Deep Just-in-Time Defect Prediction: How Far Are We?

Zhengran Zeng
Southern University of Science and Technology
China
12032889@mail.sustech.edu.cn

Yuqun Zhang*
Southern University of Science and Technology
China
zhangyq@sustech.edu.cn

Haotian Zhang
Kwai Inc.
China
zhanghaotian@kuaishou.com

Lingming Zhang
University of Illinois Urbana-Champaign
United States
lingming@illinois.edu

ABSTRACT

Defect prediction aims to automatically identify potential defective code with minimal human intervention and has been widely studied in the literature. Just-in-Time (JIT) defect prediction focuses on program changes rather than whole programs, and has been widely adopted in continuous testing. CC2Vec, state-of-the-art JIT defect prediction tool, first constructs a hierarchical attention network (HAN) to learn distributed vector representations of both code additions and deletions, and then concatenates them with two other embedding vectors representing commit messages and overall code changes extracted by the existing DeepJIT approach to train a model for predicting whether a given commit is defective. Although CC2Vec has been shown to be the state of the art for JIT defect prediction, it was only evaluated on a limited dataset and not compared with all representative baselines. Therefore, to further investigate the efficacy and limitations of CC2Vec, this paper performs an extensive study of CC2Vec on a large-scale dataset with over 310,370 changes (8.3 X larger than the original CC2Vec dataset). More specifically, we also empirically compare CC2Vec against DeepJIT and representative traditional JIT defect prediction techniques. The experimental results show that CC2Vec cannot consistently outperform DeepJIT, and neither of them can consistently outperform traditional JIT defect prediction. We also investigate the impact of individual traditional defect prediction features and find that the added-line-number feature outperforms other traditional features. Inspired by this finding, we construct a simplistic JIT defect prediction approach which simply adopts the added-line-number feature with the logistic regression classifier. Surprisingly, such a simplistic approach can outperform CC2Vec and DeepJIT in defect prediction, and can be 81k X/120k X faster in training/testing. Furthermore, the paper also provides various practical guidelines for advancing JIT defect prediction in the near future.

CCS CONCEPTS

• **Software and its engineering** → **Software defect analysis**.

*Yuqun Zhang is the corresponding author.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ISSTA '21, July 11–17, 2021, Virtual, Denmark
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-8459-9/21/07…$15.00
https://doi.org/10.1145/3460319.3464819

KEYWORDS

Software Defect Prediction, Just-In-Time Prediction, Deep Learning

ACM Reference Format:
Zhengran Zeng, Yuqun Zhang, Haotian Zhang, and Lingming Zhang. 2021. Deep Just-in-Time Defect Prediction: How Far Are We?. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '21), July 11–17, 2021, Virtual, Denmark.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3460319.3464819

1 INTRODUCTION

Software development nowadays has been increasingly involved with rapidly changing requirements, diverse and evolving infrastructures, and fierce peer pressure among developers, all of which together can raise intense time and resource constraints on software quality assurance tasks. Therefore, it is essential to identify program defects as early as possible such that developers/testers can focus on specific program scopes to reduce the overall testing and debugging time and efforts. To this end, Just-in-Time (JIT) defect prediction approaches have been proposed [32] to provide early predictions about likely program defects during software evolution. In particular, compared with traditional defect prediction approaches which mainly operate at the file or module level [10, 27, 45, 46], JIT defect prediction approaches work at the fine-grained code-change level to provide precise hints about potential defects.

Among the existing general-purpose JIT defect prediction approaches, CC2Vec, proposed at ICSE'20 [19], has been demonstrated to be the state of the art. CC2Vec basically learns the vector representations for code additions and deletions, and then integrates that with the previous DeepJIT approach [18] to construct a deep neural network for JIT defect prediction. More specifically, CC2Vec first applies a hierarchical attention network (HAN) [51] to deliver the distributed vector representations of both code additions and deletions. Meanwhile, the previous DeepJIT approach is leveraged to obtain two embedding vectors representing commit messages and overall code changes via convolutional neural networks (CNNs) [25]. Next, the resulting vectors of CC2Vec and DeepJIT are concatenated and input to a fully-connected network for final defect prediction. Taking two open-source projects with a total of 37k commits (with 3.6k defects) , CC2Vec can outperform DeepJIT by over 7% in terms of the AUC score.

Meanwhile, despite its advanced design, the effectiveness of CC2Vec is still unclear because (1) CC2Vec was only evaluated upon a limited dataset with marginal improvements (i.e., around 7% in terms of the AUC [26] score) against the previous state-of-the-art approach DeepJIT; (2) CC2Vec is built on deep neural networks and thus relies on training data quality; and (3) there is no comprehensive comparison between CC2Vec (or DeepJIT) and other representative traditional approaches. Therefore, in this paper, to

fully understand the efficacy and limitations of the general-purpose deep-learning-based JIT defect prediction approaches, we construct a large-scale dataset by extending the original one to include the latest program versions and introducing additional popular open-source projects. As a result, we collect 310k commits from six well-known open-source projects. To the best of our knowledge, this is so far the largest evaluation in the literature on general-purpose deep-learning-based JIT defect prediction.

The experimental results show that overall the performance of DeepJIT and CC2Vec can be replicated on their original dataset. However, surprisingly, the GitHub version of DeepJIT, which abstracts all the detailed code changes into simple labels (i.e., "added_code/ removed_code"), performs better than the DeepJIT version described in the original DeepJIT paper, which leverages the detailed code-change information. This indicates that feeding detailed information into machine learning models does not always help with JIT defect prediction. Our experimental results on the extended dataset further show that CC2Vec fails to outperform DeepJIT under most studied projects. Also, we find that neither DeepJIT nor CC2Vec can consistently outperform traditional JIT defect prediction on all the studied projects under either within-project or cross-project prediction scenarios. Therefore, we further investigate how different popular traditional defect prediction features can impact the performance of JIT defect prediction. Interestingly, the experimental results show that the added-line-number feature (which simply computes the number of added lines of code in each commit) can provide the higher prediction accuracy than all other studied features.

At last, inspired by the findings of our study, we construct a simplistic and fast approach, *LApredict*, which simply adopts the added-line-number feature with the logistic regression classifier for JIT defect prediction. We also compare *LApredict* with all the studied approaches on top of the extended dataset under both within-project and cross-project scenarios. Surprisingly, the results show that *LApredict* can outperform CC2Vec and all other existing approaches for most cases in JIT defect prediction, and can be 81k X/120k X faster than CC2Vec in training/testing. Furthermore, the superiority of *LApredict* is even enlarged in cross-project validation. Accordingly, our study also reveal various practical guidelines for how to further advance JIT defect prediction in the near future. In summary, this paper makes the following contributions.

- **Dataset.** An extensive dataset with 310k commits, collected from all program versions of six popular projects over the last 10 years. Such a dataset provides not only a much larger amount of real-world software defects but also their distributions over a larger time span, which can potentially benefit and impact all future JIT defect prediction research.
- **Study.** An extensive study of state-of-the-art general-purpose JIT defect prediction approaches (CC2Vec and DeepJIT) on the proposed extended dataset, with detailed quantitative and qualitative analysis on their strengths and limitations.
- **Technique.** A simplistic and fast approach, *LApredict*, which simply adopts the added-line-number feature with the traditional logistic regression classifier instead of building complex deep neural networks for JIT defect prediction.

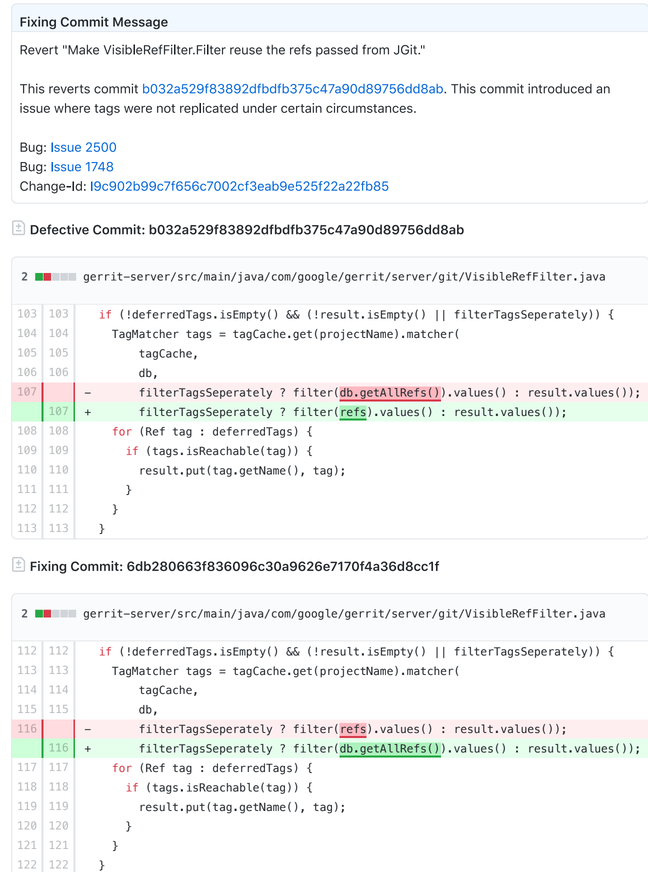


Figure 1: An illustrative example

- **Implications.** An empirical evaluation of *LApredict*, state-of-the-art CC2Vec and DeepJIT, and two other representative JIT defect prediction approaches, indicating that simplistic/-traditional approaches/features can easily outperform the advanced deep-learning-based approaches and be 81k X/120k X faster in training/testing. The paper also reveals various practical guidelines for future JIT defect prediction.

