/0.530
	ND	0.608/0.607	0.666/0.679	0.647/0.659	0.582/0.595	0.695/0.711	0.624/0.649
	NF	0.638/0.660	0.661/0.692	0.673/0.699	0.623/0.650	0.711/0.740	0.636/0.689
	Entropy	0.633/0.651	0.648/0.678	0.667/0.694	0.641/0.633	0.649/0.703	0.638/0.689
	LA	0.730/0.750	0.681/0.744	0.700/0.775	0.717/0.755	0.680/0.777	0.663/0.730
	LD	0.560/0.606	0.568/0.587	0.617/0.670	0.651/0.677	0.665/0.686	0.594/0.668
	LT	0.487/0.522	0.512/0.506	0.487/0.466	0.523/0.499	0.589/0.696	0.500/0.521

Table 4.19: The median values of MCC for each context feature, each extended context feature, COMB, each indentation feature, the change features and each of the change features. Each cell indicates the values of MCC by RF (left) and LR (right) models, respectively.

Feature Types	**Features**	**Projects**					
		Bitcoin	**Camel**	**Gerrit**	**Gimp**	**Hadoop**	**Osmand**
Context	NCW(c,1,+)	0.178/0.279	0.115/0.171	0.212/0.323	0.172/0.235	0.200/0.348	0.228/0.404
	NCKW(c,10,+)	0.205/0.333	0.111/0.170	0.181/0.284	0.230/0.244	0.243/0.329	0.193/0.366
	NCCW(c,1,+)	0.229/0.343	0.167/0.309	0.253/0.408	0.197/0.308	0.189/0.402	0.185/0.373
	NCCKW(c,10,+)	0.249/0.346	0.155/0.251	0.279/0.343	0.251/0.286	0.263/0.388	0.274/0.370
	COMB	0.385/0.330	0.298/0.326	0.397/0.401	0.305/0.308	0.381/0.402	0.390/0.366
Indentation	AS	0.302/0.398	0.183/0.286	0.311/0.385	0.194/0.264	0.235/0.378	0.198/0.336
	AB	0.268/0.377	0.173/0.277	0.293/0.354	0.231/0.278	0.213/0.366	0.177/0.349
Traditional	Changes	0.257/0.186	0.207/0.185	0.288/0.245	0.189/0.094	0.309/0.362	0.223/0.134
	NS	0.000/0.112	0.111/0.111	0.181/0.202	0.121/0.117	0.097/0.074	0.062/0.112
	ND	0.175/0.175	0.231/0.204	0.255/0.262	0.138/0.118	0.254/0.315	0.203/0.267
	NF	0.156/0.248	0.235/0.234	0.288/0.299	0.211/0.240	0.313/0.336	0.262/0.265
	Entropy	0.168/0.209	0.130/0.193	0.228/0.291	0.135/0.169	0.166/0.321	0.215/0.239
	LA	0.296/0.277	0.214/0.269	0.281/0.345	0.263/0.279	0.226/0.363	0.245/0.329
	LD	0.132/0.220	0.081/0.143	0.168/0.239	0.183/0.170	0.236/0.233	0.142/0.282
	LT	0.086/0.074	0.065/0.049	0.049/0.073	0.071/0.000	0.125/0.288	0.073/0.068

Table 4.20: The median values of Brier score for each context feature, each extended context feature, COMB, each indentation feature, the change features and each of the change features. Each cell indicates the values of Brier score by RF (left) and LR (right) models, respectively.

Feature Types	**Features**	**Projects**					
		Bitcoin	**Camel**	**Gerrit**	**Gimp**	**Hadoop**	**Osmand**
Context	NCW(c,1,+)	0.278/0.215	0.295/0.239	0.279/0.221	0.239/0.234	0.274/0.217	0.321/0.237
	NCKW(c,10,+)	0.270/0.216	0.281/0.241	0.285/0.224	0.234/0.235	0.255/0.225	0.298/0.239
	NCCW(c,1,+)	0.257/0.200	0.277/0.223	0.250/0.208	0.236/0.220	0.294/0.210	0.295/0.236
	NCCKW(c,10,+)	0.245/0.203	0.268/0.230	0.262/0.213	0.227/0.231	0.260/0.219	0.289/0.239
	COMB	0.225/0.210	0.285/0.224	0.221/0.209	0.190/0.226	0.245/0.226	0.237/0.239
Indentation	AS	0.228/0.198	0.258/0.226	0.233/0.210	0.246/0.229	0.279/0.220	0.295/0.239
	AB	0.238/0.214	0.264/0.227	0.250/0.210	0.240/0.250	0.283/0.218	0.294/0.240
Traditional	Changes	0.201/0.202	0.186/0.192	0.207/0.215	0.169/0.171	0.189/0.194	0.239/0.246
	NS	0.276/0.248	0.273/0.242	0.360/0.236	0.211/0.199	0.268/0.248	0.289/0.249
	ND	0.275/0.243	0.289/0.232	0.309/0.229	0.236/0.216	0.279/0.221	0.343/0.243
	NF	0.314/0.234	0.295/0.232	0.295/0.236	0.230/0.235	0.257/0.227	0.337/0.243
	Entropy	0.273/0.231	0.275/0.223	0.282/0.220	0.221/0.214	0.288/0.217	0.337/0.234
	LA	0.226/0.249	0.258/0.228	0.243/0.224	0.211/0.235	0.266/0.239	0.284/0.244
	LD	0.311/0.249	0.288/0.247	0.298/0.246	0.235/0.246	0.258/0.246	0.329/0.247
	LT	0.363/0.248	0.374/0.247	0.383/0.250	0.359/0.247	0.320/0.206	0.383/0.250

Table 4.21: Spearman rank correlation between NCCW and modified NCCKWs in the studied projects. We average correlations in the studied projects. Each cell shows the average correlation. "*" refers to that at least one non statistical significant correlation in the studied projects. "leave" statement has 'nan' for all cells. This is because that there does not exist "leave" statement in the Gerrit project. Spearman rank correlation needs to compute the variance for the denominator of its formulation. In this case, all values are zero, and therefore, variance is also zero. Hence, we got the zero division error, and the value is 'nan'. We observed that the correlation values are not strong in the other projects for "leave" statement.

	NCCW	break	case	catch	continue	default	do	else	except	for	goto	finally	if	exists	not	leave	return	switch	throw	try	while
NCCW	1.000	0.371	0.374	0.306	0.261	0.455	0.313	0.560	0.061	0.590	0.076*	0.197*	0.719	0.225	0.456	nan	0.709	0.259*	0.316	0.368	0.349
break		1.000	0.548	0.188	0.257	0.342	0.207	0.340	0.040*	0.349	0.088*	0.112*	0.397	0.127	0.241	nan*	0.338	0.505	0.184	0.208	0.340
case			1.000	0.159	0.172	0.352	0.187	0.309	0.038*	0.300	0.076*	0.118	0.341	0.127	0.246	nan*	0.339	0.597	0.182	0.181	0.221
catch				1.000	0.186	0.194	0.186*	0.247	0.038*	0.272	0.023*	0.282	0.310	0.194	0.293	nan*	0.299	0.126	0.443	0.596	0.251
continue					1.000	0.171	0.167	0.256	0.027*	0.315	0.054*	0.110*	0.318	0.135	0.214	nan*	0.238	0.139	0.171	0.219	0.265
default						1.000	0.228	0.328	0.069*	0.353	0.051*	0.126*	0.389	0.136	0.306	nan	0.373	0.305	0.207	0.232	0.211
do							1.000	0.249	0.052*	0.268	0.077*	0.154	0.299	0.142	0.280	nan*	0.283	0.149	0.175	0.225	0.243
else								1.000	0.042*	0.439	0.075*	0.159*	0.718	0.188	0.347	nan	0.527	0.232	0.261	0.286	0.319
except									1.000	0.066	0.035*	0.083*	0.050*	0.065*	0.066	nan*	0.053*	0.033*	0.045*	0.048*	0.048*
for										1.000	0.054*	0.181*	0.583	0.187	0.377	nan	0.489	0.242	0.277	0.326	0.334
goto											1.000	0.016*	0.083*	0.037*	0.066*	nan*	0.068*	0.078*	0.022*	0.032*	0.070*
finally												1.000	0.201*	0.166*	0.186	nan*	0.175*	0.085*	0.212	0.310	0.170
if													1.000	0.239	0.427	nan	0.698	0.265	0.339	0.369	0.381
exists														1.000	0.236	nan*	0.209	0.080	0.167*	0.234	0.175
not															1.000	nan	0.396	0.176	0.326	0.340	0.241
leave																nan	nan	nan*	nan*	nan	nan*
return																	1.000	0.252*	0.305	0.345	0.332
switch																		1.000	0.139	0.139	0.184
throw																			1.000	0.354	0.216
try																				1.000	0.277
while																					1.000

Chapter 5

Deep Learning Based Just-In-Time Defect Prediction with Semantic Properties as Features

An earlier version of this chapter is published in the IPSJ Journal [84]. In 2019, a related study is published [62].

5.1 Introduction

In recent years, machine learning models are frequently used in defect prediction [7,66,76,171,176,187] since there are many high-quality tools to implement machine learning models [2, 65, 126] and various data [73, 141]. In particular, deep learning models attract attention from researchers and practitioners in defect prediction since deep learning models are successful in several research fields including one in software engineering [31,78,87,91,171,176,183].

Indeed, some of the studies used deep learning models as a defect prediction model [91, 171, 176]. Yang et al. [176] applied a deep learning model called *deep belief network (DBN)* to the change features [71]. Wang et al. [171] applied a DBN to source code that were decomposed into an abstract syntax tree (AST). Li et al. [91] applied a convolutional neural network (CNN) to the source code.

However, to the best of our knowledge, by 2018, nobody applies any deep learning models to *source code changes*, which are changed source code on a commit, in *change-level defect prediction (a.k.a. just-in-time defect prediction)*. As we described in Chapter 4, just-in-time defect prediction provides several key advantages [71] such as faster feedback compared to other defect prediction. In addition, we suppose that applying a deep learning model to the source code changes may automatically retrieve additional information that is not investigated so far such as our context features (Chapter 4) and we do not need to consider keeping the number of features low (Chapter 3).

