Noname manuscript No.
(will be inserted by the editor)

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

Masanari Kondo · Daniel M. German · Osamu Mizuno · Eun-Hye Choi ·

the date of receipt and acceptance should be inserted later

Abstract 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 consider n-lines ($n = 1, 2, \cdots$) that precede and follow the changed lines (which we call *context lines*), and propose metrics that measure them, which we call "*Context Metrics*." Specifically, these context metrics are defined as the number of words/keywords in the context lines. In a large-scale empirical study using six open source software projects, we compare the performance of using our context metrics, traditional code churn metrics (e.g., the number of modified subsystems), our *extended context metrics* which measure not only context lines but also changed lines, and *combination metrics* that use two extended context metrics at a prediction model for defect prediction. The results show that context metrics that consider the context lines of added-lines achieve the best median value in all cases in terms of a statistical test. Moreover, using few number of context lines is suitable for context metric that considers words, and using more number of context lines is suitable for context metric that considers keywords. Finally, the combination metrics of two extended context metrics significantly out-

Masanari Kondo, Osamu Mizuno
Software Engineering Laboratory (SEL)
Kyoto Institute of Technology, Japan
E-mail: m-kondo@se.is.kit.ac.jp, o-mizuno@kit.ac.jp

Daniel M. German
Department of Computer Science, University of Victoria,
Victoria, BC, Canada
E-mail: dmg@uvic.ca

Eun-Hye Choi
Information Technology Research Institute,
National Institute of Advanced Industrial Science and Technology, Japan
E-mail: e.choi@aist.go.jp

perform all studied metrics in all studied projects w. r. t. the area under the receiver operation characteristic curve (AUC) and Matthews correlation coefficient (MCC).

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 [58].

There exists plenty of work aimed at predicting defective components [2, 8, 16, 29, 38]. In particular, several prior research work has focused on predicting defective changes called *change-level defect prediction*—also called *just-in-time defect prediction* [10, 22, 27, 37]. 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 [17] and providing faster feedback than other defect prediction methods [22]. Previous research has used metrics based on measuring the code changes (e.g., churn–*changed lines*) in just-in-time defect prediction [22, 27, 37].

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 paper, 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" [53]. In this paper, 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 paper proposes several *context metrics*. The different metrics 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 metrics. We refer to a context metric which uses a set of the parameters as a variant of context metrics. We empirically study the best-performing variant in terms of defect prediction performance. We also compare the context metrics that are the best-performing variants with traditional *code churn metrics* (*change metrics* [22, 27, 37] and *indentation metrics* [18]), *extended context metrics* and combination metrics that use two extended context metrics. Indentation metrics 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 metrics as code churn metrics, since they are computed on changed lines. In order to improve the predicting power of the context metrics in defect prediction, we also define extended context metrics. *Extended context metrics* count the number of words/keywords in both, the context lines and the changed lines. Hence, extended context metrics are hybrids of the context metrics and traditional code churn

metrics. In addition, we use *combination metrics* that use two extended context metrics 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 metrics in defect prediction.

Using six large open source software projects (from different domains) we empirically evaluate the defect prediction power of context metrics and compare them against traditional change metrics. 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 metrics on defect prediction?
RQ2: Do context lines improve the performance of defect prediction?
RQ3: What is the impact of combination metrics of context metrics on defect prediction?

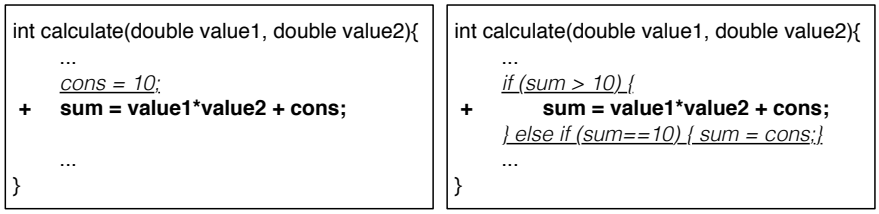
The main findings of our paper are as follows:

- The best performing context metrics are the ones that measure the context of added-lines only.
- The prediction power of context metrics varies when different sizes of the context (number of lines around the change) are used. The optimal size of the context for the metric that uses number of words is smaller than the optimal size for the metric that uses keywords.
- The number of "goto" statements in context lines and changed lines is a good indicator of defective commits.
- Our proposed combination metrics of extended context metrics significantly outperform all the metrics that are used in this paper, and achieve the best-performing metrics 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).

This paper is organized as follows: Section 2 shows motivation example. Section 3 introduces related work. Section 4 explains our proposed context metrics. Section 5 presents our case study design. Section 6 describes research questions and methodology. Section 7 presents the results of our case study. Section 8 discusses the results. Section 9 describes the threats to the validity of our findings. Section 10 presents the conclusion.

2 Motivating Example

Let us start from a simple example to illustrate the use of context lines to measure the complexity of changes. Figure 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 1(a) has simple context lines: there is one assignment before the changed line and one empty line after the changed line. The changed in



(a) Simple context lines. (b) Complex context lines.

Fig. 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.

Figure 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 paper, we introduce two types of new metrics that use the context lines: context metrics and extended context metrics, and evaluate their performance in defect prediction.

There are complexity metrics, such as Halstead's complexity metrics [14] and McCabe's Cyclomatic complexity metrics [32], that can capture the complexity of the function being changed and take into consideration the context; however, (1) to compute these metrics we need all the lines of the functions, (2) these metrics are limited because they require a parser, and (3) complexity metrics are not optimized for code churn. In contrast, context metrics 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.

3 Related Work

3.1 Source Code Churn

Many researchers have studied source code churn for software defect, reliability and quality [12, 22–25, 27, 37, 39–41, 43]. Source code churn measures changes and extensions of source code in a period of time [42]. Munson et al. [39] reported that, as a system is developed (evolved), complexity of the system is also changed. They proposed a methodology to produce an indicator of defects based on this tendency. Nagappan et al. [40] predicted defect density between different releases of Windows Server 2003. Comparing traditional code churn metrics with relative code churn met-

rics, which relate proportion of code churn such as size of its component, they found the relative code churn metrics are strong metrics for the defect density.

Prior studies proposed more complex code churn metrics [16, 18]. Hassan [16] proposed code churn metrics based on the code change process. He applied Shannon entropy (from information theory) to the code change process in order to formulate his metrics.

Hindle et al. [18] proposed indentation metrics that measure the indentations of added-lines and fixed-lines of changes. They studied the correlations between the indentation metrics and traditional complexity metrics (McCabe's Cyclomatic complexity [32] and Halstead's complexity [14]). They showed that the indentation metrics are mildly or strongly correlated with the traditional complexity metrics and the indentation is potentially its own complexity metric [18]. Because indentation metrics use the information in changed or added lines, we refer to indentation metrics as a type of code churn metric. This paper is the first study to investigate the effectiveness of indentation metrics for defect prediction.

In this paper, we compare the prediction power of 6 types of metrics in defect prediction. These metrics are: 1) context metrics, 2) traditional code churn metrics [22, 27, 37], 3) each of traditional code churn metrics, 4) code churn metrics based on indentation metrics [18], 5) extended context metrics (which are combinations between context metrics and a traditional code churn metric) and 6) combination metrics of extended context metrics (which are two extended context metrics that are (1) number of words and (2) number of a certain keyword at a prediction model).

3.2 Text-Based/Just-In-Time Defect Prediction

Many researchers have tackled the problem of defect prediction [1, 3, 20, 22, 27, 28, 31, 36, 52, 60, 61, 63]. In addition, several researchers have proposed metrics to predict defective components [2, 8, 16, 29, 38]. Mizuno et al. [36] applied spam filter to defect prediction problem. Śliwerski et al. [52] proposed a method that automatically identifies changes that lead to defects in the future.

Textual information has also being used for defect prediction [1, 27, 31, 36, 60]. Kim et al. [27] used not only metadata and complexity metrics but also text information to build a prediction model and predicted defects. They used change-log messages, source code and file names as input to their predictors.

Wang et al. [60] used the programs' Abstract Syntax Trees (ASTs) as a representation of source code. They applied a deep learning technique to ASTs in order to learn semantic features from token vectors.

Several researchers have worked on just-in-time defect prediction [1, 10, 16, 20, 22, 27–29, 37, 61]. Just-in-time defect prediction aims at identifying defective code changes, such as commits, instead of identifying defective files or packages as in traditional file/package-level defect prediction. For example, Kamei et al. [22] focused on predicting the risk of commits. They used change metrics to predict defective commits at the time of committing commits. Yang et al. [61] applied a deep learning technique as a prediction model to change metrics and conducted just-in-time defect prediction. Just-in-time defect prediction has the following three benefits that address

the challenges on file/package-level defect prediction [22]: (1) prediction targets are fine-grained, (2) relevant-developers can be identified, and (3) the prediction-period is faster. In this paper, we use context metrics for just-in-time defect prediction.

There are several widely known pitfalls that should be avoided in defect prediction [54, 55]. For example, Tan et al. [54] reported that cross validation technique is frequently used to evaluate prediction models [3, 20, 22, 27, 28]. However, this technique risks to mix past and future commits; an unrealistic scenario that artificially improves results. In our study, we take into consideration their recommendations to avoid these potential pitfalls. This technique called *online change classification* is a validation technique without the risks. We describe the details in Section 5.4.