The replication package for this paper, including all our data, source code, and documentation, is publicly available online at:

https://github.com/ZZR0/ISSTA21-JIT-DP

2 BACKGROUND

2.1 Deep JIT Defect Prediction: Example

In this section, we present a defective commit (commit b032a5) and its fixing commit (commit 6db280) of project Gerrit [2] in Figure 1 to demonstrate the challenges of JIT defect prediction. Specifically in commit b032a5, the developer replaced a database request operation with a parameter passed from JGit, e.g., "`db.getAllRefs()`" (Line 107) is replaced with variable "`refs`". It was until two years later when the developers discovered that commit b032a5 introduced an issue that "`refs`" and "`db.getAllRefs()`" are not always equivalent, i.e., "`db.getAllRefs()`" returns the whole "`Refs`" object of "`db`" while "`refs`" can only return partial "`Refs`" object.

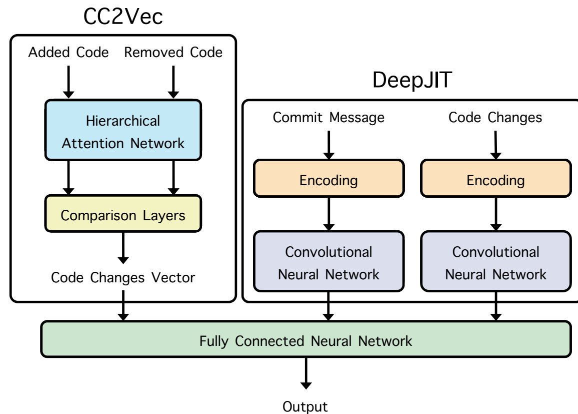


Figure 2: The overall framework of CC2Vec + DeepJIT

Therefore, commit 6db280 was further made to replace "`refs`" back with "`db.getAllRefs()`".

From the example, we can observe that the defect refers to an only one-line discrepancy of the source code between the two commits, which can be rather challenging to predict automatically. Many existing approaches perform JIT defect prediction based on classic features (e.g., the number of modified files and the file size) and traditional machine learning techniques (e.g., logistic regression [12]). However, traditional techniques can hardly learn useful information for all possible cases, e.g., simply extracting the traditional syntactic features for comparative analysis can hardly work for this example with a single-line change. Intuitively, to identify such defects, it is helpful to understand their specific program semantics, e.g., investigating program control/data flows. However, such analysis can be rather challenging for complex programs and involve intensive manual efforts. Therefore, recently, researchers have leveraged deep learning models to learn more useful semantic information for precise JIT defect prediction [18, 19]. For example, the deep learning models can potentially learn from other historical defects that the change in commit b032a5 can be dangerous.

2.2 State-of-the-Art CC2Vec

Figure 2 presents the overall neural network structure for CC2Vec [19]. Specifically, as the state-of-the-art general-purpose JIT defect prediction approach, CC2Vec constructs a hierarchical attention network (HAN) [51] to embed added and deleted code of each changed file associated with one given commit respectively. Specifically, the adopted HAN first builds vector representations for lines, and further uses them to build vector representations of hunks. At last, such hunk vector representations are aggregated to construct the embedding vectors corresponding to the added or removed code.

CC2Vec further adopts multiple comparison functions to capture the difference between the derived embedding vectors of the added and removed code for exploring their relationship, where each comparison function can produce a vector. Next, all the resulting vectors are concatenated into one embedding vector to represent the file-level added/removed code changes. Eventually, all the embedding vectors associated with all the changed files under one commit are aggregated as the *distributed* vector representations of the code additions and deletions.

In addition, CC2Vec also adopts a previous JIT defect prediction approach, DeepJIT [18], to strengthen the prediction efficacy. DeepJIT constructs two convolutional neural networks (CNNs) [25] to extract the features of (1) commit messages and (2) overall code changes of a given commit. Note that, the DeepJIT code-change feature extraction is totally different from CC2Vec and can be complementary: DeepJIT simply aggregates added/removed code as one single input, and thus ignores the *distributed* information between code additions and deletions considered by CC2Vec (which treats added/removed code as two different inputs for HAN).

Eventually, all the vectors derived by CC2Vec and DeepJIT are input to a fully-connected layer for computing the likelihood that the given commit incurs a defect.

Although CC2Vec attempts to leverage the semantic information out of code changes for facilitating the JIT defect prediction efficacy, the adopted input information is still limited, i.e., only the textual code changes and their associated commit messages are considered. On the other hand, there is no detailed exploration for validating the actual contributions of different adopted components. Moreover, the study of CC2Vec only considers two open-source projects QT [5] and OpenStack [4] with a total of 37k commits. Such a dataset may be too limited to demonstrate the generalizability and scalability of CC2Vec. Lastly, the program versions of the adopted projects are a bit outdated, i.e., QT contains data from June 2011 to March 2014 while OpenStack contains data from November 2011 to February 2014, making the evaluation incapable of demonstrating whether the findings vary over time since no recent data are included.

3 STUDY ON DEEP JIT DEFECT PREDICTION

As discussed in Section 2.2, state-of-the-art CC2Vec was only evaluated upon a limited dataset against DeepJIT. Therefore, it is essential to extend the efficacy evaluation of CC2Vec (and DeepJIT) to a more diverse and larger dataset with more representative baseline approaches to thoroughly understand deep JIT defect prediction.

3.1 Dataset Collection

In general, we attempt to collect influential open-source projects for conducting our study. In particular, since one of our study tasks is to replicate the evaluations of CC2Vec, we first retain all the projects used in the evaluations of the original CC2Vec paper, i.e., QT and OpenStack. Next, we choose to extend the existing dataset by including the projects which are not only influential but also diverse from the existing ones. To this end, we also adopt Eclipse JDT and Platform [1], which haven been widely adopted by prior defect prediction work [20, 21, 49] and are programmed mainly in Java (while QT and OpenStack are mainly in C++). Moreover, we also include project Gerrit [2], a professional code review tool which has been adopted by many commercial and open-source projects for their code review process[53]. Lastly, we select project Go [3] for our study since it is a representative and popular modern programming language. In this way, we have a diverse set of real-world projects in different programming languages.

Furthermore, for each studied project, we collect all its code commits between 2011-01-01 and 2020-12-01 (which subsumes the time range covered by the CC2Vec dataset) to explore how JIT defect prediction results vary across time. Specifically, following prior

Table 1: Dataset statistics

Project	# Changes	% Defect	Median Defect Fix Delay (days)	Language
QT	95758	15.16	526.29	C++
OpenStack	66065	31.68	280.06	C++
JDT	13348	41.20	251.28	Java
Platform	39365	37.74	259.01	Java
Gerrit	34610	8.64	294.58	Java
Go	61224	36.75	243.21	Golang

work [18, 19, 31], we apply the widely-used SZZ algorithm [22, 41] to identify the defective commits. First, we identify defect-fixing commits by analyzing their commit messages. Next, we locate the modified lines of the defect-fixing commits using the *Diff* command. Furthermore, we use the *Blame* command to detect the commits inducing the modified lines as defective commits. Meanwhile, following prior work [31], all other commits unmarked by the SZZ algorithm will be treated as the correct instances for training/testing. At last, we filter such commits based on the procedure by Kim et al. [22] and McIntosh et al. [31] to remove whitespace commits, comment commits, merged commits, and other suspicious commits. Note that we also eliminate the authentication latency issue [6] by following prior work [6, 31, 42], e.g., we remove recent data according to Column "Median Defect Fix Delay (days)" in Table 1, which represents the median time interval for the defect from appearance to be fixed. The reason is that many defects from recent data have not been fully detected/fixed yet, and may affect our study. Finally, we obtain a dataset with six projects and 310,370 total real-world commits including 81,300 defects, enabling an extensive evaluation and study of CC2Vec. Table 1 presents the detailed dataset statistics. To the best of our knowledge, this is the largest dataset for general-purpose deep-learning-based JIT defect prediction.

3.2 Research Questions

We investigate the following research questions for studying CC2Vec:

- **RQ1:** *Why do DeepJIT and CC2Vec work?* For this RQ, we explore what makes the deep learning models effective in JIT defect prediction. In particular, we reproduce the experiments in the DeepJIT [18] and CC2Vec [19] papers and conduct a more detailed analysis of their adopted input components than the original papers, including exploring each input component's contribution to the overall models.
- **RQ2:** *How do DeepJIT and CC2Vec perform on the extended dataset?* For this RQ, we conduct an extensive study for the performance comparison between CC2Vec, DeepJIT, and other representative traditional JIT defect prediction approaches upon the extended dataset. We also investigate not only the AUC score adopted in the original study, but also other widely used metrics for prediction effectiveness.
- **RQ3:** *How do traditional defect prediction features perform for JIT defect prediction?* For this RQ, we adopt the features widely used in traditional JIT defect prediction and investigate their respective performance impact.
- **RQ4:** *Can a simplistic approach without deep learning outperform DeepJIT/CC2Vec for JIT defect prediction?* For this RQ, we attempt to design/implement a simplistic but effective JIT defect prediction approach without deep learning.

```
Commit Id: "0041b2267144f215fee9a6c4c99739e0559a527b"
Message: "remove qtalgorithms usage from qt designer ..."
⌄ Code Change: Array
    0: "added _ code removed _ code"
    1: "added _ code removed _ code"
    2: "added _ code removed _ code"
    3: "added _ code removed _ code"
    4: "added _ code removed _ code"
    5: "added _ code removed _ code"
    6: "added _ code removed _ code"
```

Figure 3: DeepJIT example with abstracted code changes

3.3 Results and Analysis

3.3.1 RQ1: Why Do DeepJIT and CC2Vec Work? For this RQ, we replicate the experiments on DeepJIT and CC2Vec under exactly the same setting and dataset of their original papers [18, 19]. We also explore how the three different inputs of CC2Vec (the DeepJIT code-change vector, the DeepJIT commit-message vector, and the CC2Vec code-change vector) impact JIT defect prediction.