In this chapter, we applied a deep learning model to the source code changes of a commit to identify if such a commit is defective or not in just-in-time defect prediction. In particular, we used a CNN as our defect prediction model. CNNs are known as one of the successful deep learning models in image recognition; however, CNNs are successful in text classification and defect prediction as well. We consider the source code changes as texts to apply a CNN. We call our defect prediction model based on a CNN with the source code changes *Word-CNN (W-CNN)*. When constructing a CNN, we took Kim's CNN [78] into account. We conducted our experiments on seven open source projects in Java and C++.

To evaluate W-CNN, we built the following three research questions:

RQ1: **Can W-CNN be trained by source code changes?**

Motivation: Nobody applies a deep learning model to the source code changes directly so far. In this RQ, we studied can our W-CNN be trained by the source code changes.
Results: Our W-CNN can be trained by the source code changes. In addition, W-CNN can predict defective commits.

RQ2: **Does W-CNN improve defect prediction performance compared to a previous deep learning defect prediction model?**
Motivation: Yang et al. [176] have already identified defective commits with a deep learning model called *Deeper*. They used a DBN [60] with the change features of commits. Hence, we compared our W-CNN and Deeper.
Results: W-CNN outperformed Deeper in terms of the defect prediction performance for all studied projects.

RQ3: **How long do we need to train and use W-CNN?**
Motivation: A deep learning model needs longer training time compared to other machine learning models such as a logistic regression model. In this RQ, we studied the time cost of W-CNN.
Results: W-CNN needs longer time to train its model compared to Deeper. In contrast, W-CNN needs around 0.001 seconds to predict if a commit is defective. Hence, the time cost to use W-CNN is low.

Some of the key contributions of this chapter are as follows:

(1) We found that W-CNN can be used to predict defective commits in just-in-time defect prediction.

(2) We observed that W-CNN needs longer time to train its model while the prediction performance is better compared to a previous deep learning defect prediction model.

5.2 Word-convolutional neural network (W-CNN)

CNNs are one of the successful algorithms in image classification [87]. In addition, researchers recently reported that CNNs perform well in text classification as well [31,78,183]. In this chapter, we used a CNN as a text classification model to identify defective commits in which we consider source code changes as texts.

In general, a deep learning model gains the ability to learn the training data according to the number of layers. Hence, prior studies have used many layers for their model. For example, Zhang et al. [183] used a nine-layer CNN as a text classification model.

However, the model with many layers needs more training data. Indeed, Zhang et al. trained their CNN with around one million training data. However, our target data (a software repository) include around ten thousand commits if such a repository is relatively large.

Hence, we took a relatively small CNN model that was proposed by Kim [78] into account to construct our just-in-time defect prediction model. This CNN model has three hidden layers and can be trained by around 4,000-10,000 training data.

Figure 5.1 depicts an overview of our proposed W-CNN. W-CNN consists of two phases: *"Preprocessing"* where we extract the source code changes from the commits of a repository and *"Defect Prediction by CNN"* where we retrieve features from the source code changes and classify the commits into defective or not. We describe the procedure as follows:

(1) Preprocessing: For each commit, we extract added source code, source code that surrounds added/deleted source code, and file names as our *source code changes*. The source code changes are converted into a set of *text elements* with a *lightweight source code preprocessor (lscp)* [163]. Such text elements are mapped to numerical values to convert each commit into a *numerical feature vector*.

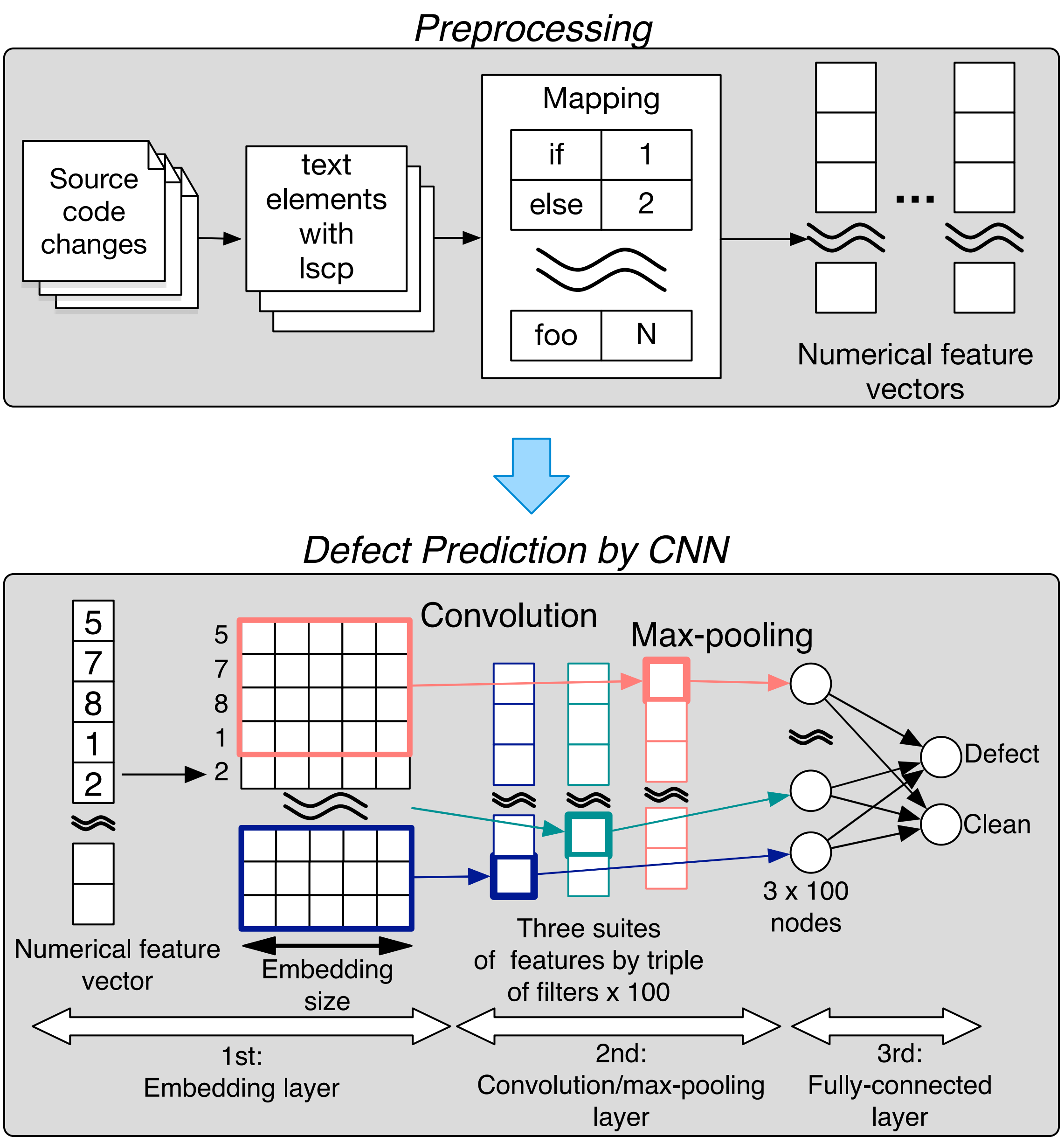


Figure 5.1: An overview of proposed defect prediction W-CNN.

(2) Defect Prediction by CNN: We use the numerical feature vectors to train the CNN and identify defective commits with the CNN.

5.2.1 Preprocessing

Creating source code changes

We extracted the source code changes from source files[*] only. This is because changes in non-source files (e.g., document files) may not induce any defects. As we described above, the source code changes include added source code and source code that surrounds added/deleted source code (*context lines*). The reason why we include the context lines is that we hypothesize that the context lines can make source code changes more informative to be processed by a text classification model. The number of context lines is three in this chapter. We discussed this parameter in Section 5.5.1. Then, we excluded comment lines. Finally, we grouped all the changes for each file and added its file name at the top of the group of the changes. We call them *source code changes*. Figure 5.2 shows an example of source code changes from two commits in the Camel project. The red lines indicate file names; the other lines indicate changes.

Split source code changes into text elements with lscp

We have to convert input texts to words for applying the CNN [78]. Hence, we need to split source code changes into text elements (i.e., tokens). To do this, we used *lscp* that was proposed by Thomas [163]. lscp splits source code with heuristics and does not use any parsers. Hence, we can apply lscp to a part of source code such as our source code changes. lscp needs some parameters to split texts. We described our parameter setting as follows:

- The target data is source code

[*]Here, a source file indicates a file with an extension of java, c, h, cpp, hpp, cxx, and hxx. We studied C++ and Java only.

[Commit: d80f93cca26f82126f408179fdc8c3c6c1ccbc7f Source Code Change]

components/camel-kafka/src/main/java/org/apache/camel/component/kafka/KafkaConfiguration.java } public String getSaslMechanism() { return saslMechanism; } public void setSaslMechanism(String saslMechanism) { this.saslMechanism = saslMechanism; } public String getSecurityProtocol() { return securityProtocol; }

[Commit: 2645cc184f549da4c2ce398a8ea9704927524b2e A Part of Source Code Change]

components/camel-kafka/src/main/java/org/apache/camel/component/kafka/KafkaConfiguration.java private Integer reconnectBackoffMs = 50; @UriParam(label = "common", defaultValue = SaslConfigs.DEFAULT_SASL_MECHANISM) private String saslMechanism = SaslConfigs.DEFAULT_SASL_MECHANISM; @UriParam(label = "common", defaultValue = SaslConfigs.DEFAULT_KERBEROS_KINIT_CMD) private String kerberosInitCmd = SaslConfigs.DEFAULT_KERBEROS_KINIT_CMD; @UriParam(label = "common", defaultValue = "60000") private Integer kerberosBeforeReloginMinTime = 60000; @UriParam(label = "common", defaultValue = "0.05") private Double kerberosRenewJitter = SaslConfigs.DEFAULT_KERBEROS_TICKET_RENEW_JITTER;

Figure 5.2: An example of source code changes in the Camel project.

- Extract identifiers
- Remove comments
- Remove numerical values
- Make all source code lower case
- Do not apply a stemming analysis
- Split identifiers if such identifiers are compound words such as camel case
- Remove punctuations
- Remove small tokens that consist of only one character

Table 5.1: Statistical values of the number of tokens in the source code changes of a commit for each project.