4 Context Metrics

In this section, we describe the implementation of the proposed context metrics. 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 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 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' [1].

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

For explaining context metrics, we define the following terminology:

[1] 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.

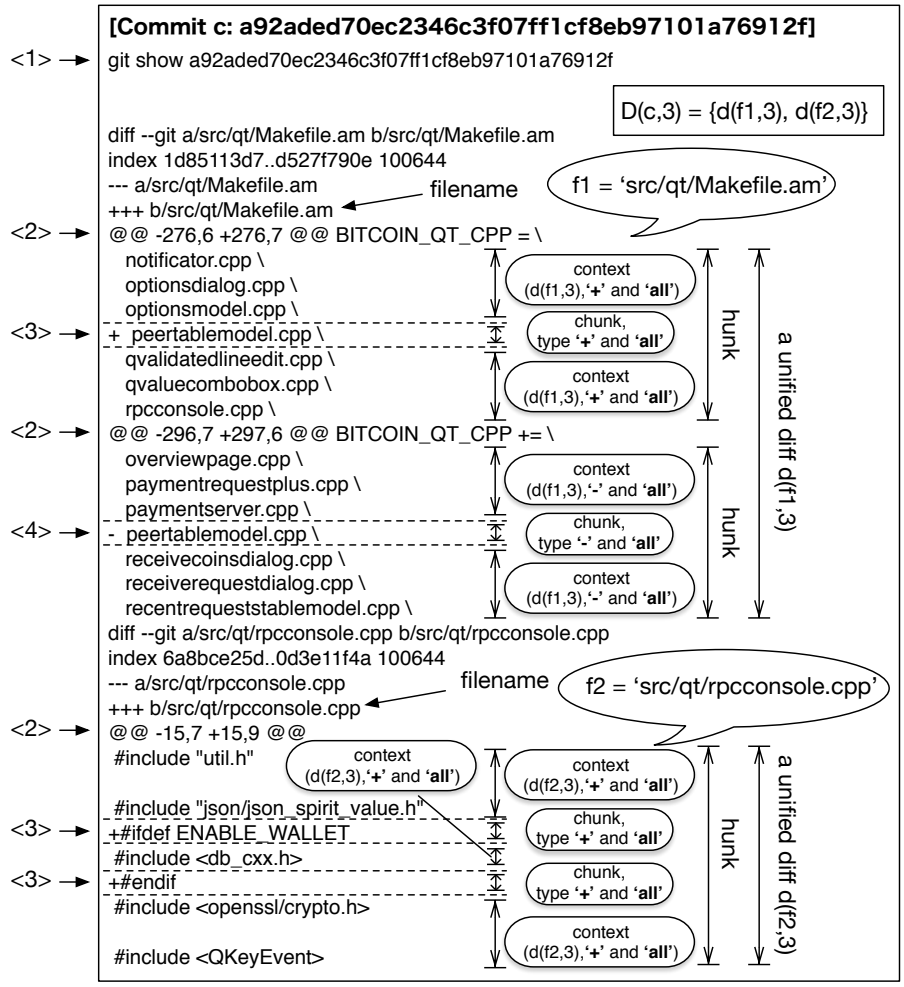


Fig. 2: An example of unified diffs of a commit with context size equal to three produced by `git show` (<1>) in Bitcoin project; due to the space limitation, we remove the metadata of this commit (the commit comment and the author information). This commit consists of two source code file diffs. The 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/'.

Table 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

- 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 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 2 shows all studied keywords)[2] 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 metrics:

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

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

The defined context metrics are described in Table 3. To compute the context metrics 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[3] of commit c with the given number of lines of context, n, using the following command:
   `git show --unified=n c`

[2] The keywords refer to reserved words (statements) in C++ that are shown by Microsoft Visual Studio [35]. 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".

[3] 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 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

Table 3: Different context metrics. "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 '-'.

Metrics	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.

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.
   (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`[4] [59] 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 metric $NCW/NCKW$ 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 3 depicts an example showing how the context metrics 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 metrics.

The intuition behind counting words or keywords:

Our definition of context metrics 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.

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

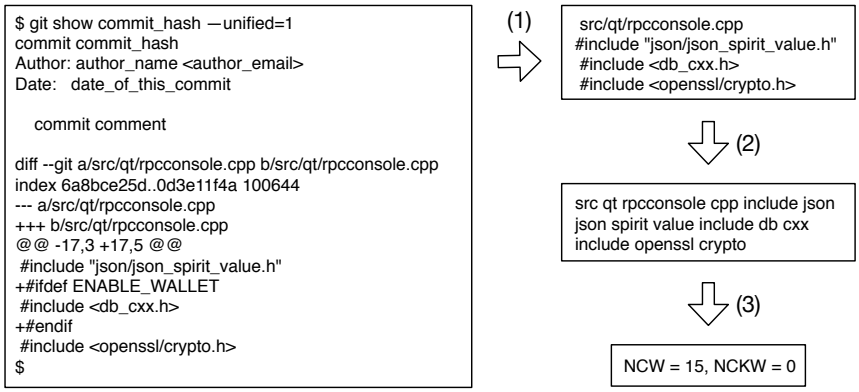


Fig. 3: Example showing how NCW and NCKW are computed from a unified diff. The unified diff corresponds to the change from Figure 2; due to the space limitation, we remove several hunks, the commit comment, the author information, and the commit hash from the unified diff. The number of context lines 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 metrics.

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

5 Case Study Design

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

5.1 Indentation Metrics

We compare context metrics with indentation metrics. We study two indentations metrics: *Added Spaces (AS)*, defined by Hindle et.al [18]; AS is the sum of the number of white spaces on all the '+' lines in a commit.

We additionally define a new indentation metric *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 metrics as way to predict defects:

Table 4: Change metrics.

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

The indentation metrics have been used as a proxy to measure complexity of source code [18]. 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 expect that changes to code that has more indentation might result in more defects that changes to code that has less indentation.

5.2 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. [47] provide publicly. Commit Guru is a web application, which identifies and predicts defective commits for Git repositories and calculates the change metrics (Table 4) that are often used for just-in-time defect prediction [22].

In this paper, we use Commit Guru to calculate the change metrics [22]. We use the change metrics in RQ2 to compare with the context metrics in order to study what is the impact of the context metrics on defect prediction. Then, we use the change metrics, and their subsets (each of the change metrics) as studied metrics.

We refer to each metric in the change metrics as a subset of the change metrics. When using a subset of the change metrics, we pick up a metric from the change

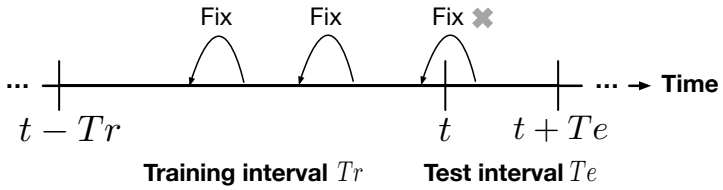


Fig. 4: An example of the time sensitive change classification. The cross in gray indicates the information of fixing a commit is not used in the training interval.

metrics, and use that metric for defect prediction. This is because each of the change metrics is also a churn metric. However, several metrics do not strongly relate to code churn. For example, Purpose metric (i.e., FIX, described in Table 4) is not affected by code churn. Hence, we remove three types of metrics from all the change metrics when considering their subsets that are Purpose metric (i.e., FIX), History metrics (i.e., NDEV, AGE, and NUC), and Experience metrics (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 metrics. 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 metrics, to avoid using several strongly correlated metrics in the prediction, we apply the following preprocessing proposed and described in [22]:

- 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 [62] to the changed metrics to normalized to a mean of 0 and a variance of 1.

5.3 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 [54]. 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* [54].

Time sensitive change classification uses only past commits to label past commits and build prediction models for future commits. Figure 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.

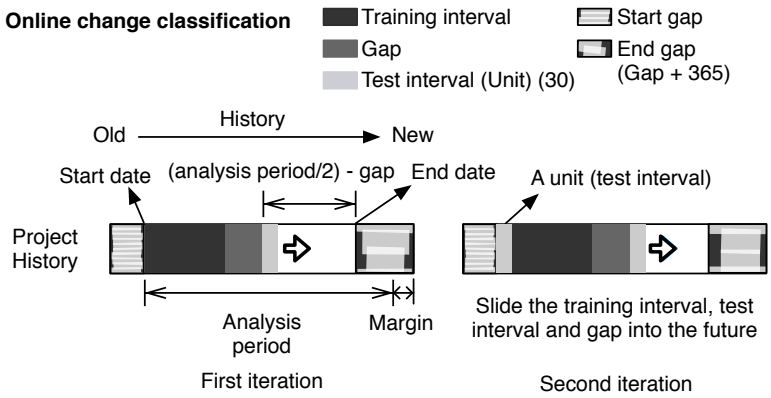


Fig. 5: An overview of the online change classification. We show two iterations as an example. The part of the rectangle in black is the training data (training interval) labeled using the commits in the training interval and gap (in dark gray). The part of the rectangle in light gray is the test data (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 5.4.

However, Tan et al. [54] reported three challenges. First, because defective commits are typically detected and fixed in 100–300 days [26], 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 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. [54] recommended to use *online change classification*.