Note that after carefully inspecting the source code provided by the DeepJIT and CC2Vec GitHub pages, we find that instead of using the vector representations of detailed code changes as declared in the original DeepJIT paper, DeepJIT and the DeepJIT component in CC2Vec actually abstract each changed file within one commit into a simple "added_code/removed_code" label (with a maximum of 10 labels for each commit), as shown in Figure 3. Therefore, we also collect the vector representations of the detailed code changes according to the original DeepJIT paper and implement them for both DeepJIT and the DeepJIT component in CC2Vec. Eventually, we end up with a complete replication study on DeepJIT and CC2Vec by involving not only reusing the provided source code on GitHub but also reimplementing their original versions described in the original papers (i.e., we modify the "added_code/removed_code" labels back to the detailed source code they represent for DeepJIT).

Table 2 shows our reproduced experimental results in terms of AUC scores. Note that considering the model randomness, we run all our experiments for 16 runs as recommended by prior work [36], and present the mean, min, max, and standard deviation values across all 16 runs. In the table, DeepJIT$_{GitHub}$ refers to the DeepJIT version using the "added_code/removed_code" labels (i.e., consistent with the original DeepJIT implementation on GitHub) and DeepJIT$_{Paper}$ refers to the DeepJIT version vectorizing the detailed given code changes (i.e., consistent with the original DeepJIT paper description). Also, we denote DeepJIT$_{GitHub}$+CC2Vec as CC2Vec$_{GitHub}$, denote DeepJIT$_{Paper}$+CC2Vec as CC2Vec$_{Paper}$. We can observe that in general, the original experimental results can be replicated. For example, the Mean AUC score is 0.7705 for DeepJIT$_{GitHub}$ and 0.7841 for DeepJIT$_{GitHub}$+CC2Vec in our reproduced results, while they were 0.7595 and 0.8155, respectively, in the original papers[1]. Interestingly, the standard deviations of 16 identical runs are less than 0.0030 and the differences between the minimum and maximum AUC values are no greater than 0.0094 for all the replicated techniques. We have similar findings for all our subsequent experiments. Therefore, due to such rather stable results, we only show the experimental results for one run for all our subsequent experiments.

[1]Note that considering the randomness in the 5-fold cross validation process used by the original papers and the discrepancies on the execution environments, such performance discrepancies can be tolerated and neglected.

Table 2: Replication study results for DeepJIT and CC2Vec

AUC Score	QT				Openstack				Mean			
	Mean	Min	Max	S.D.	Mean	Min	Max	S.D.	Mean	Min	Max	S.D.
DeepJIT$_{GitHub}$	.7959	.7940	.7979	.0012	.7452	.7429	.7472	.0015	.7705	.7691	.7719	.0009
DeepJIT$_{Paper}$	.7724	.7683	.7756	.0021	.7152	.7124	.7218	.0026	.7438	.7417	.7465	.0014
CC2Vec$_{GitHub}$	.8118	.8106	.8129	.0008	.7564	.7539	.7579	.0012	.7841	.7823	.7852	.0008
CC2Vec$_{Paper}$	.7788	.7780	.7805	.0007	.7378	.7328	.7404	.0018	.7583	.7566	.7601	.0008

DeepJIT code-change vector effectiveness. We can further observe that the performance of DeepJIT$_{GitHub}$ is slightly better than DeepJIT$_{Paper}$. For instance, 0.7959 vs. 0.7724 on QT and 0.7452 vs. 0.7152 on OpenStack. We also have similar observations for DeepJIT$_{GitHub}$ + CC2Vec and DeepJIT$_{Paper}$ + CC2Vec (0.8118 vs. 0.7788 on QT and 0.7564 vs. 0.7378 on OpenStack). Such observations indicate that the "added_code/removed_code" labels, which are literally simply the number of changed files, can outperform the detailed code-change vectors described in the original DeepJIT paper for JIT defect prediction. This finding is rather surprising to us, as it implies that while the power of the code-change semantics can be leveraged by deep learning techniques, a simple traditional feature (e.g., the number of changed files) can be more helpful in JIT defect prediction! Note that based on such experimental results, for all the following studies, we would apply DeepJIT$_{GitHub}$ and its associated combinations with CC2Vec, as DeepJIT$_{GitHub}$ not only performs better but also was the version used to produce the experimental results in the original CC2Vec and DeepJIT papers.

Finding 1: The performance of DeepJIT and CC2Vec can overall be replicated on the originally adopted benchmark projects (QT and OpenStack). Interestingly, the two DeepJIT versions perform differently: the GitHub version which abstracts detailed code changes (DeepJIT$_{GitHub}$) outperforms the original-paper version which leverages the detailed code-change semantics (DeepJIT$_{Paper}$) on both two studied projects.

DeepJIT commit-message vector effectiveness. Next, we attempt to infer how the commit messages can advance the performance of DeepJIT and CC2Vec. In particular, we use the Grad-CAM (Class Activation Mapping) algorithm [39] which is widely used for visual analysis of neural network model inputs. In particular, since the last layer of the cumulative convolution layer in a CNN model contains the richest spatial and semantic information, the Grad-CAM algorithm can weight and sum the last layer feature maps of the CNN model to deliver the contribution of each word to the model's output. Accordingly, we can derive which words in the commit messages significantly impact the model's prediction results. For instance, Figure 4a demonstrates a true positive example where DeepJIT derives that it is 80.39% a defect. Then, we calculate each word's contribution to the predicted results using the Grad-CAM algorithm and mark them under colors, where darker colors indicate more significant contributions of the associated words. We can observe that "task-number" and "qtbug-27968" make the largest contributions to the prediction result. In QT, "task-number" is always followed by a task ID which indicates this commit may be a bug fix or a feature commit. Actually, many previous studies [14, 37] have found that fixing a bug or adding a new feature is likely to result in a bug. Therefore, we can infer that DeepJIT identifies "task-number" and "qtbug-27968" to be useful and highlight their impact for defect prediction. We can also strengthen such finding via the other two examples from Figure 4b and Figure 4c, where "task-number" in Figure 4b and "fixes" in Figure 4c cause the examples to be classified as defective.

Table 3: Word rank by contribution

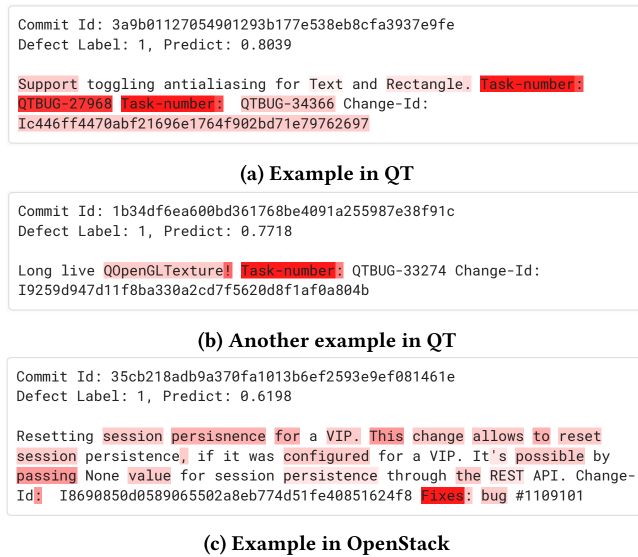
QT		OpenStack	
Rank	Word	Rank	Word
18	task-number	39	failures
443	fix	96	resolves
532	bug	474	fail
643	failures	693	bugs

We further derive the impact rank of the words (which appear more than once in QT and OpenStack) on the defect prediction results according to their average Grad-CAM scores. Table 3 presents the results of the words we consider to be relevant to the intents of their associated programs. We can observe that although some of them, e.g., "task-number" in QT and "failures" in OpenStack, rank relatively high, the others' rankings are rather less distinctive. Such results indicate that while DeepJIT can identify the importance of the words associated with the program intents under certain circumstances, its overall effectiveness can be nevertheless compromised by many other words, making the contributions of the commit-message feature somewhat limited.

Finding 2: DeepJIT and CC2Vec can extract the intent of code changes from commit messages to assist defect prediction under certain circumstances.

CC2Vec code-change vector effectiveness. Furthermore, we attempt to investigate the effect posed by the code-change vectors extracted by CC2Vec. Specifically, CC2Vec is designed to leverage its HAN model structure to extract code-change semantics information to further boost DeepJIT. However, we can observe that DeepJIT+CC2Vec only leads to limited performance improvement over DeepJIT, i.e., 1.99% and 1.50% on QT and OpenStack, respectively. From such results, we can infer that the code-change semantics information extracted by CC2Vec may have limited effectiveness for JIT defect prediction, and will further verify it in our extended experimental settings (Section 3.3.2).

