DeepLineDP: Towards a Deep Learning Approach for Line-Level Defect Prediction

Chanathip Pornprasit, *Student Member, IEEE* and Chakkrit (Kla) Tantithamthavorn, *Member, IEEE*

Abstract—Defect prediction is proposed to assist practitioners effectively prioritize limited Software Quality Assurance (SQA) resources on the most risky files that are likely to have post-release software defects. However, there exist two main limitations in prior studies: (1) the granularity levels of defect predictions are still coarse-grained and (2) the surrounding tokens and surrounding lines have not yet been fully utilized. In this paper, we perform a survey study to better understand how practitioners perform code inspection in modern code review process, and their perception on a line-level defect prediction. According to the responses from 36 practitioners, we found that 50% of them spent at least 10 minutes to more than one hour to review a single file, while 64% of them still perceived that code inspection activity is challenging to extremely challenging. In addition, 64% of the respondents perceived that a line-level defect prediction tool would potentially be helpful in identifying defective lines. Motivated by the practitioners' perspective, we present DeepLineDP, a deep learning approach to automatically learn the semantic properties of the surrounding tokens and lines in order to identify defective files and defective lines. Through a case study of 32 releases of 9 software projects, we find that the risk score of code tokens varies greatly depending on their location. Our DeepLineDP is 17%-37% more accurate than other file-level defect prediction approaches; is 47%-250% more cost-effective than other line-level defect prediction approaches; and achieves a reasonable performance when transferred to other software projects. These findings confirm that the surrounding tokens and surrounding lines should be considered to identify the fine-grained locations of defective files (i.e., defective lines).

Index Terms—Software quality assurance, line-level defect prediction, deep learning, explainable AI

1 Introduction

Software defects are prevalent and costly. Thus, Software Quality Assurance (SQA) practices play a critical role to ensure the absence of software defects. Despite several SQA tools having been heavily invested during the development phase (e.g., CI/CD, code review, static analysis), software defects may still slip through to the official releases of a software product (i.e., post-release defects) [62], [63]. Vassallo *et al*. [66] found that practitioners still control for quality only at the end of a sprint and at the release preparation stage. However, real-world software projects are extremely large and complex. Hence, it is intuitively infeasible to exhaustively perform SQA activities for all of the files of the codebase.

To address this challenge, defect prediction is proposed to help developers prioritize their limited SQA resources on the most risky files that are likely to have post-release software defects. Recent studies found that deep learning approaches, which automatically learn syntactic and semantic features, outperform traditional machine learning approaches that used process and product as features metrics [9], [34], [71]. However, there exist two main limitations in prior studies.

(1) The Granularity Levels of Deep Learning-Based Defect Predictions are Still Coarse-Grained. Defect prediction models have been proposed at various levels of granularity (e.g., packages [27], components [63], modules [28], files [9], [27], [39], [47], [71], methods [15], and commits [18], [19], [29]). However, not all lines in a defective file are actually defective. Therefore, developers still waste a large amount of SQA effort to inspect clean lines that may not lead to post-release defects (i.e., Table 1 shows that only 0.03%-2.9% lines of the whole release are defective). As such, line-level defect prediction is needed to help developers prioritize their SQA effort on the high-risk areas of source code so SQA effort can be allocated in a cost-effective manner. However, line-level defect prediction is a challenging problem and still remains largely unexplored (e.g., no existing deep learning-based approach considers sequence of tokens).

(2) The Surrounding Tokens and the Surrounding Lines Have not yet Been Fully Utilized. Source code has a hierarchical structure (i.e., tokens forming lines and lines forming files) and is contextually dependent. Thus, the same code token that appears in different lines may have different lexical meaning depending on its location (e.g., variable declaration or assigning a value to a variable). Therefore, the riskiness of code tokens should be different depending on their location. However, current deep learning approaches for file-level defect prediction [9], [34], [70], [71] can capture only the long-term sequences of code tokens without considering surrounding lines, assuming that the same code token at different lines is equally important to predict defective files—which is likely not true.

• *The authors are with the Faculty of Information Technology, Monash University, Melbourne, VIC 3800, Australia. E-mail: chanathip.sit@gmail.com, chakkrit@monash.edu.*

Manuscript received 2 December 2020; revised 5 January 2022; accepted 15 January 2022. Date of publication 21 January 2022; date of current version 9 January 2023.
The work of Chakkrit (Kla) Tantithamthavorn was partially supported by the Australian Research Council's Discovery Early Career Researcher Award (DECRA) funding scheme under Grant DE200100941.
(Corresponding author: Chakkrit (Kla) Tantithamthavorn.)
Recommended for acceptance by A. Orso.
Digital Object Identifier no. 10.1109/TSE.2022.3144348

TABLE 1
An Overview of the Studied Projects

System	Description	#Files	#LOC	#Code Tokens	#Unique Tokens	%Defective Files	%Defective Lines	Studied Releases
ActiveMQ	Messaging and Integration Patterns	1,884-3,420	142k-299k	141k-293k	16,312	2%-7%	0.08%-0.44%	5.0.0, 5.1.0, 5.2.0, 5.3.0, 5.8.0
Camel	Enterprise Integration Framework	1,515-8,846	75k-485k	94k-621k	12,202	2%-8%	0.09%-0.24%	1.4.0, 2.9.0, 2.10.0, 2.11.0
Derby	Relational Database	1,963-2,705	412k-533k	251k-329k	63,677	6%-28%	0.10%-0.63%	10.2.1.6, 10.3.1.4, 10.5.1.1
Groovy	Java-syntax-compatible OOP	757-884	74k-93k	58k-68k	16,901	2%-4%	0.10%-0.17%	1.5.7, 1.6.0.Beta_1, 1.6.0.Beta_2
HBase	Distributed Scalable Data Store	1,059-1,834	246k-537k	149k-257k	30,155	7%-11%	0.17%-1.02%	0.94.0, 0.95.0, 0.95.2
Hive	Data Warehouse System for Hadoop	1,416-2,662	290k-567k	147k-301k	29,245	6%-19%	0.31%-2.90%	0.9.0, 0.10.0, 0.12.0
JRuby	Ruby Programming Lang for JVM	731-1,614	106k-240k	72k-165k	21,630	2%-13%	0.03%-0.09%	1.1, 1.4, 1.5, 1.7
Lucene	Text Search Engine Library	805-2,806	101k-342k	76k-282k	17,733	2%-8%	0.07%-0.39%	2.3.0, 2.9.0, 3.0.0, 3.1.0
Wicket	Web Application Framework	1,672-2,578	106k-165k	93k-147k	21,993	2%-16%	0.05%-0.46%	1.3.0.beta1, 1.3.0.beta2, 1.5.3

In this paper, we began with conducting a survey study to investigate the state-of-practice of code inspection in modern code review, and practitioners' perception on a line-level defect prediction tool. We recruited participants through social media platforms targeting professionals and software developers (i.e., LinkedIn, Facebook Groups). Through an analysis of the 36 responses obtained from the participants, we addressed the following two motivating questions (MQs):

(MQ1) How do practitioners perform code inspection in modern code review process?

Results. Given a changed file in a pull request, 80.6% of the respondents currently inspect source code in a top-down order. Once defective lines are identified, 72.2% of the respondents inspect the defective lines and their related method calls, while 52.8% of the respondents inspect the defective lines and their surrounding lines. Also, 50% of the respondents spent at least 10 minutes to more than one hour to review a single file, indicating that current code review activities are still time-consuming. Importantly, 64% of the respondents perceived that code inspection activity is very challenging to extremely challenging.

(MQ2) Would a line-level defect prediction tool be helpful for practitioners?

Results. 44% of the respondents perceived that a line-level defect prediction tool would potentially be helpful in identifying defective lines. In addition, 64% of the respondents would consider using the tool if it is publicly available for free.

Motivated by the findings from our survey with practitioners, we present DeepLineDP, a deep learning approach to learn the surrounding code tokens and the surrounding lines of source code in order to predict defective files and locate defective lines, which performs as follows. First, we generate a vector representation for each code token. Then, we employ a bidirectional GRU unit [4] to capture the surrounding tokens of that source code line. Next, we employ a Hierarchical Attention Network (HAN) architecture [76] to learn the hierarchical structure of source code to predict defective files. After that, we compute the risk score of code tokens that contribute to the prediction of a given defective file using the attention mechanism [67]. We then generate a ranking of risky lines using the summation of the risk score of the code tokens that appear in that line. Through a case study of 32 releases of 9 software projects, we address the following four research questions (RQs):

(RQ1) Can our DeepLineDP be used to differentiate the riskiness of code tokens in defective and clean lines?

Results. The risk score of the same code token that appears in different lines varies greatly depending on their locations, since we observe a maximum difference of 1. For example, the risk score of a code token may be extremely risky in one line (a risk score of 1), while not being risky at all in another line (a risk score of 0). In addition, we find that the risk score of code tokens in defective lines are significantly higher than the risk score of code tokens in clean lines, suggesting that our DeepLineDP approach can be used to differentiate the riskiness of code tokens in defective and clean lines.

(RQ2) What is the accuracy of our DeepLineDP for predicting defective files?

Results. Our DeepLineDP achieves a median AUC of 0.81 and a median Balanced Accuracy of 0.63, which is 17%-37% and 3%-26% more accurate than the state-of-the-art in terms of the median AUC and the median Balanced Accuracy, respectively. The ScottKnott ESD test also confirms that our DeepLineDP approach always appears at the top rank for both measures.

(RQ3) What is the cost-effectiveness of our DeepLineDP for locating defective lines?

Results. Our DeepLineDP is 47%-250% more cost-effective than the state-of-the-art line-level defect prediction approaches for Recall@Top20%LOC, while achieving 38% and 49% less Effort@Top20%Recall than using the ErrorProne static analysis tool and the N-gram model, respectively.

(RQ4) What is the accuracy of our DeepLineDP for line-level cross-project defect predictions?

Results. Our DeepLineDP models are transferable to other software projects, since our DeepLineDP models still achieve an AUC of 0.63-0.79 and a Recall@Top20%LOC of 0.31-0.46. However, they can be slightly less accurate and less cost-effective than the models that are trained from the previous release of its own project. Nevertheless, our DeepLineDP models still achieve a reasonable performance.

Novelty and Contributions. To the best of our knowledge, the novelty and main contributions of our work are as follows:

1) A survey with 36 practitioners to understand their current code inspection practices in modern code review and their perceptions to adopt a line-level defect prediction.
2) DeepLineDP—a deep learning-based approach for line-level defect prediction that aims to automatically learn the surrounding tokens and surrounding lines to predict defective files and defective lines.
3) An extensive experiment using both within-project and cross-project evaluation settings at the file and line levels with 3 traditional measures and 3 effort-aware measures.
4) Our empirical finding shows that our DeepLineDP models (1) can differentiate the riskiness of the same code token that appears in different lines; (2) can accurately predict the defective files; (3) can effectively generate the ranking of defective lines; and (4) trained on a software project can be transferable to other software projects while achieving a reasonable performance.

To foster the replication of our study, we publish the implementation of our DeepLineDP and the baselines at https://github.com/awsm-research/DeepLineDP.

Paper Organization. The rest of this paper is organized as follows. Section 2 presents a survey study of the current code inspection practices in modern code review. Section 3 presents the architecture of our DeepLineDP approach. Section 4 presents the experimental design and results. Section 5 discloses the threats to the validity. Section 6 discusses the related work. Finally, we draw conclusions in Section 7.