5.4 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 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 time for each project from our pre-experiment (Table 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 *train-*

Table 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

ing interval, *test interval* and *gap* slide into the future by a certain interval (Figure 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 9. The result shows that these parameters have little impact for the results of our experiments.

We also use *start gap* and *end gap* [54] 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 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,

Table 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 %

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. [47]), 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.

5.5 Preprocessing by z-score

z-score is a popular normalization approach in defect prediction [62]. z-score normalizes the input data to mean 0 and variance 1. The equation of z-score is:

$$\boldsymbol{X}_{z-score} = \frac{\boldsymbol{X}_{org} - \mu}{\sigma} \tag{1}$$

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. $\boldsymbol{X}_{org}$ is a vector of all values (all commits) of a feature. $\boldsymbol{X}_{z-score}$ is a vector of all values (all commits) of a normalized feature.

5.6 Studied Projects

For our experiments, we use six open source projects: Hadoop, Camel, Gerrit, Osmand, Bitcoin and Gimp. Table 6 shows details of the projects. The studied 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 metrics and the indentation metrics 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.

5.7 Resampling Approach

While learning the defect prediction model, the learning performance is affected by imbalanced data [54]. In our case, Table 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 [22, 54, 61].

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.

5.8 Evaluation Measures

To measure the impact of the context metrics 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)[5]. Precision and Recall are frequently used in defect prediction as evaluation measures. However, several researchers warned that these measures show biased results [5, 6, 55].

AUC and Brier score are threshold-independent measures. Tantithamthavorn et al. [55] 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 [4, 62] and we want to better understand the predicting power of the metrics [21]. Therefore, we also use MCC in this paper. The threshold of MCC is 0.5.

We use the Scott-Knott ESD test [57] (using 95% significance level) to compare the context metrics and the traditional code churn metrics. The Scott-Knott test is a hierarchical clustering algorithm that ranks the distributions of values. In particular, metrics with 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 [7]. The

[5] 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.

Scott-Knot ESD test places distributions which are not significantly different, or have a negligible effect size, in the same rank. We use the `ScottKnottESD R` package[6] that was provided by Tantithamthavorn [56]. 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 metrics over all the studied projects instead of each studied project. This idea was proposed by Ghotra et al. [11]. 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 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 metric. The smallest rank, 1, indicates the best rank. The largest rank indicates the worst rank. A rank can contains multiple metrics at once. We interpret metrics which have many smallest/smaller ranks as the best metrics since it indicates that the metrics significantly outperform many others. Hence, for the Scott-Knott ESD test, we used the top-3 ranks to evaluate the metrics across the studied projects. We report metrics which have the most top-3 ranks across the studied projects as the best metrics in the Scott-Knott ESD test.

For the double Scott-Knott ESD test, we used boxplots to show the ranks of the studied metrics 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 metrics which have the smallest rank as the best metrics since it indicates that the metrics significantly outperform many others.

5.9 Prediction Models

We use two defect prediction models, logistic regression model (LR) [33] and random forest model (RF) [19]. We give a brief overview of the idea behind the prediction models:

– *Logistic Regression (LR) [33]:* LR is a frequently used defect prediction model. They build a linear model which has all metrics as explanation variables, these

[6] https://github.com/klainfo/ScottKnottESD

coefficients, and a bias. LR feeds the output of this linear model to a sigmoid function [15]. The output of the sigmoid function corresponds to the probability.
– *Random Forest (RF) [19]:* RF is an ensemble learning model. RF builds various decision trees [44] based on subsets of metrics. Finally, RF merges all the results of the decision trees, and provides the probability of defect.

Prior work [13, 56] showed that the parameter optimization of the prediction models crucially affects the prediction performance. For example, Tantithamthavorn et al. [56] 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 metrics 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 metrics and COMB. For the other metrics, we do not use the C since the number of metrics is 1 at a prediction model.

In addition, we need to consider the correlation between the studied metrics. If the studied metrics are correlated, LR would get *multicollinearity problem* [9]. When using the change metrics, we need to consider the correlation. To avoid the correlated metrics, prior work [22] proposed a preprocessing. We follow the same preprocessing of prior work [22] that was described in Section 5.2. COMB has two metrics. However, they are not correlated (see Table 21). Hence, we do not need to deal with the correlation in COMB.

For RF, we keep using the normalized change metrics 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 metrics randomly selected for each node in a tree. For example, if we set mtry=2, RF selects 2 metrics from the studied metrics 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 metrics, 1 and 2 when using COMB, and 1 when using other metrics since the number of normalized change metrics is 12, the number of metrics in COMB is 2, and the number of other metrics 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.

6 Research Questions and Methodology

6.1 Research Questions

Our proposed context metrics 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 metrics. To do this, we formulate the following research question: (RQ1) *What is the impact of the different variants of context metrics on defect prediction?*

RQ1 does not confirm what is the impact of the context metrics for defect prediction compared to the traditional code churn metrics. Hence, we also study the prediction performance of the context metrics compared to the traditional code churn metrics that are the change metrics, their subsets and the indentation metrics in order to confirm whether the context metrics are effective or not. We additionally study the performance of extended context metrics, which are combinations between the context metrics and the traditional code churn metrics for defect prediction in order to improve the predicting power of the context metrics. The extended context metrics 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 metrics, the extended context metrics, and the traditional code churn metrics. However, we do not study combination metrics between the context metrics; we use a context metric alone on a prediction model in RQ1 and RQ2. Hence, in this RQ, we study the impact of combination metrics that use two extended context metrics 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 metrics of context metrics on defect prediction?*

6.2 Methodology

We explain our experimental methodology.

6.2.1 RQ1. What is the impact of the different variants of context metrics 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 [11] and no significant difference [30, 34, 49]. The main point

in this paper is to evaluate the impact of the context metrics 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 [54]. 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 [54]. We described details in Section 5.4.

We compute the context metrics for each chunk type for each commit. We apply preprocessing to the context metrics 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 metric 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 5.9.

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 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 metric 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 metrics of the different chunk types w. r. t. the three evaluation measures. We apply the Scott-Knott ESD test [57] to the context metrics for each evaluation measure for each project. Each context metric has two values (results by LR and RF models) for each project. Then, we evaluate statistically significant differences and effect sizes between the context metrics for each evaluation measure for each project. The result is shown as a rank. For example, if a certain context metric A has the best value on a certain evaluation measure, this context metric A achieves the rank 1. If another context metric B has no significant difference to the context metric A that achieves the rank 1, this context metric B also achieves the rank 1. If another context metric C has significant difference to the context metric A and B, this context metric 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 metric. 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 metric has six ranks (results by all the studied projects) for each evaluation measure. The additional Scott-Knott ESD test compares the studied context metrics in terms of the rank. Then, we evaluate statistically significant differences and effect sizes between the context metrics 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 metric. 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 this example, we would get histograms in which the context size 1 has 40, 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 metrics variants (different chunk types and context sizes). The goal of this RQ1 is to find the best context metrics variant for just-in-time defect prediction. The best context metrics variant is considered as the context metrics in RQ2.

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

To answer this RQ, we compare the best variant of the context metrics NCW and NCKW (as determined in RQ1) with the change metrics and their subsets (both described in Section 5.2), the indentation metrics (described in Section 5.1) and the extended context metrics. We build the defect prediction models to evaluate the metrics. 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 metrics based on NCW and NCKW that measure both the context and the changed lines called *extended context metrics*. These metrics 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 7. NCCW and NCCKW use only added-lines as the changed lines. This is because it is known that a change metric, “added-lines”, is one of the best indicator of change risk [50, 51]. These metrics will show the results of the combination between the context metrics and the traditional code churn metrics. 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 metrics and their subsets that was described in Section 5.9. For the context metrics, we apply z-score to normalize to a mean of 0 and a variance of 1 since the subsets of the change metrics are also normalized by z-score.

Table 7: The extended context metrics.

Metrics	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.

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

To answer this RQ, we use our new combination metrics 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 8.3). Hence, we need to remove the correlation in order to address the multicollinearity problem [9] for using them on a prediction model.

We, hence, modify NCCKW into counting only each specific keyword instead of counting all keywords (Table 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 metrics 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 metrics for NCCW and a modified NCCKW. We call the combination metrics as *COMB*. We compare COMB with the other metrics following the same procedures of the procedure for RQ2.

7 Case Study Results

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

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

Table 8 shows the ranks of the Scott-Knott ESD test results for each evaluation measure for each context metric variant. Each cell shows the rank of a context metric 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 metric variants for each project and evaluation measure. For example, the gray cells in Table 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 metric 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

Table 8: The ranks of the Scott-Knott ESD test results for each context metric 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 15, 16, and 17).