Ablation study. At last, we attempt to investigate the impact of each individual feature input adopted by CC2Vec. To this end, we choose to adopt/remove only one DeepJIT/CC2Vec input feature at a time to train the deep JIT model for evaluating their respective effectiveness, as presented in Table 4. In the table, CC2Vec$_{Code}$, DeepJIT$_{Code}$, and DeepJIT$_{Msg}$ represent the CC2Vec code-change input, the DeepJIT code-change input, and the DeepJIT commit-message input, respectively. The last three rows present the results



(c) Example in OpenStack

Figure 4: Class Activation Mapping (CAM) examples

Table 4: The impact of different feature subsets for CC2Vec

Input	QT	OpenStack	Mean
CC2Vec_{Code}	.4881	.4849	.4865
DeepJIT_{Msg}	.7044	.6833	.6938
DeepJIT_{Code}	.7126	.7194	.7160
-CC2Vec_{Code}	.7780	.7562	.7671
-DeepJIT_{Msg}	.7075	.7166	.7120
-DeepJIT_{Code}	.6989	.6853	.6921

when removing each of the three feature inputs, e.g., "-CC2Vec_{Code}" denotes removing CC2Vec_{Code} from CC2Vec. From the table, we can observe that using individual inputs can reduce the prediction accuracy in terms of the AUC scores. For example, the average AUC of CC2Vec_{Code}, DeepJIT_{Code}, and DeepJIT_{Msg} is respectively 0.4865, 0.6938, and 0.7160. Meanwhile, removing each feature input can also reduce the prediction accuracy of CC2Vec. For example, the average AUC when removing CC2Vec_{Code}, DeepJIT_{Code}, and DeepJIT_{Msg} is respectively 0.7671, 0.7120, and 0.6921. From such results, we can also infer that the vector representations of commit messages and code changes extracted by DeepJIT (i.e., DeepJIT_{Msg} and DeepJIT_{Code}) can make more contributions than the CC2Vec code-change vector representation (i.e., CC2Vec_{Code}) for deep JIT defect prediction.

> *Finding 3: The DeepJIT commit-message/code-change vector representations contribute more for deep JIT defect prediction than the CC2Vec code-change vector representations.*

3.3.2 RQ2: How Do DeepJIT and CC2Vec Perform on the Extended Dataset? In particular, we investigate how they perform under both the within-project and cross-project JIT defect prediction scenarios on top of the extended dataset. We also choose to conduct the experiments on the *late* versions of the adopted benchmark projects (partially for experimental comparison with the previous studies on *early* versions). Note that to eliminate the authentication latency issues [6], we remove certain recent data according to the defect fix delay time. As a result, we select the latest three-year commits, e.g., commits from 2015-01-01 to 2018-01-01 for QT and from 2016-01-01 to 2019-01-01 for OpenStack. Moreover, our training data and testing data are extracted chronologically and follow the same training/testing data partitioning of the original DeepJIT and CC2Vec work, i.e., the earlier 80% data are used for training and the later 20% data are used for testing.

Furthermore, we choose to extend the performance comparison between CC2Vec, DeepJIT, and other representative approaches. To this end, we first retain DBN-JIT [50] which uses Deep Belief Network [17] to extract high-level information from the traditional defect prediction features and is adopted as the evaluation baseline in the original DeepJIT paper. Moreover, we also adopt a classic approach proposed by Kamei et al. [21] (denoted as LR-JIT), which integrates manually extracted features with the logistic regression model and has been widely adopted as the evaluation baseline for many traditional JIT defect prediction approaches [8, 29, 50].

Within-project prediction. The "WP" columns in Table 5 present the overall within-project results for CC2Vec, DeepJIT and the other selected approaches on the extended dataset. We can observe that CC2Vec fails to retain performance advantages over DeepJIT on most of the projects. Specifically, for QT, OpenStack and Go, the prediction accuracy advantages of CC2Vec over DeepJIT in terms of the AUC scores are rather limited, i.e., 0.13%, 1.33% and 0.37% respectively. On the other hand, for the remaining three studied projects, DeepJIT can even outperform CC2Vec by 0.56% to 1.29%. Since CC2Vec only differs from DeepJIT by including additional code-change vector representations, such performance deviation can indicate that the code-change vector representation extracted by CC2Vec does not generalize its advantages over the original DeepJIT features to diverse datasets, indicating CC2Vec's limited effectiveness for real-world JIT defect prediction compared with DeepJIT.

> *Finding 4: In general, CC2Vec cannot clearly outperform DeepJIT in the extended dataset, indicating that the vector representation of code changes extracted by CC2Vec do not contribute much in advancing JIT defect prediction.*

Next, we can also observe that while DeepJIT and CC2Vec can deliver rather marginal average performance advantage over the two traditional approaches LR-JIT and DBN-JIT on top of all the adopted benchmarks, i.e., 0.7065 (DeepJIT) and 0.7055 (CC2Vec) vs. 0.6726 (LR-JIT) and 0.6847 (DBN-JIT), they cannot always outperform traditional approaches under all of the adopted projects. To be specific, on OpenStack, the AUC results of the two traditional approaches LR-JIT and DBN-JIT are 0.7248 and 0.7330, compared with 0.7132 of DeepJIT and 0.7227 of CC2Vec. On Eclipse JDT, LR-JIT can also slightly outperform DeepJIT and CC2Vec.

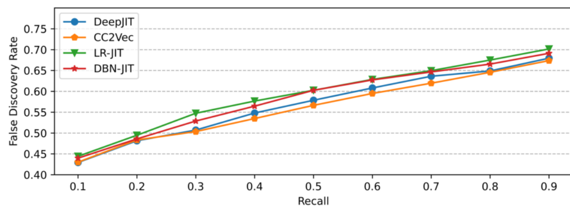
We further attempt to explore the prediction accuracy of all the studied approaches in terms of diverse metrics, e.g., false discovery rate, recall, and accuracy, as such metrics have been widely adopted in previous work [21, 49, 50]. In particular, Accuracy (ACC) is the ratio of the number of correct predictions over the total number of predictions; Recall denotes the ratio of the number of correctly predicted defective instances over the total number of defective

Table 5: Within- and cross-project JIT defect prediction on our extended dataset

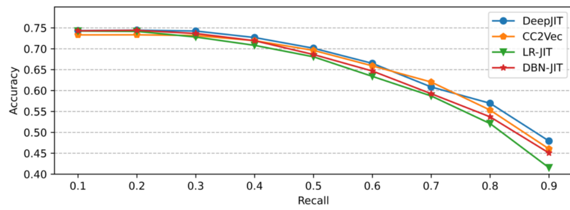
AUC Score	QT		OpenStack		JDT		Platform		Gerrit		Go		Mean		
	WP	CP	WP	CP	WP	CP	WP	CP	WP	CP	WP	CP	WP	CP	% Drop
DeepJIT	.6927	.6734	.7132	.7126	.6701	.6886	.7712	.6957	.7025	.6571	.6891	.6574	.7065	.6808	-3.63
CC2Vec	.6936	.6843	.7227	.7198	.6653	.6822	.7613	.6574	.6986	.6690	.6917	.6497	.7055	.6771	-4.03
LR-JIT	.6651	.5887	.7248	.6645	.6736	.6572	.6403	.6946	.6570	.6550	.6749	.6339	.6726	.6490	-3.52
DBN-JIT	.6752	.5844	.7330	.6707	.6426	.6595	.7016	.7113	.6835	.6602	.6724	.6334	.6847	.6532	-4.60

Table 6: With-project prediction on *early* versions

AUC	QT	OpenStack	JDT	Platform	Gerrit	Go	Mean
DeepJIT	.7144	.7140	.7491	.6912	.7875	.7314	.7317
CC2Vec	.7164	.7078	.7466	.6912	.7873	.7244	.7290
LR-JIT	.6843	.6750	.7497	.6912	.8131	.6783	.7152
DBN-JIT	.6858	.6627	.7267	.6781	.7757	.6805	.7016



(a) FDR



(b) ACC

Figure 5: FDR & ACC trends with respect to Recall

instances; False Discovery Rate (FDR) denotes the ratio of the number of instances incorrectly predicted as defective over the number of all instances predicted as defective.

While the existing approaches [21, 49, 50] simply present the ACC, Recall, and FDR values in terms of a given decision threshold (e.g., 0.5, above which a defect is predicted), we find that rather inapplicable in our study. The reason is that different projects can significantly vary from each other in terms of their defect patterns (as shown in Table 1), and a uniform threshold can hardly present their respective optimal performance. On the other hand, threshold can be easily manipulated in practice for adjusting Recall accordingly. To this end, we choose to present the ACC and FDR results in terms of different Recall values for a more comprehensive analysis. Specifically, we gradually adjust the thresholds such that the corresponding Recall values can be approached to be the scale values (with the error range < 1%).