2 A Motivating Survey

In this section, we aim to investigate (1) how practitioners perform code inspection in modern code review and (2) would a line-level defect prediction tool be helpful for practitioners. Below, we described the approach to conduct a survey and presented the results from our survey study.

2.1 Approach

Similar to Kitchenham and Pfleeger [32], we conducted our study according to the following steps: (1) design and develop a survey, (2) recruit and select participants, and (3) verify data and analyze data. We explained the detail of each step below.

(Step 1) Design and Develop a Survey. The purpose of our survey is to investigate the current practices in modern code review process. We designed our survey as a cross-sectional study where participants provided their responses at one fixed point in time. The survey consists of 7 closed-ended questions and 6 open-ended questions. For closed-ended questions, we used multiple-choices question and Likert scale from 1 to 5. Our survey consists of three parts: preliminary question; understanding current code inspection practices; and practitioners' perceptions on a line-level defect prediction tool. Our preliminary question starts with (*"Do you perform code review"*) to ensure that our survey results obtained from the right target participants, followed by roles, levels of experience, and primary programming language skills. The next part is focused on a set of questions to understand the current code inspection practices. Then, the final part is focused on practitioners' perceptions on a line-level defect prediction tool. To do so, we presented a usage scenario and an example visualization. Fig. 1 is an example defective file that is correctly predicted by our DeepLineDP. The shade colour of each code token varies based on the risk score from dark red (very risky) to light red (less risky). The example was obtained from the file `../store/Directory.java` from the release 3.1 of the Lucene project, which has 426 lines of code. With our DeepLineDP, the model correctly predicts that the line number of 227-228 are the most risky lines with a risk score of 0.99 and 0.97, respectively due to the tokens `createOutput()` and `openInput()`. We used Google Form to create our online survey. When accessing the survey, each participant was provided with an explanatory statement that describes the purpose of the study, why the participant is chosen for this study, possible benefits and risks, and confidentiality. The survey takes approximately 10 minutes to complete and is anonymous. Our survey has been rigorously reviewed and approved by the Monash University Human Research Ethics Committee (MUHREC ID: 30739).

(Step 2) Recruit and Select Participants. We recruited the practitioners via LinkedIn since we can specify the target group that we would like to access. We then sent a survey invitation to the target groups via the LinkedIn direct message. To ensure that our survey is not biased, we selected participants from various large companies. Due to the limitation of the number of invitations in LinkedIn, we were able to sent at most 100 invitations per week. In total, we sent a survey invitation to 100 practitioners via LinkedIn. We received a response rate of 15% ($\frac{15}{100}$). Then, we also received 28 additional responses

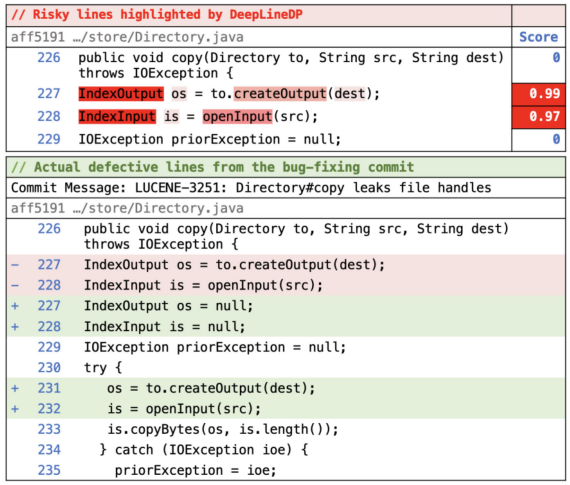


Fig. 1. An example visualization to highlight the most risky tokens obtained from our DeepLineDP approach.

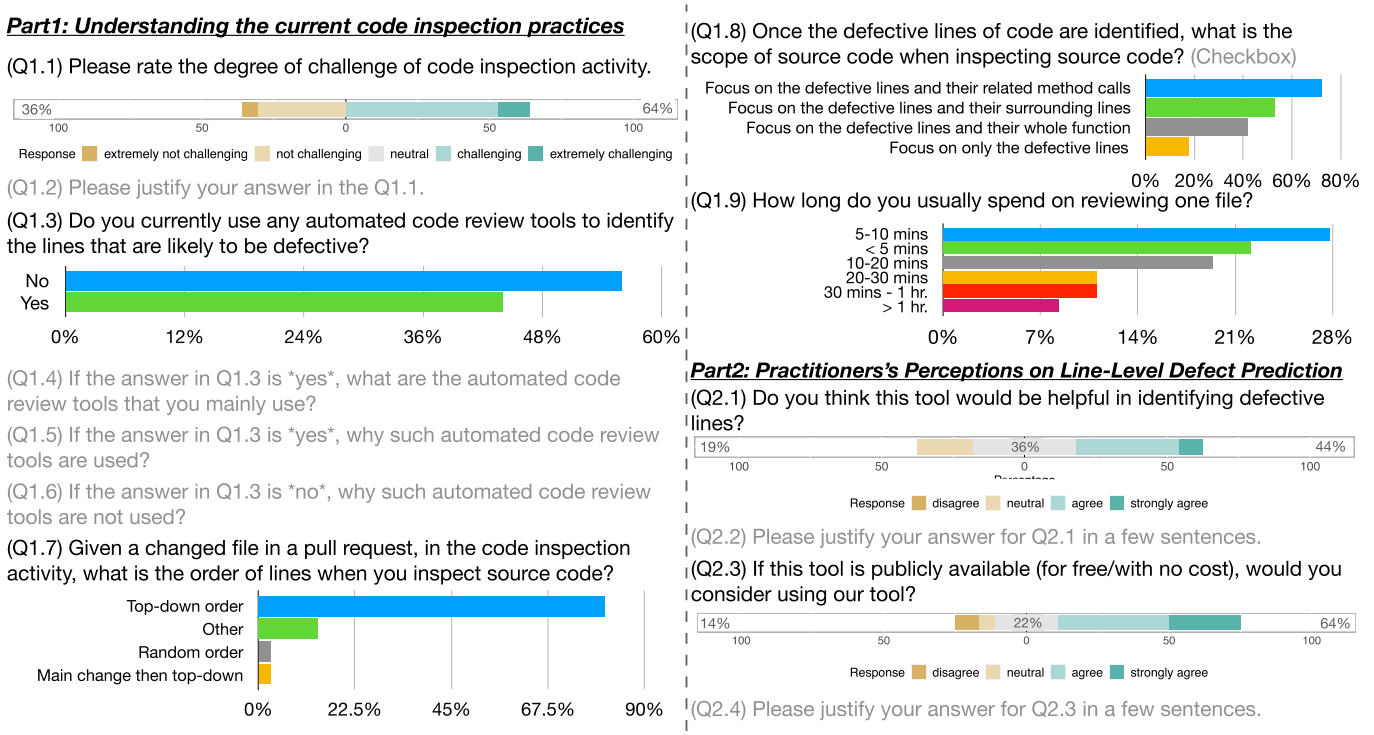


Fig. 2. (MQ1/MQ2) A summary of the survey questions and the results obtained from 36 participants.

through the survey advertisement at several software developer's communities (e.g., Facebook groups). Finally, we obtained a total of 43 responses over one week recruitment.

(Step 3) Verify and Analyze Data. To verify the completeness of the response in our survey (i.e., whether all questions were appropriately answered), we manually read all of the open-ended questions. to ensure that our survey results are derived from the right target participants, we excluded 7 responses that do not perform code review. Finally, we obtained a set of 36 responses. We presented the results of closed-ended responses in a Likert scale with stacked bar plots. We manually analyzed the responses of the open-ended questions to extract in-depth insights.

2.2 Respondent Demographics

Among all of the 36 respondents in the survey, they have the following roles in a software development team: (full-stack) software engineer or software developer (88.8%), quality assurance engineer (5.6%) , team leader (2.8%), others (2.8%). The respondents have different Years of Professional Experience: less than 5 years (52.8%), 6-10 years (33.3%), 11-15 years (8.3%), 16-20 years (2.8%), more than 20 years (2.8%) The respondents described their experience in programming languages as Java (24.8%), Javascript (16.7%), C# (16.7%), Python (13.9%), C/C++ (5.6%), Go (8.3%), Kotlin (5.6%), PHP (2.8%), Scalar (2.8%), Swift (2.8%). These demographics indicate that the responses are collected from practitioners who have a variety of roles, years of experiences, and various programming language expertise. Thus, our findings are likely not bound to specific groups of practitioners.

2.3 Survey Results

Fig. 2 shows a summary of the survey questions and the survey results for MQ1 and MQ2.

(MQ1) How do practitioners perform code inspection in modern code review process?

*Findings. Developers inspect source code in a top-down order. Once defective lines are identified, they will inspect the method calls and their surrounding lines. Each file often takes at least 10 minutes to more than one hour.*To speed up code review process, automated code review tools are currently used. However, 64% of the respondents still perceived that their current code inspection activity remains very challenging, highlighting the need of a line-level defect prediction tool to help developers identify the areas of code that are likely to be defective in the future. Below, we summarize the key findings from our survey.

- Given a changed file in a pull request, 80.6% of the respondents currently inspect source code in a top-down order (i.e., from the top to the bottom). Other respondents inspect source code in a random order, or inspect the main code change followed by the top-down order.
- Once the defective lines are identified, 72.2% of the respondents focused on inspecting the defective lines and their related method calls, while 52.8% of the respondents inspect the defective lines and their surrounding lines, 41.7% of the respondents inspect the defective lines and their whole function, and 17.6% of the respondents inspect only the defective lines.
- 50% of the respondents spent at least 10 minutes to more than one hour to review a single file, indicating that current code review activities are still time-consuming.
- To aid code review process, 44.4% of the respondents currently use automated code review tools (e.g., SonarQube and ESLint), since such tools can facilitate their code review tasks (R10: *It provides massive rules for code reviews, support many languages and the*

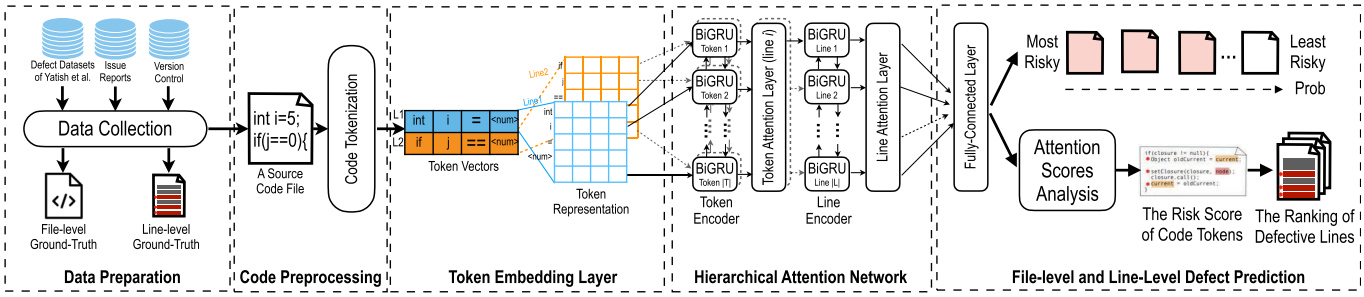


Fig. 3. An overview diagram of our DeepLineDP approach.

review report is intelligible, R25: *Help us highlight and simplify the work that they need to do*.)