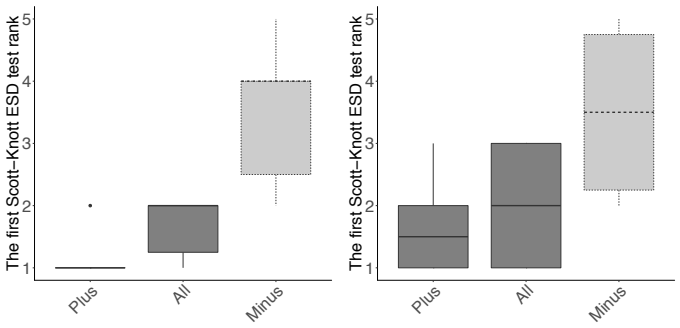
Evaluation Measures	Metrics	Chunk Types	Projects						Numbers of Ranks			
			B.	C.	Ge.	Gi.	H.	O.	*#R1*	*#R2*	*#R3*	*Sum*
AUC	NCW	+	1	1	1	1	1	2	5	1	0	6
		-	2	5	4	4	2	4	0	2	0	2
		all	1	2	2	2	1	2	2	4	0	6
	NCKW	+	1	2	2	3	1	1	3	2	1	6
		-	2	4	5	5	2	3	0	2	1	3
		all	1	3	3	3	1	1	3	0	3	6
MCC	NCW	+	2	1	2	1	3	1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

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 metrics 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 metrics for 3 of 6 studied projects.

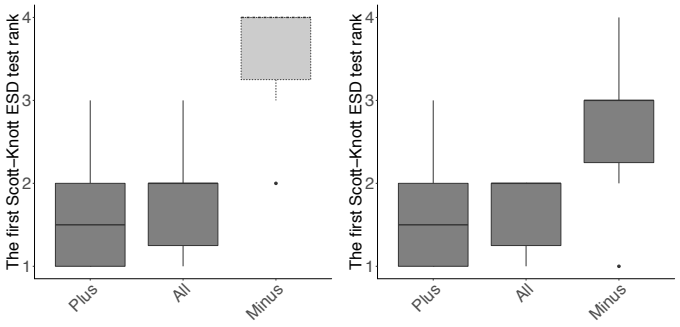
Figure 6 shows the results of the double Scott-Knott ESD test on the results for each context metric in all projects; each boxplot contains six ranks of the first Scott-Knott ESD test execution for the studied projects on a chunk type. The x-axis indicates a chunk type; Plus, Minus, and All correspond to '+', '-', and 'all'; 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 metrics.

With one exception, '+' consistently performed better than other types of chuck types. This exception is shown in Figure 6(f) shows that 'all' chunk statistically outperforms '+' chunk on NCKW on Brier score; however, the median, and 25 and 75



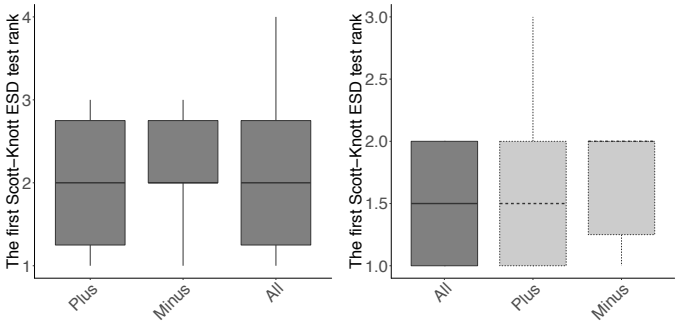
(a) NCW AUC

(b) NCKW AUC



(c) NCW MCC

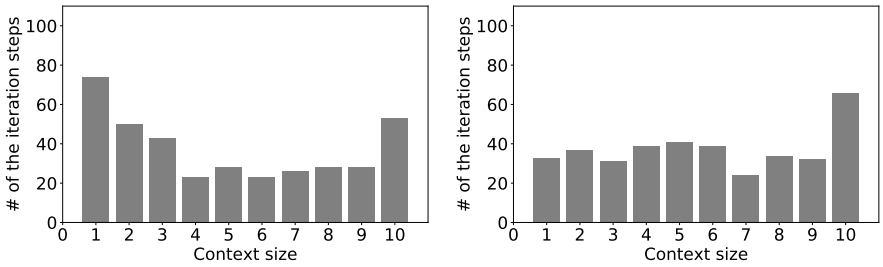
(d) NCKW MCC



(e) NCW Brier

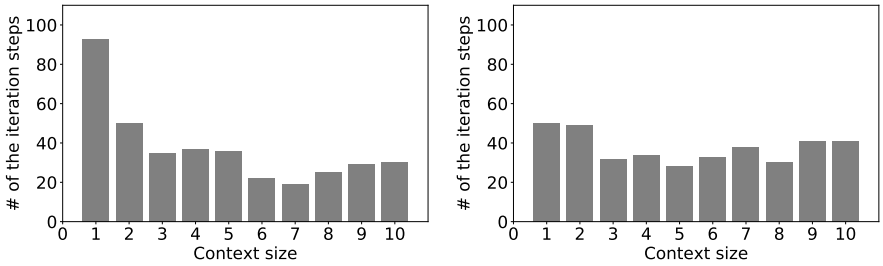
(f) NCKW Brier

Fig. 6: The results of the double Scott-Knott test on the results for each context metric in all projects. Please see text for a full explanation.



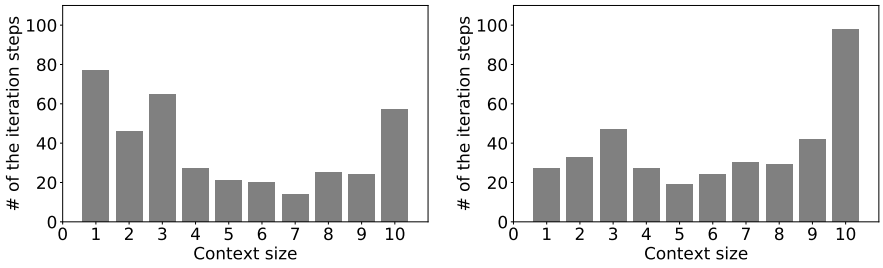
(a) NCW AUC

(b) NCKW AUC



(c) NCW MCC

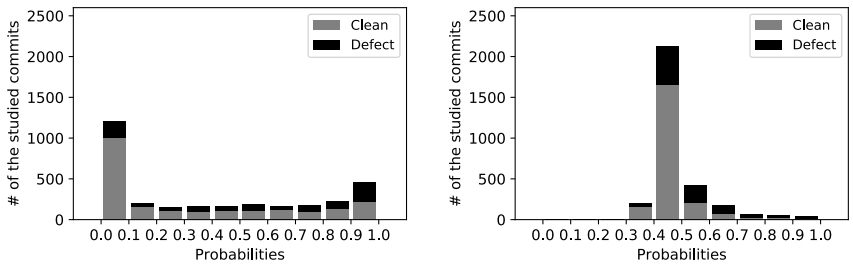
(d) NCKW MCC



(e) NCW Brier

(f) NCKW Brier

Fig. 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 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 7(a) between 1 to 10 is 376.



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

Fig. 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.

percentiles are same. Hence, we choose '+' chunk as the best chunk type for our context metrics.

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

Figure 7 shows the numbers of iteration steps that provide the best prediction performance on different context sizes. The left column of Figure 7 (Figure 7(a), 7(c) and 7(e)) shows the results for NCW with chunk type '+'. The right column of Figure 7 (Figure 7(b), 7(d) and 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 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 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 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 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.

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

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

Table 9 shows the ranks according to the Scott-Knott ESD test results of the three evaluation measures for each studied metric. Each cell includes the rank. The rank is computed across the studied metrics for each project. For example, the gray cells in Table 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 18, 19 and 20. We summarize the number of projects that are the top three ranks for each studied metric (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 paper) indicates the change metrics.

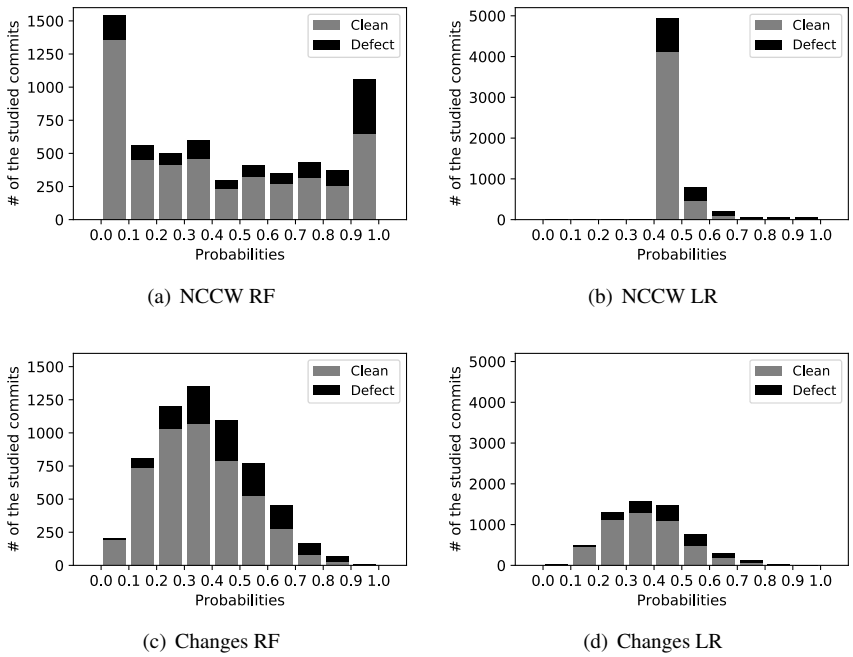
NCCW ($NCCW(c, 1, +)$) provides the top three rank prediction performance in all projects on AUC and MCC, and 5 of 6 projects on Brier score. NCCW does not provide the top one rank prediction performance on Brier score. However, this is not to be a challenge for just-in-time defect prediction. Brier 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 metrics. 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 metric NCCW has strong predicting power for just-in-time defect prediction in the studied churn metrics.