Project	25th percentile	50th percentile	75th percentile	90th percentile	Max
Hadoop	149.25	461.00	1,287.75	3,145.10	224,389.00
Camel	112.00	281.00	678.00	1,454.00	42,806.00
Gerrit	84.00	216.00	604.25	1,406.00	53,302.00
Osmand	70.00	183.00	607.00	1,716.20	324,540.00
CMake	34.00	79.00	224.00	586.00	236,896.00
Bitcoin	50.00	128.00	334.00	851.10	102,844.00
Gimp	64.00	200.00	647.25	1,939.00	1,954,189.00

- Do not remove stop words
- Do not remove programming language keywords

CNNs have the restriction that all input data must be equal. Because our input data are tokens of source code changes for each commit, we need to make the number of tokens a fixed size for all commits. In our experiment, we fixed the number of tokens to 4,000. If the number of tokens is below 4,000, we pad it with zeros. Otherwise, we removed all tokens beyond the first 4,000 tokens.

Padding zeros is a common process while removing tokens is not. In our target data, the number of tokens is very skewed. If we used the maximum number of tokens, our input data would include many zeros and make the processing cost gigantic. Our preliminary analysis showed that the 90 percentile of the number of tokens in a commit is less than 4,000 for all the projects (Table 5.1). Hence, this number can cover at least 90% commits for all projects.

Mapping and vectorization

Because CNNs can process numerical input data only, we need to map all tokens on numerical values. We sorted all tokens in descending order in terms of the appearances. Then we map tokens on numerical values from one. We call the association between numerical values and tokens a *mapping table*. A mapping table was built for each training data. Note that a token that does not appear in the mapping table is assigned to zero. Hence, a commit was converted into a one-dimensional *numerical feature vector*.

5.2.2 Defect prediction by CNN

We apply the CNN of W-CNN to the numerical feature vectors that were retrieved from the preprocessing phase. The CNN conducts (1) extracting features from the numerical feature vectors, and (2) identifying defective commits. Our three-layer CNN in W-CNN extracts features and identifies defective commits as follows:

The first layer is an embedding layer that learns an embedded representation of the tokens based on the numerical feature vectors. An embedded representation refers to a numerical vector that represents a token/word [104, 127]. Kim's CNN used word2vec [104] instead of an embedding layer to gain embedded representations. However, word2vec needs numerous data to train its model. Kim used a text corpus based on Google News that contains 100 billion words. In contrast, our target data is source code, not natural language; and therefore, using word2vec with the same corpus does not work well on our data. In addition, it is difficult to prepare the training data that contains 100 billion tokens from source code instead of words from natural language. Therefore, we used an embedding layer to make an embedded representation for each token. Because the embedding layer is a part of CNN, we can train the embedding layer as a part of CNN. This layer prepared a lookup table that converts a token into a 128-dimensional numerical vector. This 128 is the *embedding size* of the embedding layer.

The second layer is a convolution/max-pooling layer. The convolution layer trains filters to extract more informative features from the embedded representations called *feature maps*. W-CNN used filters that can learn some numerical feature vectors (some tokens) at once to consider *n*-grams [17]. Hence, we prepared three variants of filters that have a width of 128 and heights of 3, 4, and, 5 (three *n*-grams). The number of each of the variants of filters is 100. The number of outputs of the convolutional layer is 300 (100 filters with 3 variants) feature maps.

The max-pooling layer selects the important features from the extracted feature maps. We used max-pooling that selects the maximum values as important features for each feature map. Hence, the max-pooling provided 300 features as nodes.

The third-layer is a fully-connected layer between 300 features (nodes) and 2 outputs that correspond to the probabilities of defective and clean commits. We used the softmax function [121] to calculate the probabilities.

5.2.3 Hyper-parameters of CNN

We chose the same hyper-parameters as Kim's CNN. However, we modified some of them to improve defect prediction performance and the cost of training the CNN. We used *adaptive moment estimation (Adam)* [79] as a gradient descent optimization algorithm. This is because Adam makes the convergence faster than the probabilistic gradient descent. We applied a simple regularization called l_2-norm to reduce the cost of implementation. The mini-batch size is 50.

5.2.4 The difference with the previous defect prediction model

In 2018, this was the first study to apply a deep learning model to the source code changes in just-in-time defect prediction (W-CNN). However, Yang et al. [176] have already applied a deep learning model to the change features of commits in just-in-time defect prediction (Deeper). Hence, we clarify the key differences between our W-CNN and Deeper.

- *Deep Learning Model:* W-CNN uses a CNN while Deeper uses a DBN.
- *Classification:* W-CNN uses a CNN to identify defective commits while Deeper uses a logistic regression model with features that were generated

by a DBN.

- *Input Data:* W-CNN uses the source code changes while Deeper uses the change features.

5.3 Experimental design

Our experiment compared W-CNN and Deeper [176]. Figure 5.3 shows an overview of our experimental design. The main steps of our experiment are as follows:

1. Extract commits from the studied repositories and identify defective commits by Commit Guru [141].
2. Extract the source code changes and the change features from all the commits for W-CNN and Deeper respectively.
3. Apply a *resampling approach* to the commits of the training data.
4. Evaluate W-CNN and Deeper in terms of the defect prediction performance using *10-times 10-fold cross-validation*.

We described the details of the experimental design in the following of this section.

5.3.1 Preparing studied data by Commit Guru

Similar to Chapter 4, we used Commit Guru to compute the change features and identify defective commits. We used the change features as the input data for Deeper and the identified defective commits as the correct labels for each commit.

5.3.2 Studied software projects

In this chapter, we used seven open source projects that consist of enough commits to apply the defect prediction model. Table 5.2 summarizes the details of the

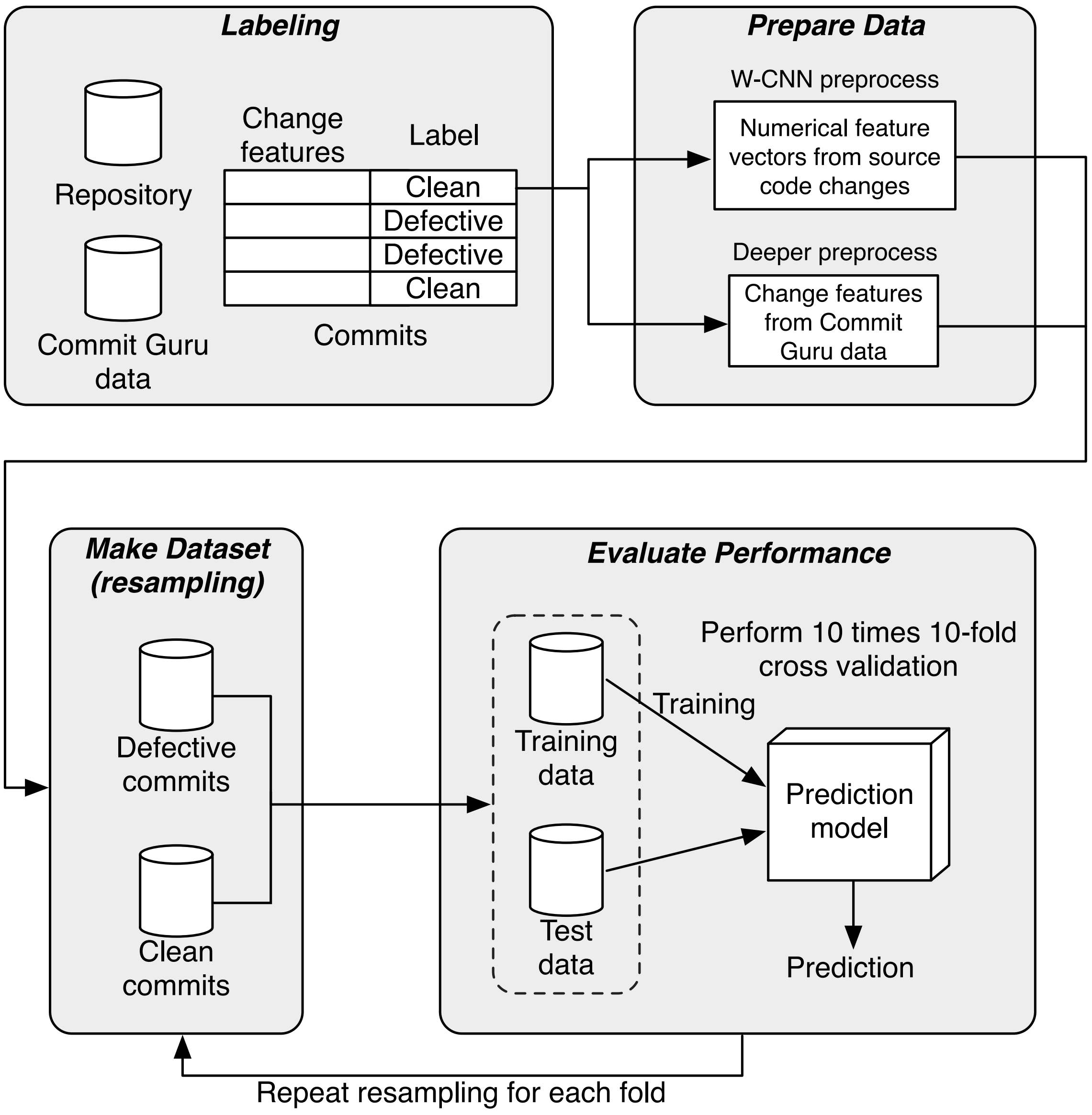


Figure 5.3: An overview of our experimental design.

Table 5.2: Subject Projects.

Project	Language	The Total Number of Commits	Defective Rate
Hadoop	Java	13,920	24.8%
Camel	Java	24,740	23.2%
Gerrit	Java	18,794	20.1%
Osmand	Java	31,366	14.0%
CMake	C++	28,400	10.1%
Bitcoin	C++	11,093	14.4%
Gimp	C++	37,116	22.5%

studied projects. Note that we did not use the same projects that were used to evaluate Deeper in the prior study [176]. This is because Commit Guru can be applied to Git repositories; however, the studied projects in the prior study are not managed by Git.

5.3.3 Deeper

Deeper [176] uses a DBN to extract more informative features from the original change features. The features were used to build a logistic regression model that identifies defective commits. Yang et al. [176] did not provide the implementation of Deeper, and therefore, we used our implementation of Deeper. The configuration of Deeper is as follows:

- It has three hidden layers.
- The input layer has 14 nodes; the output layers has 2 nodes; the number of nodes for the three hidden layers are 20, 12, and 12.
- The mini-batch size is 100. Hence the network was trained by 100 data on each mini-batch data.