- 64% of the respondents perceived that code inspection activity is very challenging to extremely challenging. They stated the reasons that it's challenging to review code that they do not own (R20: *It's challenging when I have to review the code in the repo that I do not own*), understanding the logic behind code is time consuming (R24: *People have different way of thinking. My time is usually spent on understanding their reasoning behind the code before I leave a comment to challenge their design or logic.*), and there are a lot of code to review (R26: *Many lines of code to be reviewed*).

(MQ2) Would a line-level defect prediction tool be helpful for practitioners?

Findings. *44% of the respondents perceived that a line-level defect prediction tool would potentially be helpful in identifying defective lines*, due to various reasons:

- Reduce time to do code review (R18: *It could be useful and take less time to review the code*).
- Easy to understand (R8: *the highlight and score are clear and easy to understand*, R26: *Red alert attracts your eyes so it is easier to spot what is wrong*).
- Provide some useful information (R14: *Show highlight with probability is informative*).

In addition, we find that the respondents who are (full-stack) software engineers or software developers, have less than 5 years of experience, and focus on the surrounding lines and the whole function of the identified defective lines, tend to perceive that the tool is helpful in identifying defective lines.

64% of the respondents would consider using a line-level defect prediction tool if it is publicly available for free, due to various reasons:

- Expedite software development process (R12: *If it make process go faster, why not?*).
- Interesting (R14: *It's interesting to use the tools*).

3 Learning the Surrounding Tokens and Surrounding Lines for Line-Level Defect Prediction

In this section, we present DeepLineDP, an approach to address the challenges of line-level defect prediction. Our DeepLineDP approach is designed to capture defective code lines through the use of bidirectional GRU unit to learn the surrounding tokens and surrounding lines using a hierarchical attention network. Fig. 3 provides an overview of our approach.

Overview. Our approach begins with data collection and data preparation to generate datasets that contains source code files, file-level and line-level ground-truth labels (i.e., files that are affected by a post-release defect and lines that changed or deleted to address the defect). For each source code file, we perform several preproceessing steps, including code abstraction and vocabulary size management. Next, we perform source code representation to generate a vector representation. After that, we use a hierarchical attention network to learn the hierarchical structure of source code for file-level defect prediction, while enabling line-level defect prediction. Then, the prediction layer produces the probability score of the prediction of defective files. Finally, we use the token-level attention layer to identify defective lines based on the most important tokens that contribute to the prediction of defective files.

3.1 Source Code Preprocessing

The goal of the source code preprocessing steps is to extract syntactic, semantic, and contextual code features of each code token. For each Java file, we use JavaParser[1] to build an Abstract Syntax Tree (AST) to generate a stream of tokens and determine their type by extracting syntactic information (e.g., whether a token represents an identifier, a method declaration). Then, each source code file is parsed into a set of lines and each line is parsed into a sequence of code tokens.

Vocabulary Size Management. A large vocabulary size often requires high computing time and large memory usage [9], [42], [64], [74]. To alleviate this issue, we replace integers, real numbers, exponential notation, and hexadecimal numbers with a generic ⟨*num*⟩ token, and replace constant strings with a generic ⟨*str*⟩ token. We remove special characters (i.e., {. , " ' ( ) : (space) } !) since Rahman and Rigby [52] found that these special tokens introduce noise to prediction models. We replace tokens which exist in test sets but do not exist in the training set with a special token ⟨*unk*⟩, known as the out-of-vocabulary problem. We ignore blank lines, as they do not contribute to the actual behavior of the code.

3.2 Token Embedding Layer

Unlike prior studies that represent source code as a long-term sequence [9], our goal is to maintain the hierarchical structure

1. http://javaparser.org/

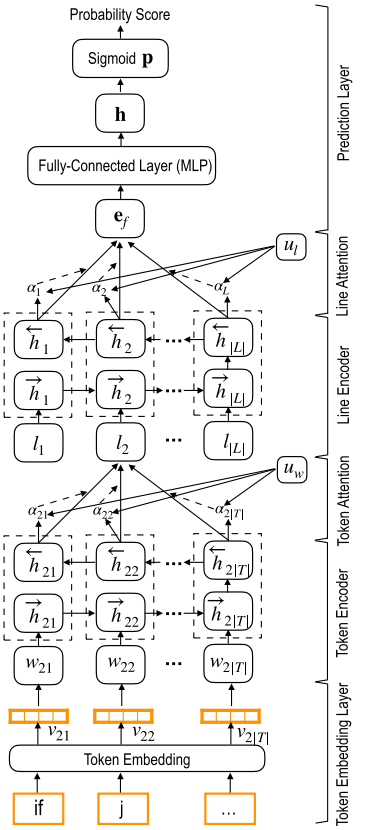


Fig. 4. An overview of a two-layer hierarchical attention network to learn the hierarchical structure of source code using a bidirectional GRU unit to capture the code context (i.e., surrounding tokens).

of source code (i.e., tokens forming lines and lines forming files). Thus, we represent the source code file by maintaining the order of lines and tokens in the original code. Each file is now a sequence of lines $\langle l_1, l_2, \ldots, l_n \rangle$ and each line is a sequence of code tokens $\langle w_1, w_2, \ldots, w_n \rangle$. For each token, we generate a vector representation using the Word2Vec function provided by the gensim Python library. In particular, we build a project-specific language model using our training dataset for each project to ensure that the vector representation is derived from the domain-specific vocabularies and to obtain the optimal distributed representations. That means the vector representations generated from domain-specific language models tend to produce more meaningful vectors (i.e., capturing better distributed relationships between tokens and their surrounding tokens better than a generic pre-trained language model). When training the Word2Vec model, we use the Continuous Bag of Words (CBOW) architecture to learn the distributed representations of tokens, since the CBOW architecture considers the surrounding tokens to generate a vector representation of the target token.

3.3 Learning the Hierarchical Structure of Source Code

We use Hierarchical Attention Network [76] to learn the hierarchical structure of source code (see Fig. 4). Specifically, we use a two-layer attention network (a token layer and a line layer). This network consists of four parts: a token encoder, a token-level attention layer, a line encoder, and a line-level attention layer. Assuming that a source code file $f \in \mathcal{F}$ has a sequence of lines $\mathcal{L} = [l_1, l_2, \ldots, l_{|\mathcal{L}|}]$, where l_i contains a sequence of tokens $\mathcal{T} = [w_{i1}, w_{i2}, \ldots, w_{i|\mathcal{T}|}]$, where w_{it} represents the code token in the i-th line, $t \in [1, |\mathcal{T}|]$.

Token Encoder. Given a line l_i with a sequence of tokens $\mathcal{T}$ and a word embedding matrix $\mathbf{W} \in \mathbb{R}^{|\mathcal{V}^{\mathcal{C}}| \times d}$, where $\mathcal{V}^{\mathcal{C}}$ is the vocabulary containing all tokens extracted from training source code files and d is the word embedding size of the representation of tokens, we first obtain a vector representation of each token for each line $v_{it} = \mathbf{W}(w_{it})$ from Word2Vec, where v_{it} indicates the vector representation of token w_{it} in the word embedding matrix $\mathbf{W}$.

We employ a bidirectional GRU [4] to summarize information from the context of a token in both directions. GRU is proposed to solve the vanishing gradient problem, which is commonly found in a standard RNN architecture when training with a long sequence of tokens. To capture the contextual information, the bidirectional GRU includes a forward GRU $\overrightarrow{h}_{it} = \overrightarrow{\text{GRU}}(v_{it}), t \in [1, |\mathcal{T}|]$ that reads the line l_i from left (w_{i1}) to right ($w_{i|\mathcal{T}|}$) and a backward GRU $\overleftarrow{h}_{it} = \overleftarrow{\text{GRU}}(v_{it}), t \in [|\mathcal{T}|, 1]$ that reads the line l_i from right ($w_{i|\mathcal{T}|}$) to left (w_{i1}). We obtain an annotation for a given token w_{it} by concatenating the forward hidden state $\overrightarrow{h}_{it}$ and backward hidden state $\overleftarrow{h}_{it}$, i.e., $h_{it} = [\overrightarrow{h}_{it} \oplus \overleftarrow{h}_{it}]$.

Token Attention. Based on the intuition that not all tokens contribute equally to the semantic representation of the source code line, we use the attention mechanism [67] to highlight the tokens that are more important to the semantics of the source code lines and aggregate the representation of those informative tokens to form a line vector.

We first feed the token annotation h_{it} through a one-layer Multi-Layer Perceptron (MLP) to get a hidden representation (u_{it}) of h_{it}, i.e., $u_{it} = \tanh(\mathbf{W}_w h_{it} + b_w)$. Similar to prior studies [19], [76], we define a token-level context vector (u_w) that can be seen as a high level representation of the answer to the fixed query "what is the most informative token" over the tokens. The token context vector u_w is randomly initialized and learned during the training process. Then, we measure the importance of the token as the similarity of u_{it} with a token-level context vector u_w. Then, we compute a normalized contribution (attention) α_{it} of token w_{it} of the line l_i through a softmax function [7]: $\alpha_{it} = \frac{exp(u_{it}^T u_w)}{\sum_t exp(u_{it}^T u_w)}$. For each line l_i, its vector is computed as a weighted sum of the embedding vectors of the tokens based on their importance as follows: $s_{it} = \sum_t \alpha_{it} h_{it}$.

Line Encoder. Given source code lines (i.e., l_i), we use a bidirectional GRU to encode the lines as follows: $\overrightarrow{h}_i = \overrightarrow{\text{GRU}}(l_i), i \in [1, |\mathcal{L}|]$ and $\overleftarrow{h}_i = \overleftarrow{\text{GRU}}(l_i), t \in [|\mathcal{L}|, 1]$. Similar to the Token Encoder, we obtain an annotation of the source code line l_i by concatenating the forward hidden state $\overrightarrow{h}_i$ and the backward hidden state $\overleftarrow{h}_i$ of this line. The annotation of the line l_i is denoted as $h_i = [\overrightarrow{h}_i \oplus \overleftarrow{h}_i]$, which summarizes the line l_i considering its neighboring lines.

Line Attention. We again use an attention mechanism to highlight the lines that are more important to the semantics of the source code files and aggregate the representation of those informative lines to form a file vector. We first define a line-level context vector $u_l = \tanh(\mathbf{W}_l h_i + b_s)$. Then, we compute a normalized contribution (attention) α_i of line u_l of the line l_i through a softmax function [7]: $\alpha_i = \frac{exp(u_l^T u_s)}{\Sigma_i exp(u_l^T u_s)}$.

Here u_s is a sentence-level context vector which is compared with u_l to measure the importance of a code line. For each file, $\mathbf{e}_f$ is the embedding vector of the file that is computed as a weighted sum of the embedding vectors of the lines based on their importance as follows: $\mathbf{e}_f = \Sigma_i \alpha_i h_i$.