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 metrics and a churn metric LA have strong predicting power. All of the metrics 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 metrics, we conclude that AS, AB and LA have strong predicting power.

The change metrics that use all of the churn metrics 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 metrics might be close to 0.5 or the correct label than probabilities given by the other studied metrics. The probabilities that are close to 0.5 indicate that the probabilities are close to the correct label in incorrect results. Figure 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 metrics. We

Table 9: The ranks of the Scott-Knott ESD test results for studied metrics. *#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.

Metric Types	Metrics	Projects						Numbers of Ranks			
		B.	**C.**	**Ge.**	**Gi.**	**H.**	**O.**	*#R1*	*#R2*	*#R3*	*Sum*
					(a) **AUC**						
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**						
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**						
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



(a) NCCW RF (b) NCCW LR

(c) Changes RF (d) Changes LR

Fig. 9: The number of studied commits in Camel project. The x-axis refers to the probabilities using each metric on either RF (left column) or LR (right column) models.

Table 10: The values of our proposed difference of the LR model. The gray cells refer to the smallest difference values by the metrics within each project.

Metric Types	Metrics	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

can observe that when using the RF model, the probabilities that were computed by the change metrics 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 metrics. However, the mean squared differences (Brier score) of the results of the change metrics are smaller than NCCW in the half of the projects (Table 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 metric 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:

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

This is a simple variant of the Brier score.

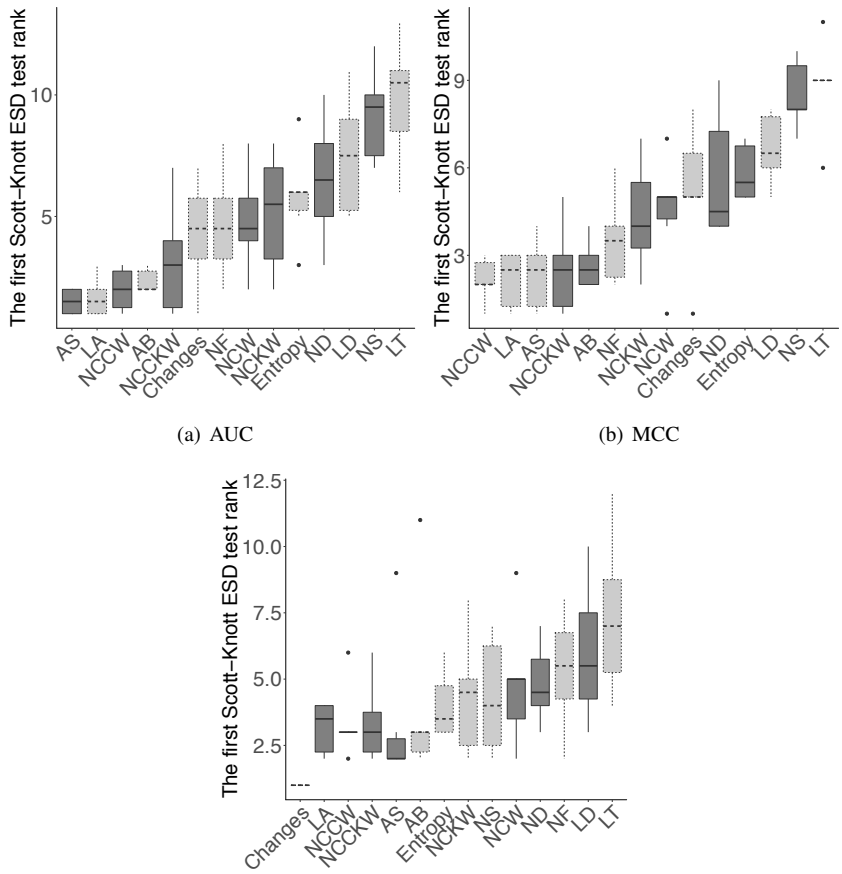
Table 10 shows the values of the difference of the LR model. The gray cells indicate the smallest values of the difference between the metrics in a project. We can observe that the change metrics achieve gray cells in the majority (5 of 6) of projects. This result implies that although probabilities that were computed by the NCCW are close to 0.5 than the change metrics, the difference of the results of the change metrics is smaller than NCCW. Hence, the probabilities are close to the correct label than NCCW. This is the reason why the change metrics provide that all projects are in the top three rank on Brier score.

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

Figure 10 shows the results of the double Scott-Knott ESD test on the results for each studied metric in all projects; each boxplot contains six ranks of the first Scott-Knott ESD test execution for the studied projects on a studied metric. 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 metric on both AUC and MCC. The change metrics are the best-performing metrics on Brier score, and AS is the second best-performing metric. This result provides that AS is a top rank metric across the studied projects on AUC and MCC, and the change metrics are the top rank metrics across the studied projects on Brier score.

The extended context metric, NCCW, and the churn metric, LA, are also better metrics 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 metric 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 metrics across the studied projects on AUC and MCC.



(a) AUC

(b) MCC

(c) Brier score

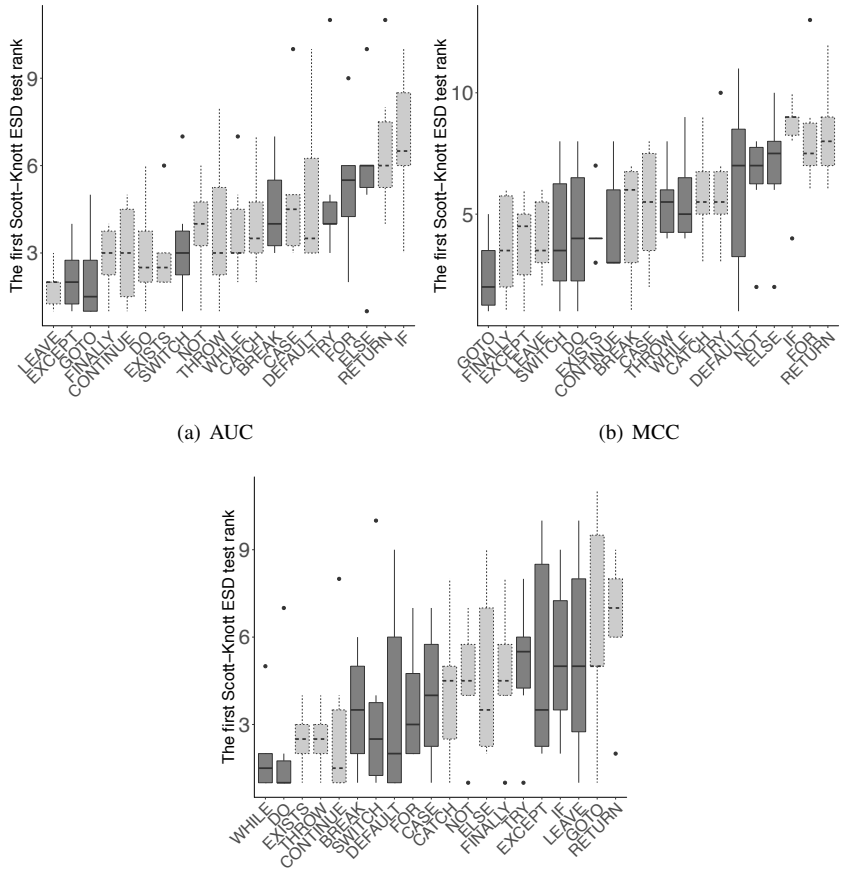
Fig. 10: The double Scott-Knott ESD test results for each studied metric in all projects. Please see text for a full explanation.

In this RQ, we study the metrics 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 metrics, and the basic predicting power of the studied metrics in Section 8 (discussion).

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

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

Figure 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



(a) AUC (b) MCC

(c) Brier score

Fig. 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.

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

Table 11: The ranks of the Scott-Knott ESD test results for studied metrics. *#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.

Metric Types	Metrics	Projects						Numbers of Ranks			
		B.	**C.**	**Ge.**	**Gi.**	**H.**	**O.**	*#R1*	*#R2*	*#R3*	*Sum*
					(a) **AUC**						
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**						
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**						
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

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 metric on the combination with NCCW. In addition, the modified NCCKW is not strongly correlated with NCCW (see Table 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 metrics as *COMB*.

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

Table 11 shows the ranks according to the Scott-Knott ESD test results of the three evaluation measures for each studied metric. 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 metrics that achieve the top-one rank prediction performance. This result indicates that COMB are the best prediction metrics in all the studied metrics. COMB achieves at least the top-three rank prediction performance for all studied projects in Brier score.

COMB statistically outperforms the other studied metrics.

Figure 12 shows the results of the double Scott-Knott ESD test on the results for each studied metric in all projects; each boxplot contains six ranks of the first Scott-Knott ESD test execution for the studied projects on a studied metric. The x-axis indicates a metric; 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.