In addition, we normalized all the change features to a range between 0 and 1 as follows:

$$\mathbf{X}_{0-1} = \frac{\mathbf{X}_{org} - X_{\min}}{X_{\max} - X_{\min}} \tag{5.1}$$

where $\mathbf{X}_{org}$ is a vector of all values of a feature. $X_{\min}$ is the least value and $X_{\max}$ is the maximum value of the given feature.

Finally, we need to decide the number of learning iterations since Yang et al. did not provide that. We plotted the values of the test loss and decided on 50 as the learning iterations.

5.3.4 Resampling

As we described in Chapter 4, the imbalanced data may affect the prediction performance [155]. In this experiment, Table 5.2 shows that the defective rates are around 10–20%. Hence we applied a resampling approach. In this chapter, we used the random under-sampling that we used in Chapter 4 as well.

5.3.5 10-times 10-fold cross validation

In our experiments, we split the commits at random into a training and a test data. To reduce a bias of this random data selection, we used 10-fold cross-validation 10 times on all data. This follows the existing practice in previous defect prediction approaches [160, 176]. As we described in the Section 4.4.3, using 10-fold cross validation may use future commits to predict past commits. However, we use neither the time sensitive change classification nor the online change classification (Section 4.4.3 and Section 4.4.4). This is because one of the aims of this chapter is to investigate if a deep learning model can learn the features of the source code changes of defective commits. If we used either the time sensitive change classification or the online change classification, mislabelled commits would exist in the training data so that it would be biased to investigate the ability to learn the features of source code changes.

5.3.6 Evaluation measures

To measure the prediction efficacy, we used the area under the receiver operation characteristic curve (AUC) similar to the previous chapters.

5.4 Results

5.4.1 RQ1: Can W-CNN be trained by source code changes?

Motivation: By 2018, researchers have not applied a deep learning model to the source code changes. Hence, we first need to figure out whether W-CNN is able to identify defective commits or not.

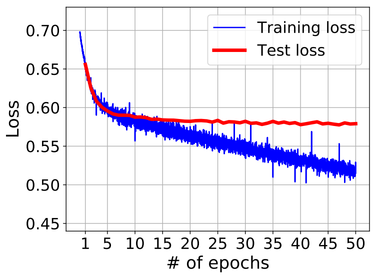
To address this RQ, we tuned the number of training iterations.[†] Usually, the larger number of iterations, the tighter W-CNN fits to the training data. However, a poorly chosen number of iterations causes overfitting or underfitting [107]. Overfitting occurs if the model is too specific for training data, and therefore, the performance for test data is decreased. Underfitting occurs if the model is not adapted well enough.

Approach: To do this, we plotted *training loss* and *test loss*. Training loss is a value computed by the loss function on training data. Test loss is a value computed by the loss function on test data. In the experiment, we used the cross-entropy as the loss function [121]. We plotted both training loss and test loss by the epoch, with up to 50 epochs, and investigated the training loss and the test loss. If the number of epochs increases but the test loss is either equivalent or increases, we determined that the model is overfitting.

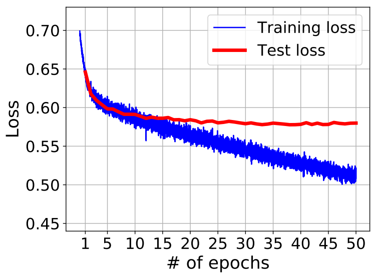
In addition, we computed the average AUC values across the studied projects by five epochs. The AUC value for each project is the average within the project on 10 times 10-fold cross-validation.

Results: **The training loss and the test loss continuously decreased up to**

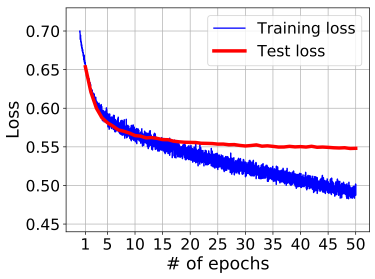
[†]In our experiment, the number of training iterations is counted by the epoch.



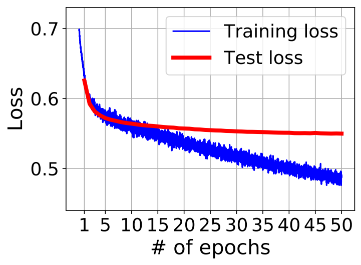
(a) Hadoop



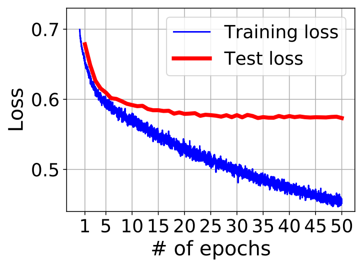
(b) Camel



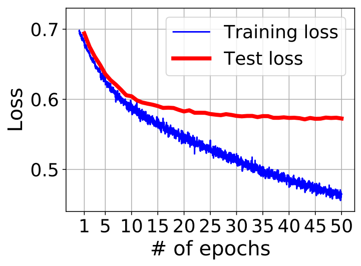
(c) Gerrit



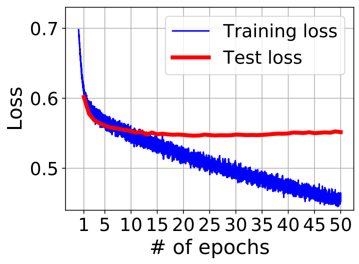
(d) Osmand



(e) CMake



(f) Bitcoin



(g) Gimp

Figure 5.4: The training loss and the test loss for each project. The blue lines refer to the training loss; the red lines refer to the test loss. The x-axis indicates the number of epochs.

Table 5.3: Average AUC values across the studied projects by five epochs.

# epochs	5	10	15	20	25	30	35	40	45	50
AUC	0.788	0.796	0.802	0.807	0.810	0.812	0.813	0.814	0.815	0.815

about 50 epochs. Figure 5.4 shows the training loss and the test loss for each project. We observed that the training loss and the test loss decreased up to about 50 epochs. Hence, W-CNN can be trained by the source code changes. The test loss decreased rapidly until about epoch 5 (Hadoop, Camel, Osmand, Gimp) to 10 (Gerrit, CMake, Bitcoin). After these epochs, the test loss decreased slightly. On the other hand, the training loss continuously decreased after these epochs. Hence, W-CNN can be trained well after 5–10 epochs; after that, overfitting may occur.

AUC increased while increasing the epoch until about 50. W-CNN resulted in an average AUC value of 0.815 in the end. Table 5.3 shows the average AUC values across the studied projects by five epochs. While increasing the epoch, the AUC value also increased. On the epoch 50, the AUC value resulted in 0.815. However, the differences between AUC values are small for a large number of epochs. Indeed, the difference between epoch 45 and 50 is almost zero. Hence, we determined 50 epochs to be optimal for training W-CNN.

5.4.2 RQ2: Does W-CNN improve defect prediction performance compared to a previous deep learning defect prediction model?

Motivation: In this RQ, we compared the prediction performance between W-CNN and Deeper to clarify whether a deep learning model with the source code changes improves the prediction performance or not.

Approach: We compared W-CNN and Deeper in terms of AUC. We computed the average of AUC on 10 times 10-fold cross-validation for each project.

Table 5.4: Average AUC values for each project by W-CNN and Deeper.

Project	**W-CNN**	**Deeper**
Bitcoin	0.810	0.666
Camel	0.796	0.642
CMake	0.819	0.656
Gerrit	0.830	0.724
Gimp	0.828	0.645
Hadoop	0.800	0.660
Osmand	0.823	0.696

Results: **W-CNN achieved better prediction performance compared to Deeper for all the projects.** Table 5.4 shows the average AUC values on 10 times 10-fold cross-validation for each project. Therefore, we concluded that a deep learning model with the source code changes can improve the prediction performance compared to a deep learning model with the change features.

5.4.3 RQ3: How long do we need to train and use W-CNN?

Motivation: Compared to other machine learning models such as logistic regression, deep learning models need a longer training time. The training time depends on the size of its architecture. To evaluate the efficiency of W-CNN, we investigated the time cost to train and use W-CNN.

Approach: The training time is the time to train the model. For W-CNN and Deeper, the training time was measured by the time from the beginning to the end of the training for one fold on cross-validation. For example, W-CNN was trained for 50 epochs in one fold training. Hence, the training time is the time to run 50 epochs. We defined the test time as the time to identify a commit that is averaged across all commits in the test data. The training time and the test time were measured on 10 times 10-fold cross-validation and considered the average

Table 5.5: The training and test times in seconds.

Project	**W-CNN**	**Deeper**
Bitcoin	571.604 (0.001)	12.446 (0.001)
Camel	1880.006 (0.001)	45.199 (0.001)
CMake	1016.564 (0.001)	23.269 (0.001)
Gerrit	1262.085 (0.001)	27.908 (0.001)
Gimp	2728.318 (0.001)	63.976 (0.001)
Hadoop	1114.774 (0.001)	26.499 (0.001)
Osmand	1423.162 (0.001)	34.149 (0.001)

value of the cross-validation. We ran the experiment on an Intel Xeon CPU E5-1620 v3 @ 3.50 GHz, NVIDIA GeForce GTX TITAN X (3584 cuda cores, 12 GB of RAM).

Results: **W-CNN required longer training times, however, the test times were as short as with Deeper.** Table 5.5 shows the training times and the test times of W-CNN and Deeper. The left values indicate the training times; the right values, surrounded by parentheses, are the test times. We observed that W-CNN required longer training times. However, the longest training time we observed was approximately 2,728 seconds (45 minutes). In addition, the test times were as short as Deeper. Hence, the time cost of W-CNN is acceptable for a practical defect prediction.

5.5 Threats to validity

5.5.1 Construct validity

We decided the number of context lines to define the source code changes in W-CNN. In our study, we used three lines, which is the default context of the `git show` command. Considering a different number of context lines may further improve the efficacy depending on projects. As future work, we will investigate the optimal number of context lines.

In the experiment, we used AUC as our evaluation measure. As we described in Section 3.6, we will investigate other performance measures.

We used the cross-validation to eliminates bias from a fixed selection of training and test data. However, the cross-validation may use future commits to predict past commits. One of the aims of this chapter is to investigate if a deep learning model can learn the features of the source code changes of defective commits. Hence, the cross-validation is acceptable in this chapter. However, future studies are necessary to investigate the performance of W-CNN on the online change classification (Section 4.4.4) to apply W-CNN to a practical scenario.