3.4 File-Level and Line-Level Defect Prediction Layer

The embedding vector $\mathbf{e}_f$ is a high level representation of the source code file which can be used as features for file-level defect prediction. This vector is fed to a one-layer Multi-Layer Perceptron (MLP) (a.k.a. a fully-connected layer) to produce a prediction score $\mathbf{h} = \mathbf{w}_h \cdot \mathbf{e}_f + b_h$, where $\mathbf{w}_h$ is the weight matrix used to connect the embedding vector $\mathbf{e}_f$ with the hidden layer, and $\mathbf{b}_h$ is the bias value. Finally, the prediction score is passed to an output layer to compute a probability score for a given source code file. We use Sigmoid function to compute the predicted probability of a given source code file as follows: $\mathbf{p}(y_i = 1|f_i) = \frac{1}{1+\exp(-\mathbf{h}\cdot\mathbf{w}_0)}$, where $y_i \in \mathbf{Y}$ is the probability score of the ith file and f_i is the file that we want to predict.

To identify defective lines, we first extract the attention score, ranges from -1 to 1, of each code token in that defective file. The attention score, calculated by the attention mechanism [67], is used as a proxy to indicate the riskiness of code tokens. The concept of the attention score is similar to the importance score of the random forest's variable importance that is widely-used in software engineering, but having different calculations. To do so, we rank the attention score obtained from the token attention layer in a descending order. Then, we select the top-k tokens that have the highest attention scores. Then, we compute the line-level risk score as the summation of the risk score of any tokens that appear in the top-k. Finally, we produce a ranking of the most risky lines based on the risk scores of each line. In this paper, we choose the k of 1,500.

4 Study Design and Results

The goal of this paper is to empirically assess our hypothesis whether the hierarchical structure and code context (i.e., surrounding tokens and surrounding lines) should be considered to identify the fine-grained locations of defective code (i.e., defective files and defective lines). Below, we present the study design followed by the results.

Line-Level Defect Datasets. In this paper, we use the benchmark line-level defect dataset prepared by Wattanakriengkrai *et al.* [73]. The dataset consists of 32 software releases that span across 9 open-source software systems. Table 1 shows a statistical summary of our studied dataset. Each release contains 731 to 8,846 files, 74,349-567,804 lines of code, and 58,659-621,238 code tokens. Below, we discuss the detailed steps that were used to collect file-level and line-level ground-truths.

(Step-1) Collect issue reports. For each studied system, all issue reports were retrieved from the JIRA Issue Tracking System (ITS). Then, the unique identifier of the issue reports (IssueID), the issue report type (e.g., a bug or a new feature), and the affected releases (i.e., the releases that are affected by a given issue report) were extracted. Since the goal of release-based defect prediction is to predict if a file will be affected by an issue report in the future, only the issue reports that were classified as bug, were reported after the studied release, and affected the studied releases were focused.

(Step-2) Collect a snapshot of source code. For each release, a snapshot of source code at the release date (i.e., the list of files of a given release) was collected from the Git Version Control System (VCS).

(Step-3) Identify Defect-Fixing Commits. Defective files are defined as files that are affected by post-release defects [44], [77]. To identify defective files, commits that addressed a post-release defect (i.e., defect-fixing commits) were needed to be found. To do so, defect-fixing commits that are associated with the defect reports that affected the studied release were found, using regular expressions to search for the issue IDs in the commit messages. Then, files that are fixed for the defect reports were labelled as defective, otherwise clean.

(Step-4) Identify Defective Lines in the Code Snapshot at the Release Date. Defective lines are defined as lines in the code snapshot at the release date that are affected by a post-release defect report [51], [54]. To identify defective lines, the defective files (in the defect-fixing commits) were examined and identified which lines in the code snapshot at the release date are fixed or modified (i.e., modification or deletion operations) to address defects. Then, lines that are fixed or modified for the post-release defect reports were labelled as defective, otherwise clean.

With these four steps, line-level defect datasets, which is a snapshot of source code at the release date and defective files and defective lines at the release date that later are fixed or modified to address for software defects, could be produced.

Training Details. The implementation of our DeepLineDP model is based on Pytorch, an open-source deep learning framework. The experiment is run on a computer with an AMD Ryzen 9 5950X 16-Core @3.4 GHz, a RAM of 64GB, and an NVIDIA RTX 3090 GPU with 24 GB memory. The average model training time is 5 minutes. Below, we describe the hyperparameter settings that we used for our approach.

We use the word embedding vector size d of 50 to train Word2Vec model. We use the learning rate of 0.001. The model is trained at 10 epochs. The batch size is set to 32. We use the binary cross-entropy as the loss function. We use the Adam optimizer [31] to minimize the loss function, since it has been shown to be computationally efficient and require low memory consumption. We use Dropout (the dropout ratio is 0.5) and Layer Normalization to prevent overfitting. The reason for this is that Watson *et al.* [72] found that the Dropout is one of the most commonly-used techniques to prevent overfitting and underfitting in software engineering tasks. We do not employ the early stopping technique, since we found that the characteristics of our release-based datasets are different from release to release. Thus, the problem of release-based defect prediction is different from other deep learning tasks (e.g., an image classification) where the whole corpus is from the same context.

Hyper-Parameters Tuning. We use the following hyper-parameter when fine-tuning our DeepLineDP models: Bi-

GRU hidden cells of token encoder = $\{32, 64, 128\}$, Bi-GRU hidden cells of line encoder = $\{32, 64, 128\}$, MLP hidden cells of token attention = $\{64, 128\}$, MLP hidden cells of line attention = $\{64, 128\}$, learning rate = $\{0.01, 0.001, 0.0001\}$. During the hyper parameter calibration phase, we start with Bi-GRU hidden cells of token encoder = 64, Bi-GRU hidden cells of line encoder = 64, MLP hidden cells of token attention = 64, MLP hidden cells of line attention = 64 and learning rate = 0.001. We then randomly calibrate each hyper parameter, one at a time, to observe the change of model's performance on the validation set. Then, we selected the best model based on the validation set, not the testing set.

Evaluation Measures. We use traditional measures (i.e., AUC, Balanced Accuracy, MCC) for evaluating file-level defect prediction and effort-aware measures (i.e., Recall@Top20%LOC, Effort@Top20%Recall, Initial False Alarms) for evaluating line-level defect prediction.

AUC is an area under the ROC Curve (i.e., the true positive rate and the false positive rate). AUC values range from 0 to 1, with a value of 1 indicates perfect discrimination, while a value of 0.5 indicates random guessing.

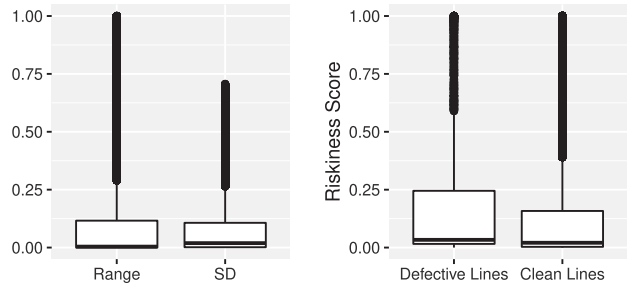
Balanced Accuracy measures the average of True Positive Rate (i.e., how many predicted defective files are correct) and True Negative Rate (i.e., how many predicted clean files are correct). Balanced accuracy is computed as: $(\frac{\text{TP}}{\text{TP+FN}} + \frac{\text{TN}}{\text{TN+FP}})/2$, where TP, TN, FP and FN refer to True Positive, True Negative, False Positive, and False Negative, respectively. A high balanced accuracy indicates that an approach can accurately predict defective files and clean files.

Matthews Correlation Coefficients (MCC) measures a correlation coefficients between actual and predicted outcomes using the following calculation: $\frac{TP \times TN - FP \times FN}{\sqrt{(TP+FP)(TP+FN)(TN+FP)(TN+FN)}}$. An MCC value ranges from -1 to +1, where an MCC value of 1 indicates a perfect prediction, and -1 indicates total disagreement between the prediction.

Recall@Top20%LOC measures how many defective lines that can be accurately found when inspecting the top 20% LOC of the whole release. A high value of Recall@Top20%LOC indicates that our approach can rank many actual defective lines at the top and many actual defective lines can be found given a fixed amount of effort. On the other hand, a low value of Recall@Top20%LOC indicates that many clean lines are in the top-20% LOC and developers need to spend more effort to identify defective lines.

Effort@Top20%Recall measures how much effort (i.e., LOC) required to find the 20% actual defective lines of the whole release. A low value of Effort@Top20%Recall indicates that developers spend a small amount of effort to find the top-20% actual defective lines. On the other hand, a high value of Effort@Top20%Recall indicates that developers spend a large amount of effort to find the top-20% actual defective lines.

Initial False Alarm measures the number of clean lines that developers need to inspect until the first defective line is found for each file [20]. A low IFA value indicates that few clean lines are ranked at the top, while a high IFA value indicates that developers will spend unnecessary effort on clean lines. The intuition behinds this measure is that developers may stop inspecting if they could not get promising results (i.e., find defective lines) within the first few inspected lines [46].



(a) The distribution of the range (i.e., Max-Min) of the risk scores of code tokens in a file.

(b) The distribution of the risk scores between actual defective lines and actual clean lines.

Fig. 5. (RQ1) The variation of the risk scores of code tokens in defective lines and clean lines.

Note that we do not measure AUC, MCC and Balanced Accuracy for line-level defect predictions since we formulate line-level prediction as a ranking task. In contrast, AUC, MCC and Balanced Accuracy measures are designed for classification tasks. Thus, these measures are not applicable to our line-level defect prediction.

Below, we present the approach and results of our four research questions (RQs).

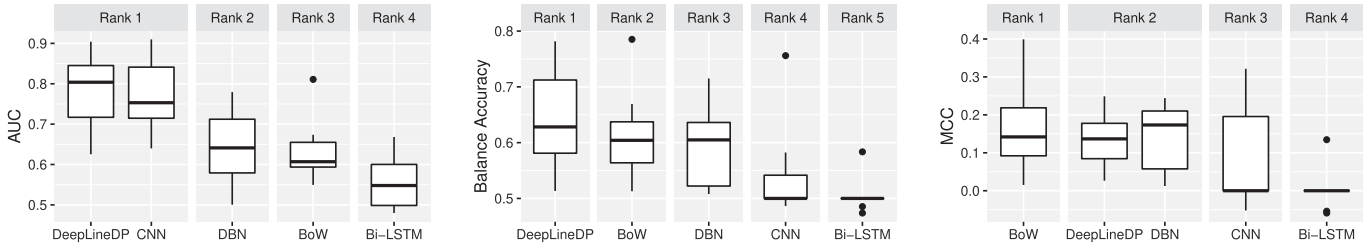
(RQ1) Can our DeepLineDP be used to differentiate the riskiness of code tokens in defective and clean lines?

Motivation. Not all code tokens and all lines in a defective file are defective. A code token that appears in different lines may have different riskiness. Thus, we investigate if the risk score of code tokens varies depending on their locations.

Approach. We aim to investigate if the risk score of the same code token at different lines varies depending on their locations. To answer this RQ, we use the attention scores calculated by the attention mechanism as a proxy to indicate the risk of code tokens that contribute to the prediction of that defective file. We hypothesize the same code token at different lines should have different riskiness. Thus, we first analyze the variation of the risk scores of each unique code token that appears in different lines in a file using the range (i.e., max-min) and the standard deviation (S.D.). Finally, we analyze the distribution of the risk scores of code tokens that appear in both defective lines and clean lines.