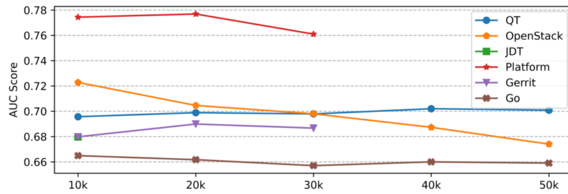
Figure 5a and 5b demonstrate the average FDR and ACC for all studied projects in terms of different Recall values. We can observe that the overall performance of advanced deep JIT defect prediction approaches is slightly better than the traditional approaches. However, such advantages are rather marginal and do not appear for all Recall values and all studied projects. We can also observe that DeepJIT slightly outperforms CC2Vec in terms of ACC, while CC2Vec slightly outperforms DeepJIT in terms of FDR. To summarize, DeepJIT and CC2Vec cannot always outperform traditional approaches in terms of FDR/ACC under identical Recall values.

> *Finding 5: DeepJIT and CC2Vec cannot always outperform traditional approaches on all the studied projects, and the finding holds for multiple widely-used metrics.*

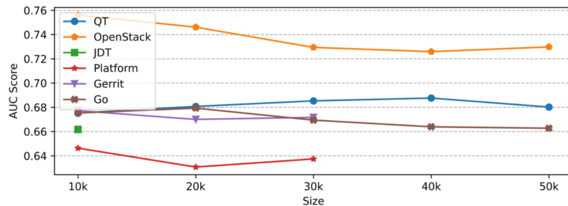
Furthermore, we perform a within-project experiment in the *early* versions of the studied projects from 2011-01-01 to 2014-01-01 (roughly the same as the project versions in DeepJIT and CC2Vec), as shown in Table 6. We can find that compared with the AUC scores on the *late* versions of the studied projects (shown in Table 5), the AUC scores on the early versions can be quite inconsistent. For instance, for DeepJIT, the AUC scores on the two settings can be close on QT and OpenStack, i.e., 0.7144 vs. 0.6927 on QT and 0.7140 vs. 0.7132 on OpenStack, while they can be quite different on other projects, e.g., 0.7491 vs. 0.6701 on JDT. The same is true for all studied approaches. Accordingly, we can derive that all studied approaches cannot ensure consistent performance even under the identical benchmarks with different versions.

The above performance inconsistency implies that for certain projects, their defect patterns can potentially vary between different program versions, causing potential challenges for designing training setups to optimize defect prediction. To better explore such concern, we further attempt to investigate the performance impact by adopting different training sets. In particular, we construct the training set with its size ranging from 10k to 50k historical commits for each studied project, and fixate most recent 2k samples for testing, i.e., 20% of the minimum training set size. Note that our training sets are constructed in the reverse chronological order so that larger training sets include more outdated historical data compared with smaller training sets; also, some projects will end with smaller training sets if they do not have sufficient historical data within last 10 years.

Figure 6 presents the AUC trends in terms of the training data size for DeepJIT and LR-JIT. Note that we select these two approaches as representatives since the performance of CC2Vec and DBN-JIT are similar with DeepJIT and LR-JIT, respectively. We can observe that in general, both approaches enable fluctuant prediction accuracy under the increasing training data size. For instance, the average AUC of OpenStack is degraded from 0.7294 to 0.7018, while in contrast, the average AUC of QT increases from 0.6854 to 0.6906. Moreover, even each approach itself can encounter different trends under the increasing training data size. For instance, on Platform, the AUC of DeepJIT first slightly increases from 0.7744 to 0.7769 and then drops to 0.7610. In contrast, the AUC of LR-JIT first decreases from 0.6463 to 0.6307, and then increases to 0.6374 eventually. Such



(a) DeepJIT



(b) LR-JIT

Figure 6: Impact of training set size

results directly indicate that simply expanding training set by including more historical data cannot always advance the prediction accuracy for both deep-learning-based and traditional approaches. We can also infer that the outdated historical data may not conform to the defect patterns of more recent data, and thus have limited effectiveness, which is consistent with prior findings on traditional JIT defect prediction [31]. Therefore, it can be rather challenging to adopt the optimal training set for JIT defect prediction.

Finding 6: Simply adding more historical data cannot improve the prediction accuracy for both advanced deep JIT defect prediction approaches and traditional approaches.

Cross-project prediction. While the original DeepJIT and CC2Vec papers do not include any cross-project evaluation, in this paper, we further study the performance of DeepJIT, CC2Vec, and the other studied approaches for cross-project validation, which attempts to address the issues caused by insufficient data from the projects under prediction. Specifically, similar as existing work [7, 20], our cross-project validation collects historical data from all the other adopted projects rather than the project under prediction for training a classifier, i.e., we use the training data from all the *other* studied projects for training a classifier and use the same testing data as the within-project evaluation shown in Table 5 for testing.

The "CP" columns in Table 5 present the AUC scores for the studied approaches in cross-project prediction. We can observe the that deep JIT defect prediction approaches also cannot consistently outperform the traditional approaches on all the projects in this scenario. To be specific, on Platform, the AUC scores of the two traditional approaches (LR-JIT and DBN-JIT) are 0.6946 and 0.7113, compared with 0.6957 of DeepJIT and 0.6574 of CC2Vec. Also, compared with the within-project evaluation results (presented in the "WP" columns in Table 5), we can observe that for all the studied approaches, their within-project prediction accuracy can outperform their cross-project prediction accuracy for almost all the studied projects. For instance, the average AUC score of DeepJIT drops slightly form 0.7065 to 0.6808 in cross-project

Table 7: Studied 14 basic change features

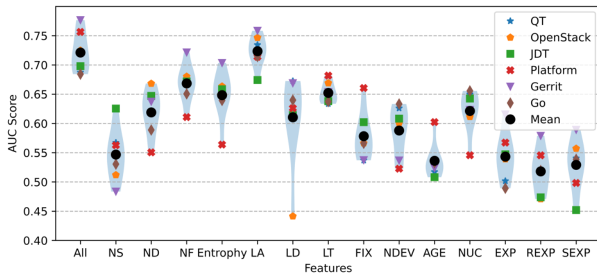
Name	Description
NS	The number of modified subsystems [32]
ND	The number of modified directories [32]
NF	The number of modified files [35]
Entropy	Distribution of modified code across each file [9, 15]
LA	Lines of code added [33, 34]
LD	Lines of code deleted [33, 34]
LT	Lines of code in a file before the change [24]
FIX	Whether or not the change is a defect fix [14, 37]
NDEV	The number of developers that changed the modified files [30]
AGE	The average time interval between the last and current change [13]
NUC	The number of unique changes to the modified files [9, 15]
EXP	Developer experience [32]
REXP	Recent developer experience [32]
SEXP	Developer experience on a subsystem [32]

validation, while the average AUC scores of LR-JIT and DBN-JIT drop from 0.6726 and 0.6847 to 0.6490 and 0.6532, respectively. Such results conform to the finding on traditional JIT detect prediction that within-project prediction can usually outperform cross-project prediction [20]. Therefore, cross-project prediction is usually taken as the secondary choice when within-project prediction is not possible. Another interesting finding is that deep JIT approaches can incur roughly the same AUC loss as the traditional approaches when moving from within-project to cross-project prediction, i.e., -3.63% and -4.03% for two recent deep learning approaches vs. -3.52% and -4.60% for two traditional approaches. This indicates that deep JIT defect prediction approaches are not more robust than the traditional approaches in cross-project prediction.

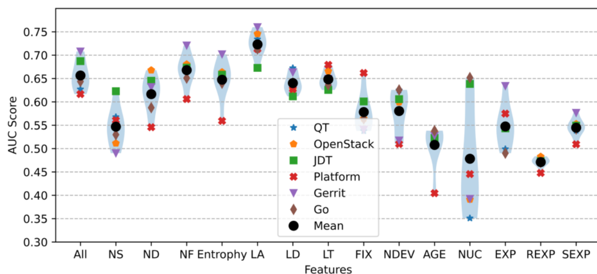
Finding 7: JIT defect prediction approaches usually perform better for within-project prediction than for cross-project prediction. In addition, deep JIT defect prediction approaches are not more robust than the traditional approaches in cross-project prediction.

3.3.3 RQ3: How Do Traditional Defect Prediction Features Perform for Just-in-Time Defect Prediction? The findings from RQ1 and RQ2 altogether reveal the following facts. First, for deep JIT defect prediction approaches, the DeepJIT code-change representations (i.e., the "added_ code/removed_ code" labels, which is essentially close to a traditional defect prediction feature, i.e., changed file number) have been shown to be more important than other feature inputs, i.e., DeepJIT commit-message representations and CC2Vec code-change representations (Table 4). Second, CC2Vec with more input features than DeepJIT can sometimes even degrade the prediction accuracy. Third, the recent deep JIT defect prediction approaches cannot consistently outperform traditional JIT defect prediction approaches under different settings/scenarios. Such facts expose that different features from traditional approaches have not been fully studied for JIT defect prediction. Accordingly, we attempt to answer this RQ by further investigating the performance of various representative traditional features for JIT defect prediction.

Our investigation is launched based on a simple and classic JIT defect prediction approach proposed by Kamei et al. [20] (i.e., LR-JIT in RQ2). More specifically, they train a logistic regression classifier on a set of 14 manually summarized features that can demonstrate the size, diffusion, history, and author experience of the associated



(a) Within-project validation



(b) Cross-project validation

Figure 7: Effectiveness study for individual features

commit (presented in Table 7). Such features have also been widely used in other JIT defect prediction work [8, 28, 49, 50, 52]. For each feature, we build a logistic regression classifier for both within- and cross-project prediction to understand its impact. Meanwhile, since the results may help inspire our own simplistic approach (RQ4), we only leverage the training data used in computing Table 5 for feature analysis to prevent data leakage [40], i.e., we perform 10-fold cross-validation [23] within the training data to explore each feature's detailed impact on JIT defect prediction.