5.5.2 External validity

In this experiment, we used seven large open source projects as our data. The subject projects are in C++ and Java and cover various domains, such as servers, web applications, etc. However, our selection does not cover every type of software, and therefore our results may not be generalizable across domains. We need additional projects and commercial projects to improve robustness.

5.5.3 Internal validity

Our experiments used Commit Guru for labeling defects in commits. As we described in Section 4.8, the repeatability and openness of Commit Guru is high.

5.6 Chapter summary

In this chapter, we proposed a new defect prediction approach, W-CNN, which is based on a convolutional neural network that was proposed by Kim [78]. To the best of our knowledge, by 2018, this was the first study that applied a deep learning model to the source code changes in just-in-time defect prediction. The results of our experiments that compared W-CNN and the previous deep learning defect prediction approach, Deeper [176], on seven large open source software projects showed the following:

- W-CNN can predict defective commits effectively, and provided better prediction performance on AUC compared to Deeper.
- While W-CNN took longer training times than Deeper, its test times were very short.

Chapter 6

Conclusion and Future Work

6.1 Summary of this thesis

Feature engineering is pivotal in software defect prediction. In particular, keeping the number of features small by using feature reduction and selection techniques and proposing new features based on domain knowledge have been frequently studied so far. However, three remaining challenges exist: (1) a large-scale comparison of feature reduction techniques, (2) using the context lines of source code as features in just-in-time defect prediction, and (3) using semantic properties as features by a deep learning model in just-in-time defect prediction.

In this thesis, we investigated these remaining challenges by empirical studies. We summarized the main findings and implications of this thesis as follows:

6.1.1 A large-scale comparison of feature reduction techniques (Chapter 3)

Finding 1: Neural network-based feature reduction techniques (i.e., RBM and AE) significantly improved the performance of unsupervised defect prediction models compared to the other feature reduction/selection techniques.

Implication: If practitioners need a defect prediction model, but, their projects do not have enough data to train a supervised defect prediction model (e.g., the beginning of the project), we recommend using an unsupervised defect prediction model with a neural network-based feature reduction technique.

Finding 2: The best-performing feature selection techniques in the prior studies (i.e., CFS and ConFS) outperformed the feature reduction techniques except FA on supervised defect prediction models. While FA has a similar performance, CFS and ConFS have a smaller performance variance.

Implication: If practitioners have enough data to train a supervised defect prediction model (e.g., the middle or end of the project), we recommend using a supervised defect prediction model with a feature selection technique.

6.1.2 Using the context lines of source code as features (Chapter 4)

Finding 3: Our proposed combination features that consist of two extended context features significantly outperformed all studied features and achieved the best-performing features in all the studied projects in two of three evaluation measures.

Implication: Future studies should consider not only the changed lines but also the context lines in defect prediction.

Finding 4: "goto" statement in the context lines and the changed lines is the best keyword to detect defective commits.

Implication: Considering the semantic properties may improve the accuracy to identify defective commits. Hence, we recommend considering semantic properties when proposing new features.

6.1.3 Using semantic properties as features by a deep learning model in just-in-time defect prediction (Chapter 5)

Finding 5: Our proposed W-CNN that is based on an existing lightweight CNN with the source code changes can predict defective commits in just-in-time defect prediction. In addition, W-CNN outperformed a deep learning model with the change features, Deeper.

Implication: Hence, we concluded that a deep learning model with semantic properties can identify defective commits accurately.

Finding 6: While W-CNN needed longer training time than Deeper, its test time was very short. In addition, the training time was also not too long.

Implication: Hence, the time cost of W-CNN is acceptable in a practical scenario. Especially, the prediction time is not different with traditional defect prediction models.

6.2 Future work

From our findings, we outline future research directions.

6.2.1 Investigating semantic properties and deep learning models more deeply to improve the interpretability of defect prediction models

We found that the semantic properties (i.e., the context lines and the source code changes) and a deep learning model (CNN) contribute to the defect prediction performance in Chapter 4 and Chapter 5. While we found that "goto" statement may improve the prediction performance, it is still difficult to explain how our prediction models identify defective commits and justify our prediction models. Lack of interpretability is a severe challenge to apply such models to practical situations and make decisions in software development. Hence, future studies are necessary to investigate semantic properties and deep learning models to improve the interpretability of such prediction models.

6.2.2 Comparing W-CNN with other text classification models in practical scenarios

In Chapter 5, we proposed W-CNN and compared W-CNN with Deeper. From the results, we found that a deep learning model with the source code changes improves the defect prediction performance. However, this comparison is small, and therefore, we need to extend the comparison to comparing W-CNN with other text classification models as well. In addition, we used cross-validation to evaluate W-CNN. To apply W-CNN to practical situations, we need to evaluate W-CNN in practical scenarios as well. Hence, future studies are necessary to compare W-CNN with other text classification models in practical scenarios.

6.2.3 Evaluating our approaches in industrial development

In this thesis, we conducted some experiments. However, the studied data of our experiments are restricted to open source projects. Our ultimate goal is to support not only open source development but also industrial development. Hence, future studies are necessary to evaluate our approaches in industrial development.

Bibliography

[1] Scikit learn: Minmaxscaler. http://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.MinMaxScaler.html. [Online; accessed 2018-7-1].

[2] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mane, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viegas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv:1603.04467*, 2016.

[3] A. Agrawal and T. Menzies. Is "better data" better than "better data miners"? In *Proceedings of the 40th International Conference on Software Engineering (ICSE)*, pages 1050–1061. IEEE, 2018.

[4] A. Alin. Multicollinearity. *Wiley Interdisciplinary Reviews: Computational Statistics*, 2(3):370–374, 2010.

[5] B. S. Alqadi and J. I. Maletic. Slice-based cognitive complexity metrics for defect prediction. In *Proceedings of the 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 411–422. IEEE, 2020.

[6] S. Amasaki. Cross-version defect prediction: use historical data, cross-project data, or both? *Empirical Software Engineering*, 25:1573–1595, 2020.

[7] L. Aversano, L. Cerulo, and C. Del Grosso. Learning from bug-introducing changes to prevent fault prone code. In *Proceedings of the 9th International Workshop on Principles of Software Evolution (IWPSE)*, pages 19–26. ACM, 2007.

[8] A. A. Bangash, H. Sahar, A. Hindle, and K. Ali. On the time-based conclusion stability of cross-project defect prediction models. *Empirical Software Engineering*, 25:5047–5083, 2020.

[9] V. R. Basili, L. C. Briand, and W. L. Melo. A validation of object-oriented design metrics as quality indicators. *IEEE Transactions on software engineering*, 22(10):751–761, 1996.

[10] R. Bellman. *Dynamic programming*. Princeton University Press, 1957.

[11] K. E. Bennin, J. Keung, P. Phannachitta, A. Monden, and S. Mensah. Mahakil: Diversity based oversampling approach to alleviate the class imbalance issue in software defect prediction. *IEEE Transactions on Software Engineering*, 44(6):534–550, 2018.

[12] K. E. Bennin, J. W. Keung, and A. Monden. On the relative value of data resampling approaches for software defect prediction. *Empirical Software Engineering*, 24:602–636, 2019.

[13] A. Bertolino. Software testing research: Achievements, challenges, dreams. In *Future of Software Engineering (FOSE '07)*, pages 85–103, 2007.

[14] E. Bingham and H. Mannila. Random projection in dimensionality reduction: applications to image and text data. In *Proceedings of the 7th International Conference on Knowledge Discovery and Data Mining*, pages 245–250. ACM, 2001.

[15] S. Boughorbel, F. Jarray, and M. El-Anbari. Optimal classifier for imbalanced data using matthews correlation coefficient metric. *PloS one*, 12(6):1–17, 2017. e0177678.

[16] D. Bowes, T. Hall, and D. Gray. Comparing the performance of fault prediction models which report multiple performance measures: recomputing the confusion matrix. In *Proceedings of the 8th International Conference on Predictive Models in Software Engineering*, pages 109–118. ACM, 2012.

[17] P. F. Brown, P. V. Desouza, R. L. Mercer, V. J. D. Pietra, and J. C. Lai. Class-based n-gram models of natural language. *Computational linguistics*, 18(4):467–479, 1992.

[18] G. G. Cabral, L. L. Minku, E. Shihab, and S. Mujahid. Class imbalance evolution and verification latency in just-in-time software defect prediction. In *Proceedings of the 41st International Conference on Software Engineering (ICSE)*, pages 666–676. IEEE, 2019.

[19] B. Caglayan, B. Turhan, A. Bener, M. Habayeb, A. Miransky, and E. Cialini. Merits of organizational metrics in defect prediction: an industrial replication. In *Proceedings of the 37th International Conference on Software Engineering (ICSE)*, pages 89–98. IEEE, 2015.

[20] V. U. B. Challagulla, F. B. Bastani, I.-L. Yen, and R. A. Paul. Empirical assessment of machine learning based software defect prediction techniques. *International Journal on Artificial Intelligence Tools*, 17(02):389–400, 2008.

[21] H. Chen, X.-Y. Jing, Z. Li, D. Wu, Y. Peng, and Z. Huang. An empirical study on heterogeneous defect prediction approaches. *IEEE Transactions on Software Engineering*, 2020. Early Access.

[22] J. Chen, K. Hu, Y. Yu, Z. Chen, Q. Xuan, Y. Liu, and V. Filkov. Software visualization and deep transfer learning for effective software defect prediction.

In *Proceedings of the 42nd International Conference on Software Engineering*, pages 578–589, 2020.

[23] D. Chicco. Ten quick tips for machine learning in computational biology. *BioData mining*, 10(35), 2017.

[24] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on software engineering*, 20(6):476–493, 1994.

[25] J. Cohen. Statistical power analysis for the behavioral sciences, 1988.

[26] D. Cui, T. Liu, Y. Cai, Q. Zheng, Q. Feng, W. Jin, J. Guo, and Y. Qu. Investigating the impact of multiple dependency structures on software defects. In *Proceedings of the 41st International Conference on Software Engineering (ICSE)*, pages 584–595. IEEE, 2019.

[27] H. K. Dam, T. Pham, S. W. Ng, T. Tran, J. Grundy, A. Ghose, T. Kim, and C.-J. Kim. Lessons learned from using a deep tree-based model for software defect prediction in practice. In *Proceedings of the 16th International Conference on Mining Software Repositories (MSR)*, pages 46–57. IEEE, 2019.