Results. The risk score of the same code token that appears in different lines varies greatly depending on their locations. Fig. 5a shows that the range of the risk scores varies from 0 to 1, where its standard deviation varies from 0 to 0.73. A range of 1 indicates that the risk score of a code token may be extremely risky in one line, while not being risky at all in another line. We find that there are 26% of code tokens that their risk score varies more than 5% when appearing in different lines, meaning that the risk score of such code tokens varies depending on their locations. This finding suggests that our DeepLineDP approach can be used to differentiate the riskiness of code tokens in defective and clean lines.

In addition, the risk score of code tokens in defective lines is significantly higher than the risk score of code tokens in clean lines. Fig. 5b presents the distribution of the risk score of code



(a) AUC-ROC (↗). (b) Balanced Accuracy (↗). (c) Mathew Correlation Coefficients (↗).

Fig. 6. (RQ2) The ScottKnott ESD ranking and the distributions of the AUC, Balanced Accuracy, and Mathew Correlation Coefficients (MCC) of our DeepLineDP and the state-of-the-art file-level defect prediction approaches. The higher (↗) the values are, the better the approach is.

tokens that appear between actual defective lines and actual clean lines. The Mann-Whitney U test confirms that the risk score of code tokens in defective lines is significantly higher than the risk score of code tokens in clean lines (p-value < 0.001) with a Cliff's δ effect size of small (δ = 0.18-0.20).

(RQ2) What is the accuracy of our DeepLineDP for predicting defective files?

Motivation. The state-of-the-art file-level defect prediction approaches [9], [34], [70], [71] do not consider the surrounding code lines when predicting defective files. Thus, we investigate if our DeepLineDP outperforms the state-of-the-art file-level defect prediction.

Approach. To answer this RQ, we evaluate our DeepLineDP using cross-release evaluation setting for each project (i.e., the first release R_1 is used for training, R_2 is used for validation, while the subsequent releases $R_3, R_4, \ldots, R_n$ are used for testing). Therefore, we have a total of 14 train-test evaluation combinations (i.e., 3 for ActiveMQ + 2 for Camel + 2 for JRuby + 2 for Lucene + 1×5 for Derby, Groovy, Hbase, Hive and Wicket). Then, we compare our DeepLineDP with the four state-of-the-art file-level defect prediction approaches:

1) BoW uses the frequency of the code tokens (aka. Bag-of-Word) as textual features to predict defective files [14]. Similar to prior work, we use a logistic regression classification technique to train the BoW features. To handle the class imbalance problem, we applied the SMOTE technique [8] to rebalance the training datasets.
2) DBN is a Deep Belief Networks architecture that automatically learns semantic features for predicting defective files [70], [71].
3) CNN is a Convolutional Neural Network (CNN) architecture that automatically learns semantic and structural features to predict defective files [33], [35].
4) BiLSTM is a bidirectional Long Short-Term Memory architecture that automatically learns semantic and syntactic features to predict defective files [9], [36], [37], [80].

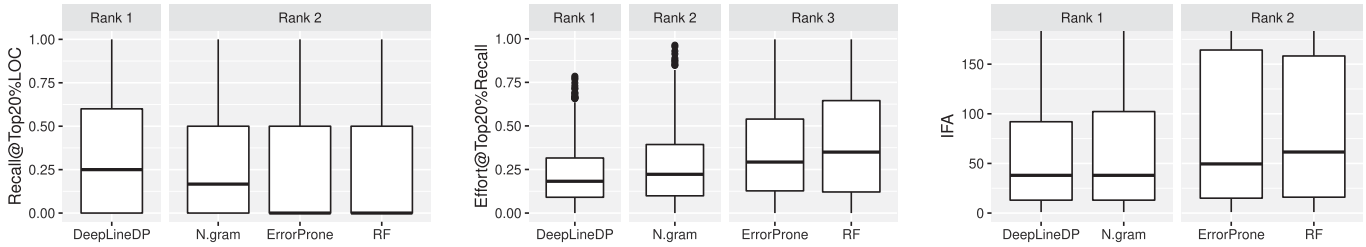
Then, we evaluate these approaches using the 3 traditional evaluation measures (i.e., AUC-ROC, Balanced Accuracy, and Matthews Correlation Coefficients (MCC)). We compute the percentage improvement (%) as follows: $\frac{(P_{\text{ours}} - P_{\text{baseline}})}{P_{\text{baseline}}} \times 100\%$. Finally, we apply a ScottKnott ESD test to cluster the distributions into statistically distinct ranks with non-negligible effect size difference [59], [60], [61]. The ScottKnott ESD test produces the ranking of the techniques while ensuring that (1) the magnitude of the difference for all of the distributions in each rank is negligible; and (2) the magnitude of the difference of distributions between ranks is non-negligible. Fig. 6 presents the ScottKnott ESD ranking and the distributions of the AUC, Balanced Accuracy, and Mathew Correlation Coefficients (MCC) of our DeepLineDP approach and the state-of-the-art file-level defect prediction approaches.

Results. Our DeepLineDP is 17%-37% and 3%-26% more accurate than the state-of-the-art in terms of the median AUC and the median Balanced Accuracy. According to the result, the median of AUC is at 0.81 for our DeepLineDP, 0.75 for CNN, 0.64 for DBN, 0.61 for BoW, and 0.55 for BiLSTM models. We also observe a median Balanced Accuracy at 0.63 for our DeepLineDP, 0.61 for DBN, 0.61 for BoW, 0.50 for CNN, and 0.50 for BiLSTM models. These findings indicate that our DeepLineDP approach outperforms the state-of-the-art file-level defect prediction approaches. The ScottKnott ESD test also confirms that our DeepLineDP approach always appears at the top rank in terms of AUC and Balanced Accuracy, indicating that the performance difference is statistically significant with non-negligible effect size. On the other hand, we observed that our DeepLineDP achieves an MCC value slightly lower than the BoW approach, since the MCC measure is designed to capture the False Positive Rates and False Negative Rates, while the Balance Accuracy captures only the True Positive Rate and True Negative Rate. When we perform a deeper investigation in the prediction results, DeepLineDP tends to produce more false positive than the BoW (i.e., incorrectly predicts clean files as defective). Nevertheless, in a deployment scenario, developers often consider the ranking of the defective files (i.e., based on a ranking of probabilities), rather than the binary classification of the defective and clean files (i.e., based on a threshold of 0.5). Thus, when defective files are ranked based on the probabilities, AUC that captures the ability to distinguish defective and clean files still confirm that our DeepLineDP still outperforms other techniques.

(RQ3) What is the cost-effectiveness of our DeepLineDP for locating defective lines?

Motivation. Line-level defect prediction is needed to help developers identify the fine-grained locations of defective code, instead of wasting their time inspecting clean lines. However, there exists only few approaches for prioritizing defective lines [11], [16], [54], [69]. Thus, we investigate if our DeepLineDP that is able to locate defective lines is more cost-effective than the state-of-the-art line-level defect prediction.

Approach. To answer this RQ, we investigate if our DeepLineDP, being able to locate defective lines, is more cost-



(a) Recall@Top20%LOC (↗). (b) Effort@Top20%Recall (↘). (c) Initial False Alarms (↘).

Fig. 7. (RQ3) The ScottKnott ESD ranking and the distributions of the Recall@Top20%LOC, Effort@Top20%Recall, and Initial False Alarms of our DeepLineDP and the state-of-the-art line-level defect prediction approaches. The higher (↗) or the lower (↘) the values are, the better the approach is.

effective than the state-of-the-art line-level defect prediction. To evaluate the cost-effectiveness of line-level defect prediction, we only focus on the correctly predicted defective files from our models. For each defective file, we first compute the attention scores using the attention mechanism to indicate the risk score of each token. Then, we compute the summation of the attention scores for each line. After that, we generate the ranking of risky lines for each file based on the risk scores. Finally, we compare the ranking of risky lines with the three state-of-the-art line-level defect prediction approaches:

1) N-gram is used to infer the unnaturalness of each code token [16], [54], [69] based on the entropy score of each code token. We selected N-gram as our baseline, since prior studies showed that N-gram is effective in identifying defective lines [54], [69] (i.e., high entropy = unnaturalness of code tokens). We use an implementation of Hellendoorn and Devanbu [16] to build cache-based language models, i.e., an enhanced n-gram model that is suitable for source code. The cache-based language models are built by learning from clean code files (i.e., the files that do not contain defective lines). Similar to prior work [16], once the entropy scores for all code tokens are calculated, the average of the entropy scores for each line is computed. Then, we rank the source code lines based on the average entropy values of tokens in each line.
2) ErrorProne [1] is a Google's static analysis tool that builds on top of a primary Java compiler (javac) to check errors in source code based on a set of error-prone rules. We use ErrorProne instead of other static analysis tools (e.g., FindBugs) since ErrorProne is able to detect more critical software defects (e.g., compilation defects) than FindBugs [1]. In this experiment, we consider lines as defective if errors are generated by ErrorProne. This mimics a top-down reading approach, i.e., developers sequentially read source code from the top to the bottom of the files.
3) Random Forest (RF) is used to predict which lines in a defective file are clean or defective. The model used DeepLineDP's initial token representation at line level as features to train the model.

We again compute the percentage improvement (%) and apply the ScottKnott ESD test to cluster the distributions into statistically distinct ranks with non-negligible difference. Fig. 7 presents the ScottKnott ESD ranking and the distributions of the Recall@Top20%LOC, Effort@Top20%Recall, and Initial False Alarms of our DeepLineDP and the state-of-the-art line-level defect prediction approaches.

Results. *Our DeepLineDP is 47% - 250% more cost-effective than the state-of-the-art line-level defect prediction approaches in terms of Recall@Top20%LOC*.We find that our DeepLineDP approach achieves a median Recall@Top20%LOC of 0.25, while N-gram achieves a median Recall@Top20%LOC of 0.17, and ErrorProne and RF achieves a median Recall@Top20%LOC of 0 . This finding indicates that, given a fixed amount of 20%LOC inspection effort, our DeepLineDP approach can correctly locate 25% of the actual defective lines, while N-gram can correctly locate only 17%, and ErrorProne and RF can correctly locate 0% of the actual defective lines.

In addition, our DeepLineDP achieves a median Effort@Top20%Recall of 0.18, which is 38% and 49% less than ErrorProne and N-gram, respectively. We find that our DeepLineDP approach achieves a median Effort@Top20%Recall of 0.18, while N-gram, ErrorProne and RF achieve a median Effort@Top20%Recall of 0.22, 0.29 and 0.35, respectively. This finding indicates that, in order to find the 20% actual defective lines, our DeepLineDP approach approximately requires developers to inspect 18% LOC of the whole release, while the N-gram approximately requires 22%, ErrorProne approximately requires 29% and the RF models approximately requires 35%. However, as mentioned in the threat to construct validity, the reported effort may not reflect the actual effort that a developer requires to inspect defective lines.

When inspecting the ranking of risky lines in each defective file, our DeepLineDP and N-gram achieve a median Initial False Alarm of 38, while the ErrorProne and the RF models achieve a median Initial False Alarm of 50 and 62, respectively. According to this finding, the ranking of risky lines generated by our DeepLineDP is better than ErrorProne and N-gram. Thus, developers will spend less inspection effort to find the first actual defective line.

The ScottKnott ESD test also confirms that our DeepLineDP approach always appears at the top rank in terms of Recall@Top20%LOC, Effort@Top20%Recall, and Initial False Alarms with non-negligible effect size. This finding suggests that the hierarchical structure and code context (i.e., surrounding tokens and surrounding lines) can be used to improve the cost-effectiveness of SQA resource management, while saving developers' inspection effort to locate actual defective lines.