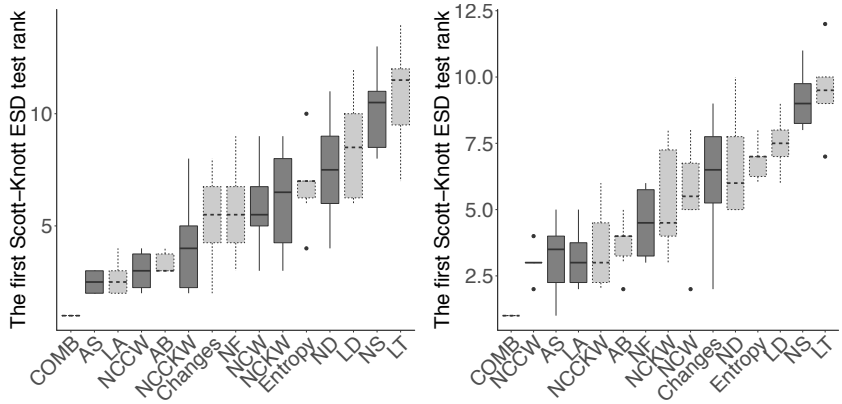
We observe that COMB are the best-performing metrics on both AUC and MCC. This result provides that COMB are the top rank metrics across the studied projects on AUC and MCC. Even on Brier score, COMB are the second rank metrics. The best-performing metrics on Brier score is still the change metrics.

8 Discussion

8.1 Are the commits identified by the context metrics different than the ones identified by the traditional churn metrics?

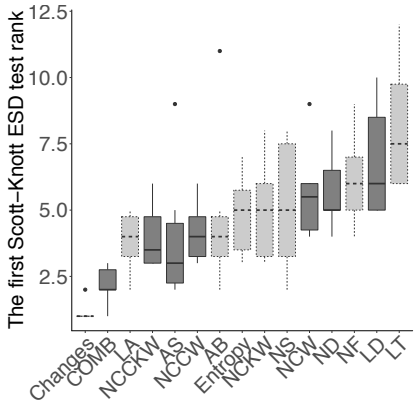
The proposed context metrics COMB identify some defective commits that other chutn metrics cannot; these commits tend to have large context lines.

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



(a) AUC

(b) MCC

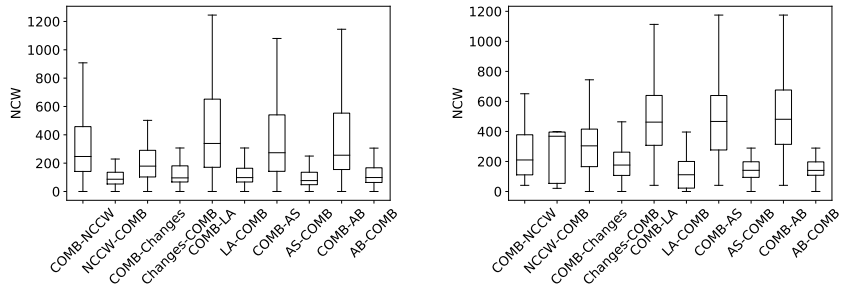


(c) Brier score

Fig. 12: The results of the double Scott-Knott ESD test on the results for each studied metric in all projects. Please see text for a full explanation.

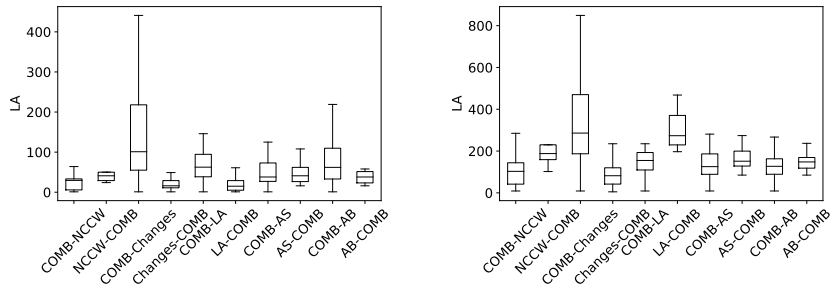
Figure 13 shows the values of the context metric NCW for the commits identified as defective in Hadoop project. We can observe that COMB identifies the commits that have higher NCW values as defective compared to the other metrics. For example, the median NCW value of COMB-Changes is higher than the median NCW value of Changes-COMB (Figure 13(a) and 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 metric.

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



(a) RF (b) LR

Fig. 13: The values of the context metric NCW for the commits identified as defective in Hadoop project. The boxplots show the cases where COMB identified the commits differently with the context metric NCCW, the change metrics, LA and the indentation metrics 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 metrics that are compared; the y-axis shows the value of NCW.



(a) Bitcoin (b) Hadoop

Fig. 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 metric NCCW, the change metrics, LA and the indentation metrics on LR model. The y-axis shows the value of LA.

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 13(a) and 13(b) indicates that COMB can uniquely identify some defective commits.

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

We observe the same tendency for the other context metrics on LR model, 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 paper. In addition, there exist commits that the traditional code churn metrics can identify that the context metrics cannot. Future studies are necessary to investigate these points.

8.2 How much do the indentation metrics improve the defect prediction performance?

Indentation metrics AS and AB have the potential to imporve defect prediction performance.

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

8.3 How redundant are the context metrics compared to the traditional metrics?

8.3.1 Motivation:

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

We present an in-depth analysis to understand the relation between our context metrics and the traditional metrics. This result produces insights of why our context metrics are not inducing redundancy, and why the context metrics can uniquely identify defective commits compared to the traditional metrics. Finally, we show the basic predicting power using *information gain* [46].

8.3.2 Approach:

We first study five context metrics (i.e., NCW, NCKW, NCCW, NCCKW, and gotoNCCKW[7]), two indentation metrics and 14 traditional change metrics based on a correlation analysis [64] and the principal component analysis (PCA) [8] to identify correlated metrics and find metrics that are important to represent the variance of the original metrics. Second, we compute information gain [46] for all the studied metrics in order to clarify the basic predicting power of the studied metrics.

We first conduct a correlation analysis on the metrics. When we use strongly correlated metrics as explanation variables for a prediction model, we get the problem

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

of multicollinearity [9]. In addition, these metrics are redundant. We use Spearman rank correlation [64] to measure the correlation between the metrics. 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 metrics which represent the highest variance of all the studied metrics. The PCA result shows which metrics can represent the variance of all the studied metrics. The PCA reduces the number of input metrics and makes new metrics. Then, the PCA shows the coefficient[8] for every new metric to convert the input metrics into the new metric. We use the coefficient of the most important new metric called the first principal component[9] to identify which metrics represent the highest variance[10]. We apply the PCA to all commits on each studied project. We suppose that metrics which represent the highest variance are important metrics in the studied metrics.

Finally, we compute information gain [46] in order to clarify the basic predicting power of the studied metrics. In our case, information gain measures the basic predicting power of each of the metrics. For example, if an original metric perfectly separates defective commits and clean commits, the value of information gain would be maximum. However, if an original metric 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 [46] is as follows:

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

where *metric* is a certain studied metric, $InfoGain(\cdot)$ is the information gain of $\cdot(metric)$, $H(\cdot)$ is Shannon entropy [48] 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 with prediction results (defective or clean).

We compute the ratio of the information gain between NCCW, and the indentation metrics and the churn metrics. Since NCCW is our proposed metric, 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 metric, NCCW has high potential to classify the commits in defect prediction rather than the certain metric.

8.3.3 Results:

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

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

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

[10] Metrics which represent higher variance of the studied metrics have higher coefficient in the first principal component.

Table 12: Spearman rank correlation between the context metrics, the indentation metrics, and the change metrics 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 metrics in this paper. These metrics are not strongly correlated (0.7 and over) with the other metrics (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 12 shows the Spearman rank correlation between all the studied metrics (including the context, the indentation and the change metrics) 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 metrics and the indentation metrics include changed lines information.

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

NCCW and NCCKW are extended metrics 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 metrics and changed lines, the context information is not redundant.

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

Table 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. 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 metric which has high coefficient in all the projects, this metric is likely to represent the highest variance of all studied metrics in all the projects.

NCCW and NCCKW include the context information and have the strong correlation to the indentation metrics and LA due to using changed line information. Hence, NCCW and NCCKW can add the context information while having the in-

Table 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

formation of the indentation metrics and LA. Hence, NCCW and NCCKW represent the highest variance.

In summary, the context metrics NCW and NCKW are not redundant metrics, and add the context information to the defect prediction model. While NCCW and NCCKW have strong correlations to the indentation metrics 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 metrics.

Figure 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 metrics. 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 metrics.

8.4 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 in such context is a good indicator of the complexity of the change (RQ1).