[28] M. D'Ambros, M. Lanza, and R. Robbes. An extensive comparison of bug prediction approaches. In *Proceedings of the 7th Working Conference on Mining Software Repositories (MSR)*, pages 31–41. IEEE, 2010.

[29] M. Dash and H. Liu. Consistency-based search in feature selection. *Artificial intelligence*, 151(1):155–176, 2003.

[30] D. Di Nucci, F. Palomba, G. De Rosa, G. Bavota, R. Oliveto, and A. De Lucia. A developer centered bug prediction model. *IEEE Transactions on Software Engineering*, 44(1):5–24, 2018.

[31] C. N. Dos Santos and M. Gatti. Deep convolutional neural networks for sentiment analysis of short texts. In *Proceedings of the 25th International Conference on Computational Linguistics (COLING)*, pages 69–78, 2014.

[32] J. C. Dunn. A fuzzy relative of the isodata process and its use in detecting compact well-separated clusters. *J. Cybernet*, 3:32–57, 1973.

[33] B. Eken. Assessing personalized software defect predictors. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, pages 488–491. IEEE, 2018.

[34] D. Falessi, J. Huang, L. Narayana, J. F. Thai, and B. Turhan. On the need of preserving order of data when validating within-project defect classifiers. *Empirical Software Engineering*, 25:4805–4830, 2020.

[35] C. Faloutsos and K.-I. Lin. FastMap: A fast algorithm for indexing, data-mining and visualization of traditional and multimedia datasets. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 163–174. ACM, 1995.

[36] Y. Fan, X. Xia, D. A. da Costa, D. Lo, A. E. Hassan, and S. Li. The impact of changes mislabeled by szz on just-in-time defect prediction. *IEEE Transactions on Software Engineering*, 2019. Early Access.

[37] D. E. Farrar and R. R. Glauber. Multicollinearity in regression analysis: the problem revisited. *The Review of Economic and Statistics*, 49(1):92–107, 1967.

[38] T. Fukushima, Y. Kamei, S. McIntosh, K. Yamashita, and N. Ubayashi. An empirical study of just-in-time defect prediction using cross-project models. In *Proceedings of the 11th Working Conference on Mining Software Repositories (MSR)*, pages 172–181, 2014.

[39] K. Gao, T. M. Khoshgoftaar, H. Wang, and N. Seliya. Choosing software metrics for defect prediction: an investigation on feature selection techniques. *Software: Practice and Experience*, 41(5):579–606, 2011.

[40] B. Ghotra, S. McIntosh, and A. E. Hassan. Revisiting the impact of classification techniques on the performance of defect prediction models. In

Proceedings of the 37th International Conference on Software Engineering (ICSE), pages 789–800. IEEE, 2015.

[41] B. Ghotra, S. Mcintosh, and A. E. Hassan. A large-scale study of the impact of feature selection techniques on defect classification models. In *Proceedings of the 14th International Conference on Mining Software Repositories (MSR)*, pages 146–157. IEEE, 2017.

[42] L. Gong, S. Jiang, R. Wang, and L. Jiang. Empirical evaluation of the impact of class overlap on software defect prediction. In *Proceedings of the 34th International Conference on Automated Software Engineering (ASE)*, pages 698–709. IEEE, 2019.

[43] A. R. Gray and S. G. Macdonell. Software metrics data analysis–exploring the relative performance of some commonly used modeling techniques. *Empirical Software Engineering*, 4:297–316, 1999.

[44] L. Guo, B. Cukic, and H. Singh. Predicting fault prone modules by the dempster-shafer belief networks. In *Proceedings of the 18th International Conference on Automated Software Engineering (ASE)*, pages 249–252. IEEE, 2003.

[45] 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.

[46] M. A. Hall. *Correlation-based feature selection for machine learning*. PhD thesis, University of Waikato Hamilton, 1999.

[47] M. A. Hall and G. Holmes. Benchmarking attribute selection techniques for discrete class data mining. *IEEE Transactions on Knowledge and Data Engineering*, 15(6):1437–1447, 2003.

[48] T. Hall, S. Beecham, D. Bowes, D. Gray, and S. Counsell. A systematic literature review on fault prediction performance in software engineering. *IEEE Transactions on Software Engineering*, 38(6):1276–1304, 2012.

[49] T. Hall, S. Beecham, D. Bowes, D. Gray, and S. Counsell. A systematic literature review on fault prediction performance in software engineering. *IEEE Transactions on Software Engineering*, 38(6):1276–1304, 2012.

[50] M. H. Halstead. *Elements of software science*. Elsevier Science Inc., 1977.

[51] J. Han and C. Moraga. The influence of the sigmoid function parameters on the speed of backpropagation learning. In *Proceedings of the International Workshop on Artificial Neural Networks*, pages 195–201. Springer, 1995.

[52] R. Harrison, S. J. Counsell, and R. V. Nithi. An evaluation of the mood set of object-oriented software metrics. *IEEE Transactions on Software Engineering*, 24(6):491–496, 1998.

[53] J. A. Hartigan and M. A. Wong. Algorithm as 136: A k-means clustering algorithm. *Journal of the Royal Statistical Society. Series C (Applied Statistics)*, 28(1):100–108, 1979.

[54] A. E. Hassan. Predicting faults using the complexity of code changes. In *Proceedings of the 31st International Conference on Software Engineering (ICSE)*, pages 78–88. IEEE, 2009.

[55] H. Hata, O. Mizuno, and T. Kikuno. Bug prediction based on fine-grained module histories. In *Proceedings of the 34th International Conference on Software Engineering (ICSE)*, pages 200–210. IEEE, 2012.

[56] S. Herbold. On the costs and profit of software defect prediction. *IEEE Transactions on Software Engineering*, 2019. Early Access.

[57] S. Herbold, A. Trautsch, and J. Grabowski. A comparative study to benchmark cross-project defect prediction approaches. *IEEE Transactions on Software Engineering*, 44(9):811–833, 2018.

[58] S. Herbold, A. Trautsch, and J. Grabowski. Correction of " a comparative study to benchmark cross-project defect prediction approaches " . *IEEE Transactions on Software Engineering*, 45(6):632–636, 2019.

[59] A. Hindle, M. W. Godfrey, and R. C. Holt. Reading beside the lines: Indentation as a proxy for complexity metric. In *Proceedings of the 16th International Conference on Program Comprehension (ICPC)*, pages 133–142. IEEE, 2008.

[60] G. E. Hinton and R. R. Salakhutdinov. Reducing the dimensionality of data with neural networks. *Science*, 313(5786):504–507, 2006.

[61] T. K. Ho. Random decision forests. In *Proceedings of the 3rd International Conference on Document Analysis and Recognition*, volume 1, pages 278–282. IEEE, 1995.

[62] T. Hoang, H. K. Dam, Y. Kamei, D. Lo, and N. Ubayashi. Deepjit: an end-to-end deep learning framework for just-in-time defect prediction. In *Proceedings of the 16th International Conference on Mining Software Repositories (MSR)*, pages 34–45. IEEE, 2019.

[63] S. Hosseini, B. Turhan, and D. Gunarathna. A systematic literature review and meta-analysis on cross project defect prediction. *IEEE Transactions on Software Engineering*, 45(2):111–147, 2019.

[64] Q. Huang, X. Xia, and D. Lo. Revisiting supervised and unsupervised models for effort-aware just-in-time defect prediction. *Empirical Software Engineering*, 24:2823–2862, 2019.

[65] R. Ihaka and R. Gentleman. R: a language for data analysis and graphics. *Journal of Computational and Graphical Statistics*, 5(3):299–314, 1996.

[66] T. Jiang, L. Tan, and S. Kim. Personalized defect prediction. In *Proceedings of the 28th International Conference on Automated Software Engineering (ASE)*, pages 279–289. IEEE, 2013.

[67] J. Jiarpakdee, C. Tantithamthavorn, H. K. Dam, and J. Grundy. An empirical study of model-agnostic techniques for defect prediction models. *IEEE Transactions on Software Engineering*, 2020. Early Access.

[68] J. Jiarpakdee, C. Tantithamthavorn, and C. Treude. The impact of automated feature selection techniques on the interpretation of defect models. *Empirical Software Engineering*, 25:3590–3638, 2020.

[69] M. Jureczko and L. Madeyski. Towards identifying software project clusters with regard to defect prediction. In *Proceedings of the 6th International Conference on Predictive Models in Software Engineering*, pages 1–10. ACM, 2010.

[70] 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*, 21:2072–2106, 2016.

[71] Y. Kamei, E. Shihab, B. Adams, A. E. Hassan, A. Mockus, A. Sinha, and N. Ubayashi. A large-scale empirical study of just-in-time quality assurance. *IEEE Transactions on Software Engineering*, 39(6):757–773, 2013.

[72] R. Kapur and B. Sodhi. A defect estimator for source code: Linking defect reports with programming constructs usage metrics. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 29(2):1–35, 2020.

[73] R.-M. Karampatsis and C. Sutton. How often do single-statement bugs occur? the manysstubs4j dataset. In *Proceedings of the International Conference on Mining Software Repositories (MSR 2020)*, pages 573–577, 2020.

[74] L. Kaufman and P. J. Rousseeuw. *Finding groups in data: an introduction to cluster analysis*. John Wiley & Sons, 2005.

[75] S. Kim and E. J. Whitehead Jr. How long did it take to fix bugs? In *Proceedings of the 2006 International Workshop on Mining Software Repositories (MSR)*, pages 173–174. ACM, 2006.

[76] S. Kim, E. J. Whitehead Jr, and Y. Zhang. Classifying software changes: Clean or buggy? *IEEE Transactions on Software Engineering*, 34(2):181–196, 2008.

[77] S. Kim, T. Zimmermann, E. J. Whitehead Jr, and A. Zeller. Predicting faults from cached history. In *Proceedings of the 29th International Conference on Software Engineering (ICSE)*, pages 489–498. IEEE, 2007.

[78] Y. Kim. Convolutional neural networks for sentence classification. *arXiv:1408.5882*, 2014.

[79] D. Kingma and J. Ba. Adam: A method for stochastic optimization. *arXiv:1412.6980*, 2014.

[80] B. A. Kitchenham, E. Mendes, and G. H. Travassos. Cross versus within-company cost estimation studies: A systematic review. *IEEE Transactions on Software Engineering*, 33(5):316–329, 2007.

[81] T. Kohonen. The self-organizing map. *Proceedings of the IEEE*, 78(9):1464–1480, 1990.