(RQ4) What is the accuracy of our DeepLineDP for line-level cross-project defect predictions?

TABLE 2
(RQ4) The Mean Performance of Our DeepLineDP Approach for Within-Project (i.e., Using the Models Trained Using its own Project) and Cross-Project Evaluation (i.e., Using the Models Trained Using Other Projects) and its Percentage Point Difference (δ =Delta Difference)

	AUC (↗)			Balanced Accuracy (↗)			MCC (↗)			Recall@20%LOC (↗)			Effort@20%Recall (↘)			Initial False Alarms (↘)		
Target	Within	Cross	δ	Within	Cross	δ	Within	Cross	δ	Within	Cross	δ	Within	Cross	δ	Within	Cross	δ
activemq	0.87	0.74	-0.13	0.61	0.66	0.05	0.12	0.20	0.08	0.39	0.31	-0.08	0.20	0.24	0.04	49	69	20
camel	0.76	0.74	-0.02	0.70	0.60	-0.10	0.17	0.13	-0.04	0.30	0.46	0.16	0.27	0.20	-0.07	51	59	8
derby	0.85	0.74	-0.11	0.75	0.56	-0.19	0.25	0.07	-0.18	0.47	0.44	-0.03	0.16	0.16	0.00	156	150	-6
groovy	0.84	0.79	-0.05	0.51	0.68	0.17	0.03	0.16	0.13	0.41	0.37	-0.04	0.16	0.19	0.03	69	97	28
hbase	0.65	0.63	-0.02	0.61	0.56	-0.05	0.14	0.08	-0.06	0.27	0.34	0.07	0.24	0.22	-0.02	125	136	11
hive	0.74	0.69	-0.05	0.67	0.60	-0.07	0.23	0.13	-0.10	0.15	0.36	0.21	0.28	0.20	-0.08	137	92	-45
jruby	0.82	0.68	-0.14	0.75	0.57	-0.18	0.17	0.05	-0.12	0.30	0.36	0.06	0.28	0.24	-0.04	211	205	-6
lucene	0.65	0.63	-0.02	0.57	0.54	-0.03	0.07	0.04	-0.03	0.53	0.39	-0.14	0.15	0.19	0.04	63	55	-8
wicket	0.75	0.78	0.03	0.56	0.58	0.02	0.06	0.08	0.02	0.88	0.69	-0.19	0.10	0.12	0.02	29	42	13

The higher (↗) or the lower (↘) the values are, the better the approach is.

Motivation. Intuitively, at the beginning of software development, the amount of historical data is limited. Thus, applying defect prediction to such early-development software is often inaccurate. Thus, prior studies suggested to train a model from a source project and test on the target project (i.e., cross-project defect prediction). Yet, Zimmermann *et al.* [79] found that such simple cross-project prediction scenario often produce inaccurate predictions. Thus, prior studies proposed various advanced approaches to improve the performance of cross-project defect prediction at different granularity levels. For example, Menzies *et al.* [40], Zhang *et al.* [78] and Xia *et al.* [75] proposed class-level cross-project defect prediction approaches. Zimmermann *et al.* [79] and Wang *et al.* [70] proposed file-level cross-project defect prediction approaches. Turhan *et al.* [65] proposed module-level cross-project defect prediction approaches. Li *et al.* [35] and Dam *et al.* [9] proposed method-level cross-project defect prediction approaches. However, none of the prior cross-project defect prediction approaches can predict defects at the line level. Thus, we investigate how accurate our DeepLineDP model is for line-level cross-project defect predictions.

Approach. To answer this RQ, we apply the DeepLineDP models trained on a given source project to the releases of the target projects other than the source project. Next, we evaluate the performance of file-level defect prediction using 3 traditional measures and the performance of line-level defect prediction using 3 effort-aware measures. Then, we compare the performance of cross-project predictions with within-project predictions. In total, we have a combination of 112 train-test evaluation settings. Table 2 presents the median performance of our DeepLineDP approach for within-project (i.e., the models trained using its own project) and cross-project evaluation (i.e., the models trained using other projects) and its percentage point difference (δ = Delta Difference).

Results. *Our DeepLineDP models are transferable to other software projects, since our DeepLineDP models achieve a reasonable AUC of 0.63-0.79.* Table 2 shows that, when the models are trained using other projects, our DeepLineDP models achieve an AUC of 0.63-0.79, a Balanced Accuracy of 0.54-0.68, an MCC of 0.07-0.16. However, the AUC performance of cross-project predictions is still comparable with within-project defect predictions, with a percentage point difference between -0.14 and +0.03. Even though we find that the AUC of the models trained using other projects can be less accurate than the models trained using its own project, we still achieve a reasonable AUC for file-level defect prediction.

Our DeepLineDP models achieve a reasonable Recall@Top20%LOC of 0.31-0.46, an Effort@Top20%Recall of 0.12-0.24, and an Initial False Alarm of 42-205. Table 2 shows that, when the models trained using other projects, our DeepLineDP can still correctly locate 31%-46% of the actual defective lines when inspecting the top 20% LOC. We also find that our DeepLineDP approximately requires 12%-24% LOC effort to find 20% of the actual defective lines. However, our DeepLineDP approach can be more or less cost-effective than the models trained using its own project, since the Recall@Top20%LOC varies between -0.19 and +0.21 when compared to within-project defect predictions. Nevertheless, we still achieve a reasonable Recall@Top20%LOC, Effort@Top20%Recall and an Initial False Alarm for line-level defect prediction. These findings suggest that developers in different software projects may use our DeepLineDP models trained from other projects to identify the fine-grained locations of defective code (i.e., defective files and defective lines).

5 Discussion

In this section, we provide additional discussions and disclose the threats to the validity.

5.1 Implications to Practitioners

Our motivating survey found that 50% of the respondents spend at least 10 minutes to more than one hour to review a single file, while 64% of the respondents perceived that current code inspection activity is very challenging to extremely challenging. Importantly, 44% of the respondents perceived that a line-level defect prediction tool would potentially be helpful in identifying defective lines, and 64% of the respondents would consider using a line-level defect prediction tool if it is publicly available for free. Our results show that our DeepLineDP can accurately predict defective files achieving an AUC of 0.63-0.91, while also accurately predicting defective lines achieving a Recall@Top20%LOC of 0.25, which is

47%–250% more cost-effective than the state-of-the-art approaches. Since the data collection steps and the model training can be automated and the model implementation cost is minimal (e.g., 5 minutes for model training time on a consumer-grade GPU), we believe that the cost of adoption of our approach in software industry would be very minimal.

5.2 DeepLineDP Versus Code Coverage

Generally, code coverage often provides sufficient information for developers to prioritize testing resources on the lines that are not covered by the test cases. Particularly, code coverage is used to indicate the proportion of lines of code that are tested and not tested when executing a set of test suites (i.e, white-box testing), indicating that lines of code that are not executed by test cases may be more defect-prone than the others.

Nevertheless, a software project that achieves 100% code coverage through white-box testing only indicates that the 100% of these lines of code have been executed by the test cases, but it does not always mean that these lines of code will not be defective in the future (i.e., the presence of bugs do not indicate the absence of bugs, since the absent bugs may appear in the future). Similarly, DeepLineDP alone may help developers detect areas that are likely to be defective in the future, but not the presence of defects. Thus, DeepLineDP and code coverage should be used in complementary in order to achieve the highest quality of software systems.

5.3 Threats to Validity

Threats to construct validity relate to the suitability of our evaluation measure. We use only three traditional measures that are less susceptible to class imbalance problems [56] and use three effort-aware measures that are prefered by practitioners [68]. However, other measures can be used for evaluation as well.

R3.16, R3.18Similar to prior work [3], [27], [39], [56], we use LOC as a proxy to measure developers' effort. However, LOC may not reflect actual effort that a developer requires to inspect defective lines. Two files with the same amount of lines of code may have different complexity levels, which requires different amounts of effort to inspect. Thus, other factors like code complexity could be incorporated into the consideration in future work. In addition, these effort measures are based on the assumption that developers inspect defective lines independently. However, as confirmed by our survey, developers often have various code inspection practices (e.g., a top-down order or a random order). Thus, future work may conduct an observational study with industrial practitioners to better understand how do practitioners perform code inspection in real-world practices.

Various cross-project defect prediction approaches have been proposed to improve the cross-project defection performance (e.g., data clustering, data normalization). However, such approaches are not designed for predictions at the line level. Thus, we do not perform any data preprocessing for our line-level cross-project defect prediction, since such file-level cross-project defect prediction approaches are specifically designed for software metrics and machine learning approaches, not for embedding vectors and deep learning.

As suggested by Rahman and Rigby [52], we removed special characters. However, it is possible that such special characters may contribute to software defects (e.g., missing parentheses). Nevertheless, the design of our DeepLineDP approach is focused on the semantic properties of source code (i.e., input sequences of source code), not the syntactic properties. Thus, other information like syntactic could be considered in future work to improve the performance.

Threats to external validity concern the generalizability of our work. Our results are limited to the studied 32 software releases from 9 software projects. Thus, our results may not generalize to other software projects.

In addition, DeepLineDP is specifically designed for release-based defect prediction models. Thus, DeepLineDP is not applicable to other types of defect prediction models (e.g., Just-In-Time defect prediction). This is due to the different structure of information. For release-based defect prediction, DeepLineDP is designed to capture the structure of source code (i.e., tokens forming lines, lines forming files). On the other hand, for commit-based defect prediction, the structure of code changes is that tokens forming lines, lines forming changed hunks, hunks forming changed files, changed files forming one commit, which requires a different neural network architecture.

Threats to internal validity concern the impact of the hyper-parameter settings on the performance of our DeepLineDP models. The performance of our models would be affected by the different weights, which are tuned by hand in our experiments. However, optimizing the parameter settings for deep learning approaches is computationally expensive.

6 Related Work

Below, we discussed the related work regarding defect prediction approaches in different aspects.

6.1 Granularity Levels of Defect Prediction Models

In the past decades, prior studies proposed defect prediction approaches at various granularity, e.g., packages [27], components [63], modules [28], files [23], [24], [25], [27], [39], methods [15], and commits [29]). However, developers still waste a large amount of SQA effort on locating actual defective lines. Recently, a survey by Wan *et al.* [68] found that practitioners prefer fine-grained defect prediction. Thus, a line-level defect prediction is needed to help developers prioritize their SQA effort on the fine-grained high-risk areas of source code in a cost-effective.

6.2 Software Features for Defect Prediction

Traditionally, the accuracy of defect prediction models heavily relies on software features. There are numerous static source code attributes [41] that share a strong relationship with software quality,. For example, file size [2], McCabe's Cyclomatic code complexity [5], code smells [45], and test smells [55]. In addition, software development practices may share a strong relationship with software quality. For example, organization structure [43], code ownership [6], [17], [50], the entropy of change [12], [13], and developers' characteristics [10]. However, these software features are specifically designed for file-level defect prediction. Thus, the approaches

that rely on these software features cannot predict defect at line level.