Figure 7a presents the AUC scores in terms of the adopted traditional features for within-project prediction, where the x-axis denotes the studied features while the y-axis denotes the AUC scores. Note that "all" indicates that all the traditional features are adopted for model training, while all other *x-axis* labels indicate that only the corresponding feature is adopted for model training. Interestingly, we find that it is possible to build a well-performed classifier for JIT defect prediction using only one single feature. In particular, the average AUC score of the model increases from 0.7212 when all the features are adopted to 0.7236 when only the added-line-number ("la") feature is used. Moreover, using the "la" feature, the AUC scores of QT, OpenStack, and Go can reach 0.7343, 0.7465, and 0.7145, while they are only 0.6872, 0.7247, and 0.6842 when all the features are used.

We further present the effectiveness of each single traditional feature for cross-project prediction in Figure 7b. In general, we find that compared with the within-project prediction results, many features are presented with significant performance variance, e.g., the "nuc" feature achieves an average AUC 0.6214 in within-project prediction, but only 0.4783 in cross-project prediction. We infer that such performance variance is caused by the divergent distributions of the studied features among different studied projects. On the other hand, the fact that the cross-project prediction accuracy of the "la" feature is quite close to its within-project prediction accuracy indicates that such a simple feature is potentially variance-resilient in its distribution among different projects.

> *Finding 8: The within-project and cross-project JIT defect prediction using only the "la" feature can achieve high prediction accuracy and negligible performance variance.*

3.3.4 *RQ4: Can a Simpler Approach Outperform DeepJIT and CC2Vec?*

Inspired by Finding 8, we propose a simplistic "la"-feature-based JIT defect prediction approach, namely *LApredict*. While it is definitely possible to integrate the "la" (added-line-number) feature with other information for more powerful JIT defect prediction, in this paper, we want to keep our design simplistic and only leverage the "la" feature with the classic logistic regression model for JIT defect prediction. We have empirically compared *LApredict* with all the other studied approaches, i.e., DeepJIT, CC2Vec, DBN-JIT, and LR-JIT, for both within- and cross-project validation (under the same setting as Table 5).

The "WP" columns in Table 8 show the AUC scores of all the compared approaches for within-project prediction. Surprisingly, we observe that *LApredict* can outperform all the other approaches on average and on most of the studied projects! Specifically, the average AUC of *LApredict* is 0.7246 while the best of the other approaches is only 0.7065. Moreover, on QT, OpenStack, and Gerrit, *LApredict* achieves an AUC of 0.7438, 0.7491 and 0.7495, while the best results for DeepJIT and CC2Vec on those three projects are only 0.6936, 0.7227 and 0.7025.

The "CP" columns in Table 8 further presents the AUC scores of all the studied approaches for cross-project prediction. We can observe that *LApredict* not only can retain its performance superiority over DeepJIT and CC2Vec, but also further expand its advantage compared with within-project prediction! The potential reason is that while using the cross-project data can potentially bring extra noises to the deep-learning-based approaches and thus degrade the prediction accuracy compared with within-project prediction, the simplistic added-line-number ("la") feature can stay steady across projects since the training data of both WP and CP reflect the same rule, i.e., the larger the "la", the higher the probability of being defective. In fact, using the logistic regression model, *LApredict* will always predict commit with a larger "la" value as more defect-prone. Also, the AUC score is calculated only based on the relative ranking of the predicted results. Thus, even different training data are used in WP/CP, the same testing data will always yield the same AUC.

We further present the time costs of all the studied approaches in terms of both training and testing time under within-project prediction in Table 9. Note that the cross-project prediction results are similar, and thus are omitted here. We can observe that *LApredict* has negligible training and testing time due to its simplistic design (i.e., single feature with traditional classifier). On the contrary, the recent deep-learning-based approaches (i.e., DeepJIT and CC2Vec) can be much most costly due to their complicated neural network design, e.g., at least 81k X/120k X slower than *LApredict* in training/testing time.

Table 8: Within- and cross-project prediction effectiveness for *LApredict*

AUC Score	QT		OpenStack		JDT		Platform		Gerrit		Go		Mean	
	WP	CP	WP	CP	WP	CP	WP	CP	WP	CP	WP	CP	WP	CP
DeepJIT	.6927	.6734	.7132	.7126	.6701	.6886	.7712	.6957	.7025	.6571	.6891	.6574	.7065	.6808
CC2Vec	.6936	.6843	.7227	.7198	.6653	.6822	.7613	.6574	.6986	.6690	.6917	.6497	.7055	.6771
LR-JIT	.6651	.5887	.7248	.6645	.6736	.6572	.6403	.6946	.6570	.6550	.6749	.6339	.6726	.6490
DBN-JIT	.6752	.5844	.7330	.6707	.6426	.6595	.7016	.7113	.6835	.6602	.6724	.6334	.6847	.6532
LApredict	.7438	.7438	.7491	.7491	.6757	.6757	.7461	.7461	.7495	.7495	.6831	.6831	.7246	.7246

Table 9: Training and testing time for all studied approaches (Seconds)

Time Cost	QT		OpenStack		JDT		Platform		Gerrit		Go		Mean	
	Train	Test	Train	Test	Train	Test	Train	Test	Train	Test	Train	Test	Train	Test
DeepJIT	1.37e+03	1.65e+01	1.37e+03	1.38e+01	1.85e+02	1.45e+01	6.35e+02	3.23e+01	8.75e+02	4.23e+01	1.07e+03	2.45e+01	9.17e+02	2.40e+01
CC2Vec	1.42e+03	1.54e+01	1.32e+03	4.89e+01	1.85e+02	4.47e+01	6.22e+02	3.79e+01	9.06e+02	3.32e+01	1.10e+03	2.61e+01	9.25e+02	3.44e+01
LR-JIT	2.07e-02	4.00e-04	2.30e-02	4.00e-04	6.90e-03	2.00e-04	1.61e-02	2.00e-04	1.88e-02	2.00e-04	1.76e-02	2.00e-04	1.72e-02	3.00e-04
DBN-JIT	4.96e+01	3.00e-03	4.74e+01	3.00e-03	7.00e+00	6.00e-04	2.26e+01	2.00e-03	3.16e+01	2.00e-03	4.00e+00	3.00e-03	3.30e+01	2.40e-03
LApredict	1.14e-02	3.00e-04	1.29e-02	4.00e-04	5.80e-03	2.00e-04	1.25e-02	2.00e-04	1.51e-02	2.00e-04	1.03e-02	2.00e-04	1.13e-02	2.00e-04

Finding 9: Our simplistic LApredict can substantially outperform the advanced deep-learning-based approaches in JIT defect prediction under both within-project and cross-project scenarios in terms of both effectiveness and efficiency.

4 IMPLICATIONS AND DISCUSSIONS

Our study reveals the following important practical guidelines for future JIT defect prediction.

Deep learning does not always help. Our study reveals that although the deep JIT defect prediction approaches can be advanced under certain scenarios, they are rather data dependant and have limited effectiveness under diverse datasets/scenarios (Findings 5, 7, and 9). Also, the interpretability issue of deep learning models makes it even harder to ensure the optimal performance of deep JIT defect prediction. Furthermore, deep learning techniques can be orders of magnitude slower than traditional classifiers. We highly recommend the researchers/developers to conduct thorough evaluations for future deep JIT defect prediction approaches.

Simple features can work. Our study shows that DeepJIT_{GitHub}, which abstracts code changes in each file into a simple label (i.e., "added_code/removed_code"), performs even better than the original DeepJIT_{Paper} described in the paper, which analyzes the detailed code changes (Finding 1). Also, the single added-line-number feature (with the logistic regression model) can already outperform all existing JIT defect prediction approaches, including the deep-learning-based approaches with rather complicated features (Finding 9). All such observations indicate that simple features should not be just ignored. In fact, they should be considered and even combined with advanced learning models and features for even more advanced JIT defect prediction.

Commit messages are helpful. Our results show that certain keywords in the commit messages are rather helpful for deep JIT defect prediction, since they can convey the intents of specific code changes (Finding 2). This suggests that developers/teams interested in JIT defect prediction should maintain strict rules for drafting informative commit messages. Furthermore, this also motivates researchers to develop more advanced approaches to better align commit messages with code changes to avoid the negative impact of noisy commit messages.

Training data selection is important. Our study shows that simply adding more training data cannot enhance the prediction accuracy for traditional or deep JIT defect prediction approaches (Finding 6). On the other hand, it is rather challenging to manually capture the most informative training set which optimizes the prediction accuracy under different benchmarks/scenarios. Therefore, we suggest researchers/developers to consider fully automated training data selection approaches targeting specific projects under prediction or even specific commits under prediction.

Cross-project validation should be considered. Our experimental results show that the existing traditional/deep JIT defect prediction approaches witnessed a decreased performance when switching to cross-project validation (Finding 7). Interesting, our simplistic *LApredict*, which only uses the added-line-number feature with the classic logistic regression model, incurs no performance drop in cross-project validation (Finding 9). Such observation motivates future researchers to investigate more robust JIT defect prediction approaches under both within- and cross-project scenarios.