[82] M. Kondo, C.-P. Bezemer, Y. Kamei, A. E. Hassan, and O. Mizuno. The impact of feature reduction techniques on defect prediction models. *Empirical Software Engineering*, 24:1925–1963, 2019.

[83] M. Kondo, D. M. German, O. Mizuno, and E.-H. Choi. The impact of context metrics on just-in-time defect prediction. *Empirical Software Engineering*, 25:890–939, 2020.

[84] M. Kondo, K. Mori, O. Mizuno, and E.-H. Choi. Just-in-time defect prediction applying deep learning to source code changes. *IPSJ Journal*, 59(4):1250–1261, 2018. in Japanese.

[85] S. Kpodjedo, F. Ricca, P. Galinier, Y.-G. Guéhéneuc, and G. Antoniol. Design evolution metrics for defect prediction in object oriented systems. *Empirical Software Engineering*, 16:141–175, 2011.

[86] H. Krasner. The Cost of Poor Quality Software in the US: A 2018 Report, 2018. Consortium for IT Software Quality.

[87] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In *Proceedings of the 25th International Conference on Neural Information Processing Systems (NIPS)*, pages 1097–1105, 2012.

[88] M. Kuhn. The caret package. http://topepo.github.io/caret/index.html, 2019. [Online; accessed 2020-11-23].

[89] N. Landwehr, M. Hall, and E. Frank. Logistic model trees. *Machine Learning*, 59:161–205, 2005.

[90] S. Lessmann, B. Baesens, C. Mues, and S. Pietsch. Benchmarking classification models for software defect prediction: A proposed framework and novel findings. *IEEE Transactions on Software Engineering*, 34(4):485–496, 2008.

[91] J. Li, P. He, J. Zhu, and M. R. Lyu. Software defect prediction via convolutional neural network. In *Proceedings of the 2017 Software Quality, Reliability and Security (QRS)*, pages 318–328. IEEE, 2017.

[92] K. Li, Z. Xiang, T. Chen, S. Wang, and K. C. Tan. Understanding the automated parameter optimization on transfer learning for cross-project

defect prediction: an empirical study. In *Proceedings of the 42nd International Conference on Software Engineering*, pages 566–577, 2020.

[93] Z. Li, X. Jing, X. Zhu, H. Zhang, B. Xu, and S. Ying. On the multiple sources and privacy preservation issues for heterogeneous defect prediction. *IEEE Transactions on Software Engineering*, 45(4):391–411, 2019.

[94] Y. Liu, Y. Li, J. Guo, Y. Zhou, and B. Xu. Connecting software metrics across versions to predict defects. In *Proceedings of the 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 232–243. IEEE, 2018.

[95] L. Madeyski and M. Jureczko. Which process metrics can significantly improve defect prediction models? an empirical study. *Software Quality Journal*, 23(3):393–422, 2015.

[96] T. Martinetz and K. Schulten. A "neural-gas" network learns topologies. *Artificial Neural Networks*, 1:397–402, 01 1991.

[97] S. Matsumoto, Y. Kamei, A. Monden, K.-i. Matsumoto, and M. Nakamura. An analysis of developer metrics for fault prediction. In *Proceedings of the 6th International Conference on Predictive Models in Software Engineering*, pages 1–9, 2010.

[98] T. J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering (TSE)*, SE-2(4):308–320, 1976.

[99] J. H. McDonald. *Handbook of Biological Statistics (3rd ed.).* Sparky House Publishing, Baltimore, Maryland., 2014.

[100] S. McIntosh and Y. Kamei. Are fix-inducing changes a moving target? a longitudinal case study of just-in-time defect prediction. *IEEE Transactions on Software Engineering*, 44(5):412–428, 2018.

[101] T. Menzies, J. Greenwald, and A. Frank. Data mining static code attributes to learn defect predictors. *IEEE Transactions on Software Engineering (TSE)*, 33(1):2–13, 2007.

[102] T. Menzies, Z. Milton, B. Turhan, B. Cukic, Y. Jiang, and A. Bener. Defect prediction from static code features: current results, limitations, new approaches. *Automated Software Engineering*, 17:375–407, 2010.

[103] Microsoft. Overview of c++ statements. https://docs.microsoft.com/ja-jp/cpp/cpp/overview-of-cpp-statements, 2016. [Online; accessed 2018-1-31].

[104] T. Mikolov. word2vec: Tool for computing continuous distributed representations of words. https://code.google.com/archive/p/word2vec/, 2015. [Online; accessed 2017-10-17].

[105] O. Mizuno and T. Kikuno. Training on errors experiment to detect fault-prone software modules by spam filter. In *Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, pages 405–414. ACM, 2007.

[106] A. Mockus and L. G. Votta. Identifying reasons for software changes using historic databases. In *Proceedings of the 22th International Conference on Software Maintenance (ICSM)*, pages 120–130. IEEE, 2000.

[107] G. Montavon, G. Orr, and K.-R. Müller. *Neural Networks: Tricks of the Trade*. Springer, 2012.

[108] S. Morasca and L. Lavazza. On the assessment of software defect prediction models via roc curves. *Empirical Software Engineering*, 25:3977–4019, 2020.

[109] T. Mori and N. Uchihira. Balancing the trade-off between accuracy and interpretability in software defect prediction. *Empirical Software Engineering*, 24:779–825, 2019.

[110] R. Moser, W. Pedrycz, and G. Succi. A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In *Proceedings of the 30th International Conference on Software Engineering (ICSE)*, pages 181–190, 2008.

[111] K. Muthukumaran, A. Rallapalli, and N. Murthy. Impact of feature selection techniques on bug prediction models. In *Proceedings of the 8th India Software Engineering Conference*, pages 120–129. ACM, 2015.

[112] N. Nagappan, T. Ball, and A. Zeller. Mining metrics to predict component failures. In *Proceedings of the 28th International Conference on Software Engineering (ICSE)*, pages 452–461. ACM, 2006.

[113] N. Nagappan, B. Murphy, and V. Basili. The influence of organizational structure on software quality. In *Proceedings of the 30th International Conference on Software Engineering (ICSE)*, pages 521–530. IEEE, 2008.

[114] J. Nam, W. Fu, S. Kim, T. Menzies, and L. Tan. Heterogeneous defect prediction. *IEEE Transactions on Software Engineering*, 44(9):874–896, 2018.

[115] J. Nam and S. Kim. CLAMI: Defect prediction on unlabeled datasets. In *Proceedings of the 30th International Conference on Automated Software Engineering (ASE)*, pages 452–463. IEEE, 2015.

[116] J. Nam and S. Kim. Heterogeneous defect prediction. In *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering (FSE)*, pages 508–519. ACM, 2015.

[117] J. Nam, S. J. Pan, and S. Kim. Transfer defect learning. In *Proceedings of the 2013 International Conference on Software Engineering (ICSE)*, pages 382–391. IEEE, 2013.

[118] NASA. Metrics data program. http://openscience.us/repo/defect/mccabehalsted/. [Online; accessed 2016-9-1].

[119] D. E. Neumann. An enhanced neural network technique for software risk analysis. *IEEE Transactions on Software Engineering (TSE)*, 28(9):904–912, 2002.

[120] C. Ni, X. Xia, D. Lo, X. Chen, and Q. Gu. Revisiting supervised and unsupervised methods for effort-aware cross-project defect prediction. *IEEE Transactions on Software Engineering*, 2020. Early Access.

[121] M. Nielsen. Neural networks and deep learning. http://neuralnetworksanddeeplearning.com/index.html, 2017. [Online; accessed 2017-9-13].

[122] T. J. Ostrand, E. J. Weyuker, and R. M. Bell. Programmer-based fault prediction. In *Proceedings of the 6th International Conference on Predictive Models in Software Engineering*, pages 1–10, 2010.

[123] F. Palomba, M. Zanoni, F. A. Fontana, A. De Lucia, and R. Oliveto. Toward a smell-aware bug prediction model. *IEEE Transactions on Software Engineering*, 45(2):194–218, 2019.

[124] S. J. Pan, I. W. Tsang, J. T. Kwok, and Q. Yang. Domain adaptation via transfer component analysis. *IEEE Transactions on Neural Networks*, 22(2):199–210, 2011.

[125] L. Pascarella, F. Palomba, and A. Bacchelli. Re-evaluating method-level bug prediction. In *Proceedings of the 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 592–601. IEEE, 2018.

[126] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and Édouard Duchesnay. Scikit-learn: Machine learning in python. *Journal of Machine Learning Research*, 12(85):2825–2830, 2011.

[127] J. Pennington, R. Socher, and C. D. Manning. Glove: Global vectors for word representation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, volume 14, pages 1532–1543, 2014.

[128] J. Petrić, D. Bowes, T. Hall, B. Christianson, and N. Baddoo. The jinx on the NASA software defect data sets. In *Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering*, pages 1–5. ACM, 2016.

[129] Y. Qu, T. Liu, J. Chi, Y. Jin, D. Cui, A. He, and Q. Zheng. node2defect: using network embedding to improve software defect prediction. In *Proceedings of the 33rd International Conference on Automated Software Engineering (ASE)*, pages 844–849. IEEE, 2018.

[130] Y. Qu, Q. Zheng, J. Chi, Y. Jin, A. He, D. Cui, H. Zhang, and T. Liu. Using k-core decomposition on class dependency networks to improve bug prediction model's practical performance. *IEEE Transactions on Software Engineering*, 2019. Early Access.

[131] R. Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann Publishers, 1993.

[132] F. Rahman and P. Devanbu. How, and why, process metrics are better. In *Proceedings of the 35th International Conference on Software Engineering (ICSE)*, pages 432–441. IEEE, 2013.

[133] S. S. Rathore and A. Gupta. A comparative study of feature-ranking and feature-subset selection techniques for improved fault prediction. In *Proceedings of the 7th India Software Engineering Conference*, pages 1–10. ACM, 2014.

[134] J. Ren, K. Qin, Y. Ma, and G. Luo. On software defect prediction using machine learning. *Journal of Applied Mathematics*, 2014:1–8, 2014. Art. ID 785435.

[135] M. E. Rice and G. T. Harris. Comparing effect sizes in follow-up studies: Roc area, cohen's d, and r. *Law and human behavior*, 29:615–620, 2005.

[136] D. Rodríguez, R. Ruiz, J. Cuadrado-Gallego, and J. Aguilar-Ruiz. Detecting fault modules applying feature selection to classifiers. In *Proceedings of the 2007 International Conference on Information Reuse and Integration*, pages 667–672. IEEE, 2007.