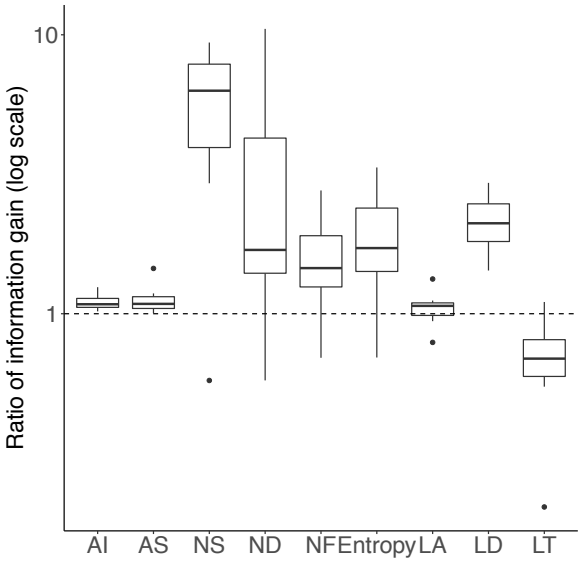


Fig. 15: The ratio of the information gain between NCCW and other metrics. The x-axis indicates the metrics that are used to compute the ratio; the y-axis indicates the ratio. The dashed line indicates that the ratio is 1.0.

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.

8.5 How are the actual AUC and MCC values of the context metrics?

We study the ranks that were computed by the Scott-Knott ESD test across the studied metrics to determine which are the best prediction metrics 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 18 and 19). From the AUC result (Table 18), COMB provides at least 0.737. This value corresponds to the strong effect size according to prior work [45]. From the MCC result (Table 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.

8.6 Practical guides (recommendations) for the parameters of the context metrics

The context metrics have two tunable parameters: the context size, and the churn type. We made our practical guides (recommendations) of optimizing the parameters of the context metrics 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 metrics of the extended context metrics that are number of words and number of "goto" keyword significantly outperform the other studied metrics. 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 metrics. COMB includes both of them. The details of how to use COMB can be found in Section 8.7.

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.

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

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

Recommendation 1: Use the indentation metric AS instead of the traditional size metrics in the change metrics. Our experiments in RQ2 show that AS significantly outperforms the other studied metrics including traditional size metrics (LA, LD and LT). In addition, AS is strongly correlated with the traditional size metric LA which has the highest performance in the change (code churn) metrics. Hence, using AS instead of the traditional size metrics 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 metrics COMB on the logistic regression model. Our experiments in RQ3 show that COMB are the best-performing metrics in AUC and MCC. In addition, our discussion shows that: (1) a context metric used in COMB, NCCW, is one of the metric that represents the highest variance of all the original metrics, and (2) the basic defect predicting power of NCCW is strong. For the interpretation of the prediction model,

Table 14: The median and 75 percentile (3Q) IQR values of the performance for the context metrics, the indentation metric and the change metrics. An IQR value is computed across all unit values for each prediction model for each project for each metric. The median/3Q IQR values are computed for each metric. 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

COMB contains only two metrics (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.

9 Threats to Validity

9.1 Construct Validity

We follow the labeling process in Commit Guru [47] in order to label each commit either defective or clean. SZZ algorithm is also a popular approach to identify defective commits [52]; 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 [54] 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 metrics. 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 metric, and (2) computing the median and 75 percentile IQR values of different unit values in all projects, prediction models, and studied metrics.

Results: Figure 16 shows the values of the evaluation measures for different unit values. We observe that all evaluation measures are stable for different unit values. In addition, we observe the same tendency for other projects, prediction models, and metrics.

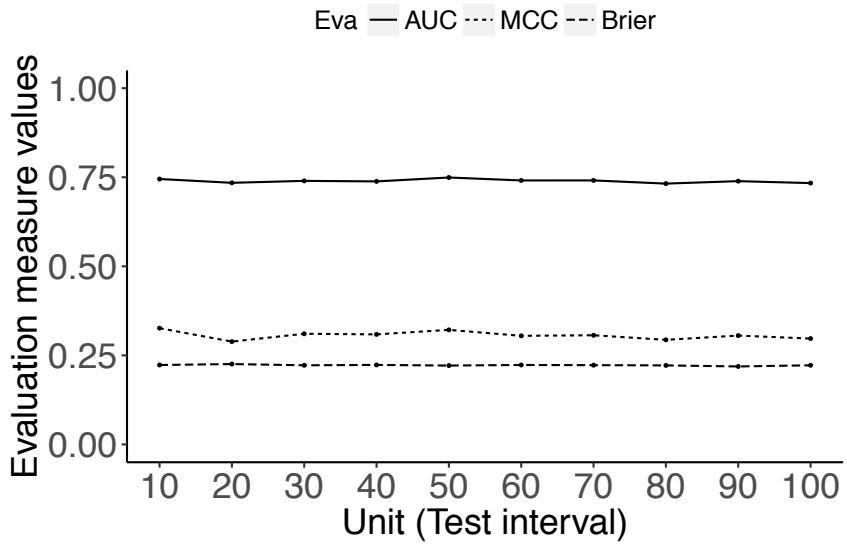


Fig. 16: The values of the evaluation measures for each unit (test interval) using the NCCW metric on LR model in the Camel project. Eva indicates evaluation measures. The x-axis indicates the unit value between 10 to 100; the y-axis indicates the values of the evaluation measures.

Table 14 shows the median and 75 percentile (3Q) IQR values for different unit values in all projects, prediction models, and studied metrics. 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.

9.2 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 metrics, extended context metrics, and combination context metrics on commercial projects for verifying our findings.

9.3 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 [4, 62] and address the pitfalls in defect prediction [57]. Hence, we believe these measures are acceptable.

10 Conclusion

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

- The chunk type '+' is the best parameter for context metrics 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 metrics, COMB, significantly outperform all the metrics, and achieve the best-performing metrics in all of the studied projects in terms of AUC and MCC.

Acknowledgment

This work was partially supported by NSERC Canada as well as JSPS KAKENHI Japan (Grant Numbers: JP16K12415).

References

1. Aversano, L., Cerulo, L., Del Grosso, C.: Learning from bug-introducing changes to prevent fault prone code. In: Proceedings of the 9th International Workshop on Principles of Software Evolution (IWPSE), pp. 19–26. ACM (2007)
2. Basili, V.R., Briand, L.C., Melo, W.L.: A validation of object-oriented design metrics as quality indicators. IEEE Transactions on Software Engineering **22**(10), 751–761 (1996)
3. Bettenburg, N., Nagappan, M., Hassan, A.E.: Think locally, act globally: Improving defect and effort prediction models. In: Proceedings of the 9th IEEE Working Conference on Mining Software Repositories (MSR), pp. 60–69. IEEE Press (2012)
4. Boughorbel, S., Jarray, F., El-Anbari, M.: Optimal classifier for imbalanced data using matthews correlation coefficient metric. PloS one **12**(6), e0177,678 (2017)

5. Bowes, D., Hall, T., Gray, D.: 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, pp. 109–118. ACM (2012)
6. Chicco, D.: Ten quick tips for machine learning in computational biology. BioData mining **10**(1), 35 (2017)
7. Cohen, J.: Statistical power analysis for the behavioral sciences (1988)
8. D'Ambros, M., Lanza, M., Robbes, R.: An extensive comparison of bug prediction approaches. In: Proceedings of the 7th Working Conference on Mining Software Repositories (MSR), pp. 31–41. IEEE (2010)
9. Farrar, D.E., Glauber, R.R.: Multicollinearity in regression analysis: the problem revisited. The Review of Economic and Statistics **49**(1), 92–107 (1967)
10. Fukushima, T., Kamei, Y., McIntosh, S., Yamashita, K., Ubayashi, N.: 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), pp. 172–181. ACM (2014)
11. Ghotra, B., McIntosh, S., Hassan, A.E.: Revisiting the impact of classification techniques on the performance of defect prediction models. In: Proceedings of the 37th International Conference on Software Engineering (ICSE), pp. 789–800. IEEE Press (2015)
12. Graves, T.L., Karr, A.F., Marron, J.S., Siy, H.: Predicting fault incidence using software change history. IEEE Transactions on software engineering **26**(7), 653–661 (2000)
13. Hall, T., Beecham, S., Bowes, D., Gray, D., Counsell, S.: A systematic literature review on fault prediction performance in software engineering. IEEE Transactions on Software Engineering **38**(6), 1276–1304 (2012)
14. Halstead, M.H.: Elements of software science. Elsevier New York (1977)
15. Han, J., Moraga, C.: The influence of the sigmoid function parameters on the speed of backpropagation learning. In: Proceedings of the International Workshop on Artificial Neural Networks, pp. 195–201. Springer (1995)
16. Hassan, A.E.: Predicting faults using the complexity of code changes. In: Proceedings of the 31st International Conference on Software Engineering (ICSE), pp. 78–88. IEEE (2009)
17. Hata, H., Mizuno, O., Kikuno, T.: Bug prediction based on fine-grained module histories. In: Proceedings of the 34th International Conference on Software Engineering (ICSE), pp. 200–210. IEEE (2012)
18. Hindle, A., Godfrey, M.W., Holt, R.C.: Reading beside the lines: Indentation as a proxy for complexity metric. In: Proceedings of the 16th International Conference on Program Comprehension (ICPC), pp. 133–142. IEEE (2008)
19. Ho, T.K.: Random decision forests. In: Proceedings of the 3rd International Conference on Document Analysis and Recognition, vol. 1, pp. 278–282. IEEE (1995)
20. Jiang, T., Tan, L., Kim, S.: Personalized defect prediction. In: Proceedings of the 28th International Conference on Automated Software Engineering (ASE), pp. 279–289. IEEE (2013)