5 THREATS TO VALIDITY

Threats to Internal Validity. The threats to internal validity mainly lie in the potential bugs in our implementation. To reduce such threats, we directly obtain the original source code from the GitHub repositories of the studied techniques. Also, we use the same hyperparameters as the original papers. The authors also carefully reviewed the experimental scripts to ensure their correctness. Meanwhile, the validity of the SZZ algorithm used for our dataset construction is also a critical threat that has been widely argued. Several studies have pointed out the limitations of SZZ [16, 38], while there are also studies demonstrating the effectiveness/accuracy of SZZ [11, 22, 48]. Despite the ongoing debate about SZZ, our focus is on the performance comparison of different prediction models, which is not directly affected by the SZZ algorithm. Therefore, we simply applied the same SZZ algorithm with DeepJIT/CC2Vec for a fair comparison.

Threats to External Validity. The threats to external validity mainly lie in the benchmark projects used in this study. To reduce such threats, we not only use all the benchmark projects studied in the original CC2Vec and DeepJIT papers, but also construct a much larger-scale JIT defect prediction dataset (with 310k commits).

Threats to Construct Validity. The threat to construct validity mainly lies in the adopted metrics in our evaluations. To reduce such threat, following the prior CC2Vec and DeepJIT work, we adopt the widely-used Area Under the Curve (AUC) score to evaluate the performance of the studied approaches. In particular, the AUC score does not need to manually set a threshold and thus can be quite objective [43]. To further reduce the threats, we also adopt other widely used metrics, e.g., Recall, ACC, and FDR.

6 RELATED WORK

JIT defect prediction has been extensively studied in the last two decades. Mockus et al. [32] built a classifier based on information extracted from historic changes to predict the risk of new code changes. Kamei et al. [21] built a Logistic Regression Model on a set of 14 manually summarized features. Meanwhile, they also took into account the effort required to find defects in their evaluation. Based on Kamei's work [21], Yang et al. [50] used Deep Belief Network (DBN) to extract higher-level information from the initial features and achieved better performance compared with Kamei's approach. Also, by adopting Kamei's features, Yang et al. [49] combined decision tree and ensemble learning to build a two-layer ensemble learning model for JIT defect prediction. Later, to enhance such approach [49], by allowing using arbitrary classifiers in the ensemble and optimizing the weights of the classifiers, Young et al. [52] proposed a new deep ensemble approach and achieved significantly better results. Meanwhile, Liu et al. [29] found that the code churn metric is effective in effort-aware JIT defect prediction, while Chen et al. [8] transformed the effort-aware JIT defect prediction task into a multi-objective optimization problem and selected a set of effective features via evolutionary algorithms. More recently, Cabral et al. [6] provided a new sampling approach to handle verification latency and class imbalance evolution of online JIT defect prediction. Also, Yan et al. [47] proposed a two-phase framework by combining JIT defect identification and localization. Note that most of such approaches are developed based on the two classic traditional JIT defect prediction approaches (Kamei et al. [21] and Yang et al. [50]) adopted for our study. On the other hand, the current state-of-the-art approaches, CC2Vec [19] and DeepJIT [18], attempt to enhance the prediction accuracy by applying modern advanced deep learning models to automatically extract features from commit messages and code changes.

There also have been various studies on JIT defect prediction. McIntosh et al. [31] conducted a study of the impact of systems evolution on JIT defect prediction models via a longitudinal case study of 37,524 changes from the QT and OpenStack systems. They found that the performance of the JIT defect prediction model decreases as the interval between training periods and testing periods increases. Tabassum et al. [42] investigated cross-project JIT defect prediction in realistic online learning scenarios. They showed that in online learning scenarios, the model trained with both within- and cross-project data achieved better performance over the model trained with within-project data only. Recently, through literature review and a survey of practitioners, Wan et al. [44] discussed the shortcomings of existing defect prediction tools and highlighted future research directions. Different from all prior studies, our work presents the first extensive study of the state-of-the-art deep JIT defect prediction approaches.

SZZ is also a critical part to be discussed [38]. Kim et al. [22] proposed an improved SZZ and claimed that such SZZ can achieve high accuracy under manual validation. Note that our dataset construction process followed such enhanced SZZ algorithms to ensure high accuracy (Section 3.1). Recently, Herbold et al. [16] conducted an empirical study on 38 Apache projects and found that many of the bug fixing commits identified by SZZ were incorrect or imprecise. Meanwhile, Fan et al. [11] investigated the impact of different SZZ variants on JIT defect prediction, and showed that different SZZ algorithms have limited impact on the prediction accuracy of the studied JIT defect prediction models. More recently, Yan et al. [48] performed a case study on 14 industrial projects and found that SZZ-tagged data can be validated to successfully derive a JIT defect prediction model which can help developers reduce their workload in real-world code review process. Despite the ongoing debate about SZZ, our focus is on the performance comparison of different prediction models, which is not directly affected by the SZZ algorithm. Therefore, we simply applied the same SZZ algorithm with DeepJIT/CC2Vec for a fair comparison.

7 CONCLUSIONS

In this study, we have investigated the effectiveness and limitations of state-of-the-art deep JIT defect prediction approaches, i.e., DeepJIT and CC2Vec. Specifically, we have constructed an extended dataset containing over 310k commits and used it to evaluate the performance of DeepJIT, CC2Vec, and two representative traditional JIT defect prediction approaches. We found that CC2Vec cannot consistently outperform DeepJIT and neither of them can be ensured to outperform the traditional JIT defect prediction approaches. We also noticed that all the studied traditional and deep JIT defect prediction approaches witnessed a performance drop in cross-project validation. Moreover, simply increasing the training data size does not improve the prediction accuracy of the studied approaches. Surprisingly, we have also demonstrated that a simplistic JIT defect prediction approach, *LApredict*, which simply uses the added-line-number feature with a traditional classifier, can already outperform CC2Vec and DeepJIT in terms of both effectiveness and efficiency on most studied projects. Lastly, our study also revealed various practical guidelines for future JIT defect prediction.

ACKNOWLEDGEMENTS

This work was partially supported by the National Natural Science Foundation of China (Grant No. 61902169) and Shenzhen Peacock Plan (Grant No. KQTD2016112514355531). This work was also partially supported by National Science Foundation under Grant Nos. CCF-1763906 and CCF-1942430, and Kwai Inc.

REFERENCES

[1] 2020. Eclipse JDT adn Eclipse Platform. Website. https://git.eclipse.org/r/.
[2] 2020. Gerrit Code Review Website. Website. https://www.gerritcodereview.com/.
[3] 2020. Golang Code Review Website. Website. https://go-review.googlesource.com/.
[4] 2020. OpenStack Code Review Website. Website. https://review.opendev.org/.
[5] 2020. QT Code Review Website. Website. https://codereview.qt-project.org/.
[6] George G Cabral, Leandro L Minku, Emad Shihab, and Suhaib Mujahid. 2019. Class imbalance evolution and verification latency in just-in-time software defect

prediction. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 666–676. https://doi.org/10.1109/ICSE.2019.00076

[7] Gemma Catolino, Dario Di Nucci, and Filomena Ferrucci. 2019. Cross-project just-in-time bug prediction for mobile apps: an empirical assessment. In *2019 IEEE/ACM 6th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*. IEEE, 99–110.

[8] Xiang Chen, Yingquan Zhao, Qiuping Wang, and Zhidan Yuan. 2018. MULTI: Multi-objective effort-aware just-in-time software defect prediction. *Information and Software Technology* 93 (2018), 1–13.

[9] Marco D'Ambros, Michele Lanza, and Romain Robbes. 2010. An extensive comparison of bug prediction approaches. In *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*. IEEE, 31–41.

[10] Geanderson Esteves, Eduardo Figueiredo, Adriano Veloso, Markos Viggiato, and Nivio Ziviani. 2020. Understanding machine learning software defect predictions. *Automated Software Engineering* 27, 3 (2020), 369–392.

[11] Yuanrui Fan, Xin Xia, Daniel Alencar da Costa, David Lo, Ahmed E Hassan, and Shanping Li. 2019. The impact of changes mislabeled by SZZ on just-in-time defect prediction. *IEEE Transactions on Software Engineering* (2019). https://doi.org/10.1109/TSE.2019.2929761

[12] John Fox. 1997. *Applied regression analysis, linear models, and related methods*. Sage Publications, Inc.

[13] Todd L Graves, Alan F Karr, James S Marron, and Harvey Siy. 2000. Predicting fault incidence using software change history. *IEEE Transactions on software engineering* 26, 7 (2000), 653–661. https://doi.org/10.1109/32.859533

[14] Philip J Guo, Thomas Zimmermann, Nachiappan Nagappan, and Brendan Murphy. 2010. Characterizing and predicting which bugs get fixed: an empirical study of Microsoft Windows. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering-Volume 1*. 495–504.

[15] Ahmed E Hassan. 2009. Predicting faults using the complexity of code changes. In *2009 IEEE 31st international conference on software engineering*. IEEE, 78–88.

[16] Steffen Herbold, Alexander Trautsch, Fabian Trautsch, and Benjamin Ledel. 2019. Issues with SZZ: An empirical assessment of the state of practice of defect prediction data collection. *arXiv preprint arXiv:1911.08938* (2019).