6.3 Deep Learning Approaches for Defect Prediction

Wang *et al.* [71] used a Deep Belief Network (DBN) architecture to represent a source code file using semantic features. Li *et al.* [33], [35] used a Convolutional Neural Network (CNN) architecture to learn the semantic and structural features of source code. Dam *et al.* [9] used a Long Short-Term Memory (LSTM) architecture to learn the semantic and syntactic features of source code. Zou *et al.* [80] and Li *et al.* [36] used a bidirectional Long Short-Term Memory architecture to learn semantic and syntactic features to predict defective files. However, the granularity level of predictions of prior deep learning-based defect prediction models is still coarse-grained (i.e., file level). Different from prior work, we present a fine-grained deep learning-based defect prediction approach that can predict defects at line level.

6.4 Line-Level Defect Prediction

Prior studies attempt to predict defective lines using various approaches [38], [48], [51], [54], [69], [73]. The simplest approach is to apply static analysis tools to identify defective lines based on a set of predefined rules. However, static analysis tools generate too many false positives warnings [26], and they may not be associated with post-release defects. Recently, Majd *et al.* [38] proposed a suite of 32 C/C++-based statement-level features (e.g., the number of binary and unary operators used in a statement) for statement-level defect prediction using a Long Short-Term Memory (LSTM) neural network architecture. However, these statement-level features only capture the static attributes of statements, which still require handcrafted features extracted by data scientists. Ray *et al.* [54] and Wang *et al.* [69] proposed to use an n-gram language model to predict the unnatural code tokens, which later can be used to identify defective lines. While the n-gram language model can capture the surrounding code tokens, its length of surrounding code tokens is limited to n tokens. Recently, studies leveraged a model-agnostic technique (i.e., LIME) from the Explainable AI domain to develop line-level defect prediction for just-in-time defect prediction [48] and release-based defect prediction [73]. Unlike existing techniques, our DeepLineDP approach is the first deep learning approach for line-level defect prediction that automatically extracting features (i.e., semantic properties).

6.5 Explainable AI for Software Engineering

Explainable AI has been actively investigated in the domain of defect prediction [57], [58]. Recently, Jiarpakdee *et al.* [22] found that explaining the predictions are as equally important and useful as improving the accuracy of defect prediction. However, their literature review found that 91% (81/96) of the defect prediction studies only focus on improving the predictive accuracy, without considering explaining the predictions, while only 4% of these 96 studies focus on explaining the predictions.

Although Explainable AI for SE is still very under-researched, recent works have shown some successful case studies to make defect prediction models more practical [48], [73], explainable [21], [30], and actionable [49], [53]. For example, Pornprasit and Tantithamthavorn [48] leveraged LIME for line-level just-in-time defect prediction. Wattanakriengkrai *et al.* [73] leveraged LIME for line-level release-based defect prediction, helping developers better identify the location of software defects. Jiarpakdee *et al.* [21] and Khanan *et al.* [30] employed model-agnostic techniques (e.g., LIME) for explaining defect prediction models, helping developers better understand why a file is predicted as defective. Rajapaksha *et al.* [53] and Pornprasit *et al.* [49] proposed local rule-based model-agnostic techniques to generate actionable guidance to help managers chart the most effective quality improvement plans. However, these studies only focus on explaining the traditional machine learning approaches. Unlike prior studies, our DeepLineDP is a deep learning approach that is designed to be interpretable by using the attention mechanism. We also make use of the attention mechanism to identify the most risky tokens and risky lines.

7 Conclusion

In this paper, we present DeepLineDP, a deep learning approach to automatically learn the semantic properties of the surrounding tokens and lines in order to identify defective files and defective lines. Through a case study of 32 releases of 9 software projects, we find that the risk score of code tokens varies greatly depending on their location. Our DeepLineDP is 14%-24% more accurate than other file-level defect prediction approaches; is 50%-250% more cost-effective than other line-level defect prediction approaches; and achieves a reasonable performance when transferred to other software projects.

References

[1] E. Aftandilian, R. Sauciuc, S. Priya, and S. Krishnan, "Building useful program analysis tools using an extensible Java compiler," in *Proc. IEEE 12th Int. Work. Conf. Source Code Anal. Manipulation*, 2012, pp. 14–23.

[2] F. Akiyama, "An example of software system debugging," in *Proc. Int. Federation Inf. Process. Congr.*, 1971, pp. 353–359.

[3] E. Arisholm, L. C. Briand, and E. B. Johannessen, "A systematic and comprehensive investigation of methods to build and evaluate fault prediction models," *J. Syst. Softw.*, vol. 83, no. 1, pp. 2–17, 2010.

[4] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," 2014, *arXiv:1409.0473*.

[5] V. R. Basili, L. C. Briand, and W. L. Melo, "A validation of object-oriented design metrics as quality indicators," *IEEE Trans. Softw. Eng.*, vol. 22, no. 10, pp. 751–761, Oct. 1996.

[6] C. Bird, N. Nagappan, B. Murphy, H. Gall, and P. Devanbu, "Putting it all together: Using socio-technical networks to predict failures," in *Proc. Int. Symp. Softw. Rel. Eng.*, 2009, pp. 109–119.

[7] G. Bouchard, "Efficient bounds for the softmax function, applications to inference in hybrid models," in *Proc. Presentation Workshop Appro. Bayesian Inference Continuous/Hybrid Syst.*, 2007.

[8] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer, "SMOTE: Synthetic minority over-sampling technique," *J. Artif. Intell. Res.*, vol. 16, pp. 321–357, 2002.

[9] H. K. Dam, T. Tran, T. T. M. Pham, S. W. Ng, J. Grundy, and A. Ghose, "Automatic feature learning for predicting vulnerable software components," *IEEE Trans. Softw. Eng.*, vol. 47, no. 1, pp. 67–85, Jan. 2021.

[10] D. Di Nucci , F. Palomba, G. De Rosa , G. Bavota, R. Oliveto, and A. De Lucia , "A developer centered bug prediction model," *IEEE Trans. Softw. Eng.*, vol. 44, no. 1, pp. 5–24, Jan. 2018.

[11] A. Habib and M. Pradel, "How many of all bugs do we find? A study of static bug detectors," in *Proc. 33rd IEEE/ACM Int. Conf. Automated Softw. Eng.*, 2018, pp. 317–328.
[12] A. E. Hassan, "Predicting faults using the complexity of code changes," in *Proc. 31st Int. Conf. Softw. Eng.*, 2009, pp. 78–88.
[13] A. E. Hassan and R. C. Holt, "The top ten list: Dynamic fault prediction," in *Proc. Int. Conf. Softw. Maintenance*, 2005, pp. 263–272.
[14] H. Hata, O. Mizuno, and T. Kikuno, "Fault-prone module detection using large-scale text features based on spam filtering," *Empirical Softw. Eng.*, vol. 15, no. 2, pp. 147–165, 2010.
[15] H. Hata, O. Mizuno, and T. Kikuno, "Bug prediction based on fine-grained module histories," in *Proc. 34th Int. Conf. Softw. Eng.*, 2012, pp. 200–210.
[16] V. J. Hellendoorn and P. Devanbu, "Are deep neural networks the best choice for modeling source code?," in *Proc. 11th Joint Meeting Found. Softw. Eng.*, 2017, pp. 763–773.
[17] K. Herzig, "Using pre-release test failures to build early post-release defect prediction models," in *Proc. IEEE 25th Int. Symp. Softw. Rel. Eng.*, 2014, pp. 300–311.
[18] T. Hoang, H. K. Dam, Y. Kamei, D. Lo, and N. Ubayashi, "DeepJIT: An end-to-end deep learning framework for just-in-time defect prediction," in *Proc. 16th Int. Conf. Mining Softw. Repositories*, 2019, pp. 34–45.
[19] T. Hoang, H. J. Kang, J. Lawall, and D. Lo, "Cc2Vec: Distributed representations of code changes," in *Proc. Int. Conf. Softw. Eng.*, 2020, pp. 518–529.
[20] Q. Huang, X. Xia, and D. Lo, "Supervised vs unsupervised models: A holistic look at effort-aware just-in-time defect prediction," in *Proc. IEEE Int. Conf. Softw. Maintenance Evol.*, 2017, pp. 159–170.
[21] J. Jiarpakdee, C. Tantithamthavorn, H. K. Dam, and J. Grundy, "An empirical study of model-agnostic techniques for defect prediction models," *IEEE Trans. Softw. Eng.*, vol. 48, no. 1, pp. 166–185, Jan. 2022.
[22] J. Jiarpakdee, C. Tantithamthavorn, and J. Grundy, "Practitioners' perceptions of the goals and visual explanations of defect prediction models," in *Proc. Int. Conf. Mining Softw. Repositories*, 2021, pp. 432–443.
[23] J. Jiarpakdee, C. Tantithamthavorn, and A. E. Hassan, "The impact of correlated metrics on defect models," *IEEE Trans. Softw. Eng.*, vol. 47, no. 2, pp. 320–331, Feb. 2021.
[24] J. Jiarpakdee, C. Tantithamthavorn, and C. Treude, "AutoSpearman: Automatically mitigating correlated software metrics for interpreting defect models," in *Proc. IEEE Int. Conf. Softw. Maintenance Evol.*, 2018, pp. 92–103.
[25] J. Jiarpakdee, C. Tantithamthavorn, and C. Treude, "The impact of automated feature selection techniques on the interpretation of defect models," *Empirical Softw. Eng.*, vol. 25, pp. 3590–3638, 2020.
[26] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge, "Why don't software developers use static analysis tools to find bugs?," in *Proc. 35th Int. Conf. Softw. Eng.*, 2013, pp. 672–681.
[27] Y. Kamei, S. Matsumoto, A. Monden, K.-I. Matsumoto, B. Adams, and A. E. Hassan, "Revisiting common bug prediction findings using effort-aware models," in *Proc. IEEE Int. Conf. Softw. Maintenance*, 2010, pp. 1–10.
[28] Y. Kamei, A. Monden, S. Matsumoto, T. Kakimoto, and K.-I. Matsumoto, "The effects of over and under sampling on fault-prone module detection," in *Proc. 1st Int. Symp. Empirical Softw. Eng. Meas.*, 2007, pp. 196–204.
[29] Y. Kamei *et al.*, "A large-scale empirical study of just-in-time quality assurance," *IEEE Trans. Softw. Eng.*, vol. 39, no. 6, pp. 757–773, Jun. 2013.
[30] C. Khanan *et al.*, "JITBot: An explainable just-in-time defect prediction BOT," in *Proc. 35th IEEE/ACM Int. Conf. Automated Softw. Eng.*, 2020, pp. 1336–1339.
[31] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," 2014, *arXiv:1412.6980*.
[32] B. A. Kitchenham and S. L. Pfleeger, "Personal opinion surveys," in *Guide To Advanced Empirical Software Engineering*, London, U.K.: Springer, 2008, pp. 63–92.
[33] J. Li, P. He, J. Zhu, and M. R. Lyu, "Software defect prediction via convolutional neural network," in *Proc. IEEE Int. Conf. Softw. Qual., Rel. Secur.*, 2017, pp. 318–328.
[34] J. Li *et al.*, "Feature selection: A data perspective," *ACM Comput. Surv.*, vol. 50, no. 6, pp. 1–45, 2017.
[35] Y. Li, S. Wang, T. N. Nguyen, and S. Van Nguyen , "Improving bug detection via context-based code representation learning and attention-based neural networks," *Proc. ACM Programm. Lang.*, vol. 3, pp. 1–30, 2019.
[36] Z. Li *et al.*, "VulDeePecker: A deep learning-based system for vulnerability detection," 2018, *arXiv:1801.01681*.
[37] G. Lin *et al.*, "Cross-project transfer representation learning for vulnerable function discovery," *IEEE Trans. Ind. Inform.*, vol. 14, no. 7, pp. 3289–3297, Jul. 2018.
[38] A. Majd, M. Vahidi-Asl, A. Khalilian, P. Poorsarvi-Tehrani, and H. Haghighi, "SLDeep: Statement-level software defect prediction using deep-learning model on static code features," *Expert Syst. Appl.*, vol. 147, 2020, Art. no. 113156.
[39] T. Mende and R. Koschke, "Effort-aware defect prediction models," in *Proc. 14th Eur. Conf. Softw. Maintenance Reeng.*, 2010, pp. 107–116.
[40] T. Menzies, A. Butcher, A. Marcus, T. Zimmermann, and D. Cok, "Local vs. global models for effort estimation and defect prediction," in *Proc. 26th IEEE/ACM Int. Conf. Automated Softw. Eng.*, 2011, pp. 343–351.
[41] T. Menzies, J. Greenwald, and A. Frank, "Data mining static code attributes to learn defect predictors," *IEEE Trans. Softw. Eng.*, vol. 33, no. 1, pp. 2–13, Jan. 2007.
[42] H. Mi, Z. Wang, and A. Ittycheriah, "Vocabulary manipulation for neural machine translation," 2016, *arXiv:1605.03209*.
[43] N. Nagappan and T. Ball, "Use of relative code churn measures to predict system defect density," in *Proc. Int. Conf. Softw. Eng.*, 2005, pp. 284–292.
[44] N. Nagappan, T. Ball, and A. Zeller, "Mining metrics to predict component failures," in *Proc. Int. Conf. Softw. Eng.*, 2006, pp. 452–461.
[45] F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, D. Poshyvanyk, and A. De Lucia , "Mining version histories for detecting code smells," *IEEE Trans. Softw. Eng.*, vol. 41, no. 5, pp. 462–489, May 2015.
[46] C. Parnin and A. Orso, "Are automated debugging techniques actually helping programmers?," in *Proc. Int. Symp. Softw. Testing Anal.*, 2011, pp. 199–209.
[47] L. Pascarella, F. Palomba, and A. Bacchelli, "Fine-grained just-in-time defect prediction," *J. Syst. Softw.*, vol. 150, pp. 22–36, 2019.
[48] C. Pornprasit and C. Tantithamthavorn, "JITLine: A simpler, better, faster, finer-grained just-in-time defect prediction," in *Proc. Int. Conf. Mining Softw. Repositories*, 2021.
[49] C. Pornprasit, C. Tantithamthavorn, J. Jiarpakdee, M. Fu, and P. Thongtanunam, "PyExplainer: Explaining the predictions of just-in-time defect models," in *Proc. 36th IEEE/ACM Int. Conf. Automated Softw. Eng.*, 2021, pp. 407–418.
[50] F. Rahman and P. Devanbu, "Ownership, experience and defects: A fine-grained study of authorship," in *Proc. Int. Conf. Softw. Eng.*, 2011, pp. 491–500.
[51] F. Rahman, S. Khatri, E. T. Barr, and P. Devanbu, "Comparing static bug finders and statistical prediction," in *Proc. 36th Int. Conf. Softw. Eng.*, 2014, pp. 424–434.
[52] M. Rahman, D. Palani, and P. C. Rigby, "Natural software revisited," in *Proc. 41st Int. Conf. Softw. Eng.*, 2019, pp. 37–48.
[53] D. Rajapaksha, C. Tantithamthavorn, C. Bergmeir, W. Buntine, J. Jiarpakdee, and J. Grundy, "SQAPlanner: Generating data-informed software quality improvement plans," *IEEE Trans. Softw. Eng.*, to be published, doi: 10.1109/TSE.2021.3070559.
[54] B. Ray, V. Hellendoorn, S. Godhane, Z. Tu, A. Bacchelli, and P. Devanbu, "On the naturalness of buggy code," in *Proc. 38th Int. Conf. Softw. Eng.*, 2016, pp. 428–439.
[55] D. Spadini, F. Palomba, A. Zaidman, M. Bruntink, and A. Bacchelli, "On the relation of test smells to software code quality," in *Proc. IEEE Int. Conf. Softw. Maintenance Evol.*, 2018, pp. 1–12.
[56] C. Tantithamthavorn, A. E. Hassan, and K. Matsumoto, "The impact of class rebalancing techniques on the performance and interpretation of defect prediction models," *IEEE Trans. Softw. Eng.*, vol. 46, no. 11, pp. 1200–1219, Nov. 2020.
[57] C. Tantithamthavorn, J. Jiarpakdee, and J. Grundy, "Explainable AI for software engineering," 2020, *arXiv:2012.01614*.
[58] C. Tantithamthavorn, J. Jiarpakdee, and J. Grundy, "Actionable analytics: Stop telling me what it is: Please tell me what to do," *IEEE Softw.*, vol. 38, no. 4, pp. 115–120, Jul./Aug. 2021.
[59] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, and K. Matsumoto, "Automated parameter optimization of classification techniques for defect prediction models," in *Proc. IEEE/ACM 38th Int. Conf. Softw. Eng.*, 2016, pp. 321–332.