21. Kamei, Y., Fukushima, T., McIntosh, S., Yamashita, K., Ubayashi, N., Hassan, A.E.: Studying just-in-time defect prediction using cross-project models. Empirical Software Engineering **21**(5), 2072–2106 (2016)
22. Kamei, Y., Shihab, E., Adams, B., Hassan, A.E., Mockus, A., Sinha, A., Ubayashi, N.: A large-scale empirical study of just-in-time quality assurance. IEEE Transactions on Software Engineering **39**(6), 757–773 (2013)
23. Karunanithi, N.: A neural network approach for software reliability growth modeling in the presence of code churn. In: Software Reliability Engineering, 1993. Proceedings., Fourth International Symposium on, pp. 310–317. IEEE (1993)
24. Khoshgoftaar, T.M., Allen, E.B., Goel, N., Nandi, A., McMullan, J.: Detection of software modules with high debug code churn in a very large legacy system. In: Software Reliability Engineering, 1996. Proceedings., Seventh International Symposium on, pp. 364–371. IEEE (1996)
25. Khoshgoftaar, T.M., Szabo, R.M.: Improving code churn predictions during the system test and maintenance phases. In: ICSM, vol. 94, pp. 58–67 (1994)
26. Kim, S., Whitehead Jr, E.J.: How long did it take to fix bugs? In: Proceedings of the 2006 international workshop on Mining software repositories (MSR), pp. 173–174. ACM (2006)
27. Kim, S., Whitehead Jr, E.J., Zhang, Y.: Classifying software changes: Clean or buggy? IEEE Transactions on Software Engineering **34**(2), 181–196 (2008)
28. Kim, S., Zhang, H., Wu, R., Gong, L.: Dealing with noise in defect prediction. In: Proceedings of the 33th International Conference on Software Engineering (ICSE), pp. 481–490. IEEE (2011)
29. Kim, S., Zimmermann, T., Whitehead Jr, E.J., Zeller, A.: Predicting faults from cached history. In: Proceedings of the 29th International Conference on Software Engineering (ICSE), pp. 489–498. IEEE (2007)
30. Lessmann, S., Baesens, B., Mues, C., Pietsch, S.: Benchmarking classification models for software defect prediction: A proposed framework and novel findings. IEEE Transactions on Software Engineering **34**(4), 485–496 (2008)
31. Li, J., He, P., Zhu, J., Lyu, M.R.: Software defect prediction via convolutional neural network. In: Proceedings of the 2017 Software Quality, Reliability and Security (QRS), pp. 318–328. IEEE (2017)
32. McCabe, T.J.: A complexity measure. IEEE Transactions on software Engineering **2**(4), 308–320 (1976)
33. McDonald, J.H.: Handbook of Biological Statistics (3rd ed.). Sparky House Publishing, Baltimore, Maryland. (2014)
34. Menzies, T., Milton, Z., Turhan, B., Cukic, B., Jiang, Y., Bener, A.: Defect prediction from static code features: current results, limitations, new approaches. Automated Software Engineering **17**(4), 375–407 (2010)
35. Microsoft: Overview of c++ statements (2016). URL https://docs.microsoft.com/ja-jp/cpp/cpp/overview-of-cpp-statements
36. Mizuno, O., Kikuno, T.: Training on errors experiment to detect fault-prone software modules by spam filter. In: Proceedings of the 6th Joint Meeting on Foundations of Software Engineering (ESEC/FSE), pp. 405–414. ACM (2007)
37. Mockus, A., Votta, L.G.: Identifying reasons for software changes using historic databases. In: Proceedings of the 22th International Conference on Software

Maintenance (ICSE), pp. 120–130. IEEE (2000)
38. Moser, R., Pedrycz, W., Succi, G.: 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), pp. 181–190. IEEE (2008)
39. Munson, J.C., Elbaum, S.G.: Code churn: A measure for estimating the impact of code change. In: Software Maintenance, 1998. Proceedings., International Conference on, pp. 24–31. IEEE (1998)
40. Nagappan, N., Ball, T.: Use of relative code churn measures to predict system defect density. In: Proceedings of the 27th international conference on Software engineering, pp. 284–292. ACM (2005)
41. Ohlsson, M.C., Von Mayrhauser, A., McGuire, B., Wohlin, C.: Code decay analysis of legacy software through successive releases. In: Aerospace Conference, 1999. Proceedings. 1999 IEEE, vol. 5, pp. 69–81. IEEE (1999)
42. Oram, A., Wilson, G.: Making software: What really works, and why we believe it. " O'Reilly Media, Inc." (2010)
43. Ostrand, T.J., Weyuker, E.J., Bell, R.M.: Where the bugs are. In: ACM SIGSOFT Software Engineering Notes, vol. 29, pp. 86–96. ACM (2004)
44. Quinlan, R.: C4.5: Programs for Machine Learning. Morgan Kaufmann Publishers (1993)
45. Rice, M.E., Harris, G.T.: Comparing effect sizes in follow-up studies: Roc area, cohen's d, and r. Law and human behavior **29**(5), 615–620 (2005)
46. Romanski, P., Kotthoff, L.: Fselector
47. Rosen, C., Grawi, B., Shihab, E.: Commit guru: Analytics and risk prediction of software commits. In: Proceedings of the 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE, pp. 966–969. ACM (2015)
48. Shannon, C.E.: A mathematical theory of communication. Bell system technical journal **27**(3), 379–423 (1948)
49. Shepperd, M., Bowes, D., Hall, T.: Researcher bias: The use of machine learning in software defect prediction. IEEE Transactions on Software Engineering **40**(6), 603–616 (2014)
50. Shihab, E.: An exploration of challenges limiting pragmatic software defect prediction. Ph.D. thesis, Queen's University (Canada) (2012)
51. Shihab, E., Hassan, A.E., Adams, B., Jiang, Z.M.: An industrial study on the risk of software changes. In: Proceedings of the 20th International Symposium on the Foundations of Software Engineering (FSE), p. 62. ACM (2012)
52. Śliwerski, J., Zimmermann, T., Zeller, A.: When do changes induce fixes? In: Proceedings of the 2th International Workshop on Mining Software Repositories (MSR), 4, pp. 1–5. ACM (2005)
53. Stevenson, A., Lindberg, C.A.: New Oxford American Dictionary. Oxford University Press (2010)
54. Tan, M., Tan, L., Dara, S., Mayeux, C.: Online defect prediction for imbalanced data. In: Proceedings of the 37th International Conference on Software Engineering (ICSE), pp. 99–108. IEEE (2015)
55. Tantithamthavorn, C., Hassan, A.E.: 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-SEIP18), p. To Appear (2018)
56. Tantithamthavorn, C., McIntosh, S., Hassan, A.E., Matsumoto, K.: Automated parameter optimization of classification techniques for defect prediction models. In: Proceedings of the 38th International Conference on Software Engineering (ICSE), pp. 321–332. ACM (2016)
57. Tantithamthavorn, C., McIntosh, S., Hassan, A.E., Matsumoto, K.: An empirical comparison of model validation techniques for defect prediction models. IEEE Transactions on Software Engineering **43**(1), 1–18 (2017)
58. Tassey, G.: The economic impacts of inadequate infrastructure for software testing. National Institute of Standards and Technology (2002)
59. Thomas W., S.: lscp: A lightweight source code preprocesser (2015)
60. Wang, S., Liu, T., Tan, L.: Automatically learning semantic features for defect prediction. In: Proceedings of the 38th International Conference on Software Engineering (ICSE), pp. 297–308. ACM (2016)
61. Yang, X., Lo, D., Xia, X., Zhang, Y., Sun, J.: Deep learning for just-in-time defect prediction. In: Proceedings of the 2015 Software Quality, Reliability and Security (QRS), pp. 17–26. IEEE (2015)
62. Zhang, F., Zheng, Q., Zou, Y., Hassan, A.E.: Cross-project defect prediction using a connectivity-based unsupervised classifier. In: Proceedings of the 38th International Conference on Software Engineering (ICSE), pp. 309–320. ACM (2016)
63. Zimmermann, T., Premraj, R., Zeller, A.: Predicting defects for eclipse. In: Proceedings of the 3th International Workshop on Predictor Models in Software Engineering (PROMISE), pp. 9–19. IEEE (2007)
64. Zwillinger, D., Kokoska, S.: CRC standard probability and statistics tables and formulae. Crc Press (1999)

A 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 15: The median values of AUC for each context metric 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).

Metrics	Chunk Types	Projects					
		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 16: The median values of MCC for each context metric 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).

Metrics	Chunk Types	Projects					
		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 17: The median values of Brier score for each context metric 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).

Metrics	Chunk Types	Projects					
		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 18: The median values of AUC for each context metric, each extended context metric, COMB, each indentation metric, the change metrics and each of the change metrics. Each cell indicates the values of AUC by RF (left) and LR (right) models, respectively.

Metric Types	Metrics	Projects					
		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 19: The median values of MCC for each context metric, each extended context metric, COMB, each indentation metric, the change metrics and each of the change metrics. Each cell indicates the values of MCC by RF (left) and LR (right) models, respectively.

Metric Types	Metrics	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 20: The median values of Brier score for each context metric, each extended context metric, COMB, each indentation metric, the change metrics and each of the change metrics. Each cell indicates the values of Brier score by RF (left) and LR (right) models, respectively.

Metric Types	Metrics	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 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

53