[17] Geoffrey E Hinton and Ruslan R Salakhutdinov. 2006. Reducing the dimensionality of data with neural networks. *science* 313, 5786 (2006), 504–507.

[18] Thong Hoang, Hoa Khanh Dam, Yasutaka Kamei, David Lo, and Naoyasu Ubayashi. 2019. DeepJIT: an end-to-end deep learning framework for just-in-time defect prediction. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE, 34–45. https://doi.org/10.1109/MSR.2019.00016

[19] Thong Hoang, Hong Jin Kang, David Lo, and Julia Lawall. 2020. CC2Vec: Distributed representations of code changes. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 518–529. https://doi.org/10.1145/3377811.3380361

[20] Yasutaka Kamei, Takafumi Fukushima, Shane McIntosh, Kazuhiro Yamashita, Naoyasu Ubayashi, and Ahmed E Hassan. 2016. Studying just-in-time defect prediction using cross-project models. *Empirical Software Engineering* 21, 5 (2016), 2072–2106. https://doi.org/10.1007/s10664-015-9400-x

[21] Yasutaka Kamei, Emad Shihab, Bram Adams, Ahmed E Hassan, Audris Mockus, Anand Sinha, and Naoyasu Ubayashi. 2012. A large-scale empirical study of just-in-time quality assurance. *IEEE Transactions on Software Engineering* 39, 6 (2012), 757–773. https://doi.org/10.1109/TSE.2012.70

[22] Sunghun Kim, Thomas Zimmermann, Kai Pan, E James Jr, et al. 2006. Automatic identification of bug-introducing changes. In *21st IEEE/ACM international conference on automated software engineering (ASE'06)*. IEEE, 81–90.

[23] Ron Kohavi et al. 1995. A study of cross-validation and bootstrap for accuracy estimation and model selection. In *Ijcai*, Vol. 14. Montreal, Canada, 1137–1145.

[24] A Güneş Koru, Dongsong Zhang, Khaled El Emam, and Hongfang Liu. 2008. An investigation into the functional form of the size-defect relationship for software modules. *IEEE Transactions on Software Engineering* 35, 2 (2008), 293–304.

[25] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. 2015. Deep learning. *nature* 521, 7553 (2015), 436–444. https://doi.org/10.1038/nature14539

[26] Stefan Lessmann, Bart Baesens, Christophe Mues, and Swantje Pietsch. 2008. Benchmarking classification models for software defect prediction: A proposed framework and novel findings. *IEEE Transactions on Software Engineering* 34, 4 (2008), 485–496. https://doi.org/10.1109/TSE.2008.35

[27] Jian Li, Pinjia He, Jieming Zhu, and Michael R Lyu. 2017. Software defect prediction via convolutional neural network. In *2017 IEEE International Conference on Software Quality, Reliability and Security (QRS)*. IEEE, 318–328. https://doi.org/10.1109/QRS.2017.42

[28] Weiwei Li, Wenzhou Zhang, Xiuyi Jia, and Zhiqiu Huang. 2020. Effort-Aware semi-Supervised just-in-Time defect prediction. *Information and Software Technology* 126 (2020), 106364. https://doi.org/10.1016/j.infsof.2020.106364

[29] Jinping Liu, Yuming Zhou, Yibiao Yang, Hongmin Lu, and Baowen Xu. 2017. Code churn: A neglected metric in effort-aware just-in-time defect prediction. In *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. IEEE, 11–19. https://doi.org/10.1109/ESEM.2017.8

[30] Shinsuke Matsumoto, Yasutaka Kamei, Akito Monden, Ken-ichi Matsumoto, and Masahide Nakamura. 2010. An analysis of developer metrics for fault prediction. In *Proceedings of the 6th International Conference on Predictive Models in Software Engineering*. 1–9. https://doi.org/10.1145/1868328.1868356

[31] Shane McIntosh and Yasutaka Kamei. 2017. Are fix-inducing changes a moving target? a longitudinal case study of just-in-time defect prediction. *IEEE Transactions on Software Engineering* 44, 5 (2017), 412–428.

[32] Audris Mockus and David M Weiss. 2000. Predicting risk of software changes. *Bell Labs Technical Journal* 5, 2 (2000), 169–180. https://doi.org/10.1002/bltj.2229

[33] Raimund Moser, Witold Pedrycz, and Giancarlo Succi. 2008. 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*. 181–190. https://doi.org/10.1145/1368088.1368114

[34] Nachiappan Nagappan and Thomas Ball. 2005. Use of relative code churn measures to predict system defect density. In *Proceedings of the 27th international conference on Software engineering*. 284–292. https://doi.org/10.1145/1062455.1062514

[35] Nachiappan Nagappan, Thomas Ball, and Andreas Zeller. 2006. Mining metrics to predict component failures. In *Proceedings of the 28th international conference on Software engineering*. 452–461. https://doi.org/10.1145/1134285.1134349

[36] Hung Viet Pham, Shangshu Qian, Jiannan Wang, Thibaud Lutellier, Jonathan Rosenthal, Lin Tan, Yaoliang Yu, and Nachiappan Nagappan. 2020. Problems and Opportunities in Training Deep Learning Software Systems: An Analysis of Variance. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 771–783.

[37] Ranjith Purushothaman and Dewayne E Perry. 2005. Toward understanding the rhetoric of small source code changes. *IEEE Transactions on Software Engineering* 31, 6 (2005), 511–526. https://doi.org/10.1109/TSE.2005.74

[38] Gema Rodríguez-Pérez, Gregorio Robles, and Jesús M González-Barahona. 2018. Reproducibility and credibility in empirical software engineering: A case study based on a systematic literature review of the use of the szz algorithm. *Information and Software Technology* 99 (2018), 164–176.

[39] Ramprasaath R Selvaraju, Michael Cogswell, Abhishek Das, Ramakrishna Vedantam, Devi Parikh, and Dhruv Batra. 2017. Grad-cam: Visual explanations from deep networks via gradient-based localization. In *Proceedings of the IEEE international conference on computer vision*. 618–626.

[40] Asaf Shabtai, Yuval Elovici, and Lior Rokach. 2012. *A survey of data leakage detection and prevention solutions*. Springer Science & Business Media.

[41] Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. 2005. When do changes induce fixes? *ACM sigsoft software engineering notes* 30, 4 (2005), 1–5.

[42] Sadia Tabassum, Leandro L Minku, Danyi Feng, George G Cabral, and Liyan Song. [n.d.]. An Investigation of Cross-Project Learning in Online Just-In-Time Software Defect Prediction. ([n. d.]).

[43] Chakkrit Tantithamthavorn, Ahmed E Hassan, and Kenichi Matsumoto. 2018. The impact of class rebalancing techniques on the performance and interpretation of defect prediction models. *IEEE Transactions on Software Engineering* (2018).

[44] Zhiyuan Wan, Xin Xia, Ahmed E Hassan, David Lo, Jianwei Yin, and Xiaohu Yang. 2018. Perceptions, expectations, and challenges in defect prediction. *IEEE Transactions on Software Engineering* (2018).

[45] Song Wang, Taiyue Liu, Jaechang Nam, and Lin Tan. 2018. Deep semantic feature learning for software defect prediction. *IEEE Transactions on Software Engineering* (2018). https://doi.org/10.1109/TSE.2018.2877612

[46] Tiejian Wang, Zhiwu Zhang, Xiaoyuan Jing, and Liqiang Zhang. 2016. Multiple kernel ensemble learning for software defect prediction. *Automated Software Engineering* 23, 4 (2016), 569–590. https://doi.org/10.1007/s10515-015-0179-1

[47] Meng Yan, Xin Xia, Yuanrui Fan, Ahmed E Hassan, David Lo, and Shanping Li. 2020. Just-In-Time Defect Identification and Localization: A Two-Phase Framework. *IEEE Transactions on Software Engineering* (2020).

[48] Meng Yan, Xin Xia, Yuanrui Fan, David Lo, Ahmed E Hassan, and Xindong Zhang. 2020. Effort-aware just-in-time defect identification in practice: a case study at Alibaba. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1308–1319. https://doi.org/10.1145/3368089.3417048

[49] Xinli Yang, David Lo, Xin Xia, and Jianling Sun. 2017. TLEL: A two-layer ensemble learning approach for just-in-time defect prediction. *Information and Software Technology* 87 (2017), 206–220. https://doi.org/10.1016/j.infsof.2017.03.007

[50] Xinli Yang, David Lo, Xin Xia, Yun Zhang, and Jianling Sun. 2015. Deep learning for just-in-time defect prediction. In *2015 IEEE International Conference on Software Quality, Reliability and Security*. IEEE, 17–26.

[51] Zichao Yang, Diyi Yang, Chris Dyer, Xiaodong He, Alex Smola, and Eduard Hovy. 2016. Hierarchical attention networks for document classification. In *Proceedings of the 2016 conference of the North American chapter of the association for computational linguistics: human language technologies*. 1480–1489.

[52] Steven Young, Tamer Abdou, and Ayse Bener. 2018. A replication study: just-in-time defect prediction with ensemble learning. In *Proceedings of the 6th International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering*. 42–47. https://doi.org/10.1145/3194104.3194110

[53] Motahareh Bahrami Zanjani, Huzefa Kagdi, and Christian Bird. 2015. Automatically recommending peer reviewers in modern code review. *IEEE Transactions on Software Engineering* 42, 6 (2015), 530–543.