[60] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, and K. Matsumoto, "An empirical comparison of model validation techniques for defect prediction models," *IEEE Trans. Softw. Eng.*, vol. 43, no. 1, pp. 1–18, Jan. 2017.
[61] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, and K. Matsumoto, "The impact of automated parameter optimization on defect prediction models," *IEEE Trans. Softw. Eng.*, vol. 45, no. 7, pp. 683–711, Jul. 2019.
[62] P. Thongtanunam, S. McIntosh, A. E. Hassan, and H. Iida, "Investigating code review practices in defective files: An empirical study of the QT system," in *Proc. 12th Work. Conf. Mining Softw. Repositories*, 2015, pp. 168–179.
[63] P. Thongtanunam, S. McIntosh, A. E. Hassan, and H. Iida, "Revisiting code ownership and its relationship with software quality in the scope of modern code review," in *Proc. IEEE/ACM 38th Int. Conf. Softw. Eng.*, 2016, pp. 1039–1050.
[64] M. Tufano, J. Pantiuchina, C. Watson, G. Bavota, and D. Poshyvanyk, "On learning meaningful code changes via neural machine translation," in *Proc. 41st Int. Conf. Softw. Eng.*, 2019, pp. 25–36.
[65] B. Turhan, T. Menzies, A. B. Bener, and J. Di Stefano , "On the relative value of cross-company and within-company data for defect prediction," *Empirical Softw. Eng.*, vol. 14, pp. 540–578, 2009.
[66] C. Vassallo, F. Palomba, A. Bacchelli, and H. C. Gall, "Continuous code quality: Are we (really) doing that?," in *Proc. 33rd ACM/IEEE Int. Conf. Automated Softw. Eng.*, 2018, pp. 790–795.
[67] A. Vaswani *et al.*, "Attention is all you need," in *Proc. 31st Int. Adv. Neural Inf. Process. Syst.*, 2017, pp. 6000–6010.
[68] Z. Wan, X. Xia, A. E. Hassan, D. Lo, J. Yin, and X. Yang, "Perceptions, expectations, and challenges in defect prediction," *IEEE Trans. Softw. Eng.*, vol. 46, no. 11, pp. 1241–1266, Nov. 2020.
[69] S. Wang, D. Chollak, D. Movshovitz-Attias, and L. Tan, "Bugram: Bug detection with n-gram language models," in *Proc. 31st IEEE/ACM Int. Conf. Automated Softw. Eng.*, 2016, pp. 708–719.
[70] S. Wang, T. Liu, J. Nam, and L. Tan, "Deep semantic feature learning for software defect prediction," *IEEE Trans. Softw. Eng.*, vol. 46, no. 12, pp. 1267–1293, Dec. 2020.
[71] S. Wang, T. Liu, and L. Tan, "Automatically learning semantic features for defect prediction," in *Proc. 38th Int. Conf. Softw. Eng.*, 2016, pp. 297–308.
[72] C. Watson, N. Cooper, D. N. Palacio, K. Moran, and D. Poshyvanyk, "A systematic literature review on the use of deep learning in software engineering research," 2020, *arXiv:2009.06520*.
[73] S. Wattanakriengkrai, P. Thongtanunam, C. Tantithamthavorn, H. Hata, and K. Matsumoto, "Predicting defective lines using a model-agnostic technique," 2020, *arXiv:2009.03612*.
[74] M. White, C. Vendome, M. Linares-Vásquez, and D. Poshyvanyk, "Toward deep learning software repositories," in *Proc. IEEE/ACM 12th Work. Conf. Mining Softw. Repositories*, 2015, pp. 334–345.
[75] X. Xia, D. Lo, S. J. Pan, N. Nagappan, and X. Wang, "HYDRA: Massively compositional model for cross-project defect prediction," *IEEE Trans. Softw. Eng.*, vol. 42, no. 10, pp. 977–998, Oct. 2016.
[76] Z. Yang, D. Yang, C. Dyer, X. He, A. Smola, and E. Hovy, "Hierarchical attention networks for document classification," in *Proc. Conf. North Amer. Chapter Assoc. Comput. Linguistics: Hum. Lang. Technol.*, 2016, pp. 1480–1489.
[77] S. Yathish, J. Jiarpakdee, P. Thongtanunam, and C. Tantithamthavorn, "Mining software defects: Should we consider affected releases?," in *Proc. IEEE/ACM 41st Int. Conf. Softw. Eng.*, 2019, pp. 654–665.
[78] F. Zhang, Q. Zheng, Y. Zou, and A. E. Hassan, "Cross-project defect prediction using a connectivity-based unsupervised classifier," in *Proc. IEEE/ACM 38th Int. Conf. Softw. Eng.*, 2016, pp. 309–320.
[79] T. Zimmermann, N. Nagappan, H. Gall, E. Giger, and B. Murphy, "Cross-project defect prediction: A large scale experiment on data vs. domain vs. process," in *Proc. 7th Joint Meeting Eur. Softw. Eng. Conf.*, 2009, pp. 91–100.
[80] D. Zou, S. Wang, S. Xu, Z. Li, and H. Jin, "VulDeePecker: A deep learning-based system for multiclass vulnerability detection," *IEEE Trans. Dependable Secure Comput.*, vol. 18, no. 5, pp. 2224–2236, Sep./Oct. 2021.

Chanathip Pornprasit (Student Member, IEEE) is currently working toward the PhD degree with Monash University, Australia. His PhD goal is applying the knowledge of machine learning and software quality assurance to assist practitioners improving software quality in software development process. His research interests include explainable AI for defect prediction models and machine learning for automated software quality assurance activity.

Chakkrit (Kla) Tantithamthavorn (Member, IEEE) is currently a senior research fellow with the Faculty of Information Technology, Monash University, Australia. From 2013 to 2020, he was recognized as the most impactful early-career SE researcher based on a bibliometric assessment of software engineering. He was the recipient of the numerous prestigious awards including the ACM SIGSOFT Distinguished Paper Award at ASE'21 and the 2020 ARC's Discovery Early Career Researcher Award (DECRA).