[137] D. Rodriguez, R. Ruiz, J. Cuadrado-Gallego, J. Aguilar-Ruiz, and M. Garre. Attribute selection in software engineering datasets for detecting fault modules. In *Proceedings of the 2007 EUROMICRO Conference on Software Engineering and Advanced Applications*, pages 418–423. IEEE, 2007.

[138] G. Rodriguezperez, M. Nagappan, and G. Robles. Watch out for extrinsic bugs! a case study of their impact in just-in-time bug prediction models on the openstack project. *IEEE Transactions on Software Engineering*, 2020. Early Access.

[139] L. Rokach and O. Maimon. Clustering methods. In *Data mining and knowledge discovery handbook*, pages 321–352. Springer, 2005.

[140] P. Romanski and L. Kotthoff. Fselector. https://cran.r-project.org/web/packages/FSelector/FSelector.pdf, 2018. [Online; accessed 2018-11-11].

[141] C. Rosen, B. Grawi, and E. Shihab. Commit guru: Analytics and risk prediction of software commits. In *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE, pages 966–969. ACM, 2015.

[142] C. E. Shannon. A mathematical theory of communication. *Bell system technical journal*, 27(3):379–423, 1948.

[143] M. Shepperd, D. Bowes, and T. Hall. Researcher bias: The use of machine learning in software defect prediction. *IEEE Transactions on Software Engineering*, 40(6):603–616, 2014.

[144] M. Shepperd, Q. Song, Z. Sun, and C. Mair. Data quality: Some comments on the NASA software defect datasets. *IEEE Transactions on Software Engineering (TSE)*, 39(9):1208–1215, 2013.

[145] E. Shihab. *An exploration of challenges limiting pragmatic software defect prediction*. PhD thesis, Queen's University (Canada), 2012.

[146] E. Shihab. Practical software quality prediction. In *Proceedings of the 2014 International Conference on Software Maintenance and Evolution (ICSME)*, pages 639–644. IEEE, 2014.

[147] E. Shihab, A. E. Hassan, B. Adams, and Z. M. Jiang. An industrial study on the risk of software changes. In *Proceedings of the 20th International Symposium on the Foundations of Software Engineering (FSE)*, pages 1–11. ACM, 2012.

[148] E. Shihab, Z. M. Jiang, W. M. Ibrahim, B. Adams, and A. E. Hassan. Understanding the impact of code and process metrics on post-release defects: A case study on the Eclipse project. In *Proceedings of the International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 1–10. ACM, 2010.

[149] S. Shivaji, E. J. Whitehead, R. Akella, and S. Kim. Reducing features to improve code change-based bug prediction. *IEEE Transactions on Software Engineering (TSE)*, 39(4):552–569, 2013.

[150] J. Śliwerski, T. Zimmermann, and A. Zeller. When do changes induce fixes? In *Proceedings of the 2th International Workshop on Mining Software Repositories (MSR)*, number 4, pages 1–5. ACM, 2005.

[151] P. Smolensky. Information processing in dynamical systems: Foundations of harmony theory. Technical report, DTIC Document, 1986.

[152] Q. Song, Y. Guo, and M. Shepperd. A comprehensive investigation of the role of imbalanced learning for software defect prediction. *IEEE Transactions on Software Engineering*, 45(12):1253–1269, 2019.

[153] A. Stevenson and C. A. Lindberg. *New Oxford American Dictionary*. Oxford University Press, 2010.

[154] S. Tabassum, L. L. Minku, D. Feng, G. G. Cabral, and L. Song. An investigation of cross-project learning in online just-in-time software defect prediction. In *Proceedings of the 42nd International Conference on Software Engineering*, pages 554–565, 2020.

[155] M. Tan, L. Tan, S. Dara, and C. Mayeux. Online defect prediction for imbalanced data. In *Proceedings of the 37th International Conference on Software Engineering (ICSE)*, pages 99–108. IEEE, 2015.

[156] C. Tantithamthavorn. *Towards a Better Understanding of the Impact of Experimental Components on Defect Prediction Models*. PhD thesis, Nara Institute of Science and Technology (NAIST), 2016.

[157] C. Tantithamthavorn and A. E. Hassan. An experience report on defect modelling in practice: Pitfalls and challenges. In *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*, pages 286–295. ACM, 2018.

[158] C. Tantithamthavorn, A. E. Hassan, and K. Matsumoto. The impact of class rebalancing techniques on the performance and interpretation of defect prediction models. *IEEE Transactions on Software Engineering*, 46(11):1200–1219, 2020.

[159] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, and K. Matsumoto. Automated parameter optimization of classification techniques for defect prediction models. In *Proceedings of the 38th International Conference on Software Engineering (ICSE)*, pages 321–332. ACM, 2016.

[160] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, and K. Matsumoto. An empirical comparison of model validation techniques for defect prediction models. *IEEE Transactions on Software Engineering (TSE)*, 43(1):1–18, 2017.

[161] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, and K. Matsumoto. The impact of automated parameter optimization on defect prediction models. *IEEE Transactions on Software Engineering*, 45(7):683–711, 2019.

[162] G. Tassey. *The economic impacts of inadequate infrastructure for software testing*. National Institute of Standards and Technology, 2002.

[163] S. Thomas, W. lscp: A lightweight source code preprocesser. https://github.com/doofuslarge/lscp, 2012. [Online; accessed 2018-1-31].

[164] S. Tiwari and S. S. Rathore. Coupling and cohesion metrics for object-oriented software: A systematic mapping study. In *Proceedings of the 11th Innovations in Software Engineering Conference (ISEC)*, pages 1–11, 2018.

[165] H. Tong, B. Liu, and S. Wang. Kernel spectral embedding transfer ensemble for heterogeneous defect prediction. *IEEE Transactions on Software Engineering*, 2019. Early Access.

[166] H. Tu, Z. Yu, and T. Menzies. Better data labelling with emblem (and how that impacts defect prediction). *IEEE Transactions on Software Engineering*, 2020. Early Access.

[167] L. van der Maaten. Accelerating t-SNE using tree-based algorithms. *Journal of Machine Learning Research*, 15(93):3221–3245, 2014.

[168] L. van der Maaten and G. Hinton. Visualizing data using t-SNE. *Journal of Machine Learning Research*, 9(86):2579–2605, 2008.

[169] U. Von Luxburg. A tutorial on spectral clustering. *Statistics and computing*, 17:395–416, 2007.

[170] S. Wang, T. Liu, J. Nam, and L. Tan. Deep semantic feature learning for software defect prediction. *IEEE Transactions on Software Engineering*, 2018. Early Access.

[171] S. Wang, T. Liu, and L. Tan. Automatically learning semantic features for defect prediction. In *Proceedings of the 38th International Conference on Software Engineering (ICSE)*, pages 297–308. ACM, 2016.

[172] S. Wattanakriengkrai, P. Thongtanunam, C. Tantithamthavorn, H. Hata, and K. Matsumoto. Predicting defective lines using a model-agnostic technique. *IEEE Transactions on Software Engineering*, 2020. Early Access.

[173] M. Wen, R. Wu, and S.-C. Cheung. How well do change sequences predict defects? sequence learning from software changes. *IEEE Transactions on Software Engineering*, 46(11):1155–1175, 2020.

[174] Z. Xu, J. Liu, X. Luo, and T. Zhang. Cross-version defect prediction via hybrid active learning with kernel principal component analysis. In *Proceedings of the 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 209–220. IEEE, 2018.

[175] Z. Xu, J. Liu, Z. Yang, G. An, and X. Jia. The impact of feature selection on defect prediction performance: An empirical comparison. In *Proceedings of the 27th International Symposium on Software Reliability Engineering (ISSRE)*, pages 309–320. IEEE, 2016.

[176] X. Yang, D. Lo, X. Xia, Y. Zhang, and J. Sun. Deep learning for just-in-time defect prediction. In *Proceedings of the International Conference on Software Quality, Reliability and Security (QRS)*, pages 17–26. IEEE, 2015.

[177] Y. Yang, Y. Zhou, J. Liu, Y. Zhao, H. Lu, L. Xu, B. Xu, and H. Leung. Effort-aware just-in-time defect prediction: simple unsupervised models could be better than supervised models. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 157–168, 2016.

[178] S. Yatish, J. Jiarpakdee, P. Thongtanunam, and C. Tantithamthavorn. Mining software defects: should we consider affected releases? In *Proceedings of the 41st International Conference on Software Engineering (ICSE)*, pages 654–665. IEEE, 2019.

[179] T. Yu, W. Wen, X. Han, and J. H. Hayes. Conpredictor: concurrency defect prediction in real-world applications. *IEEE Transactions on Software Engineering*, 45(6):558–575, 2019.

[180] X. Yu, K. E. Bennin, J. Liu, J. W. Keung, X. Yin, and Z. Xu. An empirical study of learning to rank techniques for effort-aware defect prediction. In *Proceedings of the 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 298–309. IEEE, 2019.

[181] F. Zhang, Q. Zheng, Y. Zou, and A. E. Hassan. Cross-project defect prediction using a connectivity-based unsupervised classifier. In *Proceedings of the 38th International Conference on Software Engineering (ICSE)*, pages 309–320. ACM, 2016.

[182] H. Zhang. The optimality of Naive Bayes. In *FLAIRS Conference*. AAAI Press, 2004.

[183] X. Zhang, J. Zhao, and Y. LeCun. Character-level convolutional networks for text classification. In *Proceedings of the 28th International Conference on Neural Information Processing Systems (NIPS)*, pages 649–657, 2015.

[184] Y. Zhang, R. Jin, and Z.-H. Zhou. Understanding bag-of-words model: a statistical framework. *International Journal of Machine Learning and Cybernetics*, 1:43–52, 2010.

[185] A. Zheng and A. Casari. *Feature Engineering for Machine Learning: Principles and Techniques for Data Scientists*. O'Reilly Media, Inc., 1st edition, 2018.

[186] 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 *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering (ESEC-FSE)*, pages 91–100. ACM, 2009.

[187] T. Zimmermann, R. Premraj, and A. Zeller. Predicting defects for eclipse. In *Proceedings of the 3th International Workshop on Predictor Models in Software Engineering*, page 9. IEEE, 2007.

[188] D. Zwillinger and S. Kokoska. *CRC standard probability and statistics tables and formulae*. Crc Press, 2000.