Expert Systems With Applications 147 (2020) 113156

Contents lists available at ScienceDirect

Expert Systems With Applications

journal homepage: www.elsevier.com/locate/eswa

SLDeep: Statement-level software defect prediction using deep-learning model on static code features

Amirabbas Majd[a], Mojtaba Vahidi-Asl[a,*], Alireza Khalilian[b], Pooria Poorsarvi-Tehrani[a], Hassan Haghighi[a]

[a] *Faculty of Computer Science and Engineering, Shahid Beheshti University G. C., Tehran, Iran*
[b] *Department of Software Engineering, Faculty of Computer Engineering, University of Isfahan, Isfahan, Iran*

A R T I C L E I N F O

Article history:
Received 4 July 2019
Revised 23 November 2019
Accepted 20 December 2019
Available online 23 December 2019

Keywords:
Defect
Software fault proneness
Machine learning
Fault prediction model
Software metric

A B S T R A C T

Software defect prediction (SDP) seeks to estimate fault-prone areas of the code to focus testing activities on more suspicious portions. Consequently, high-quality software is released with less time and effort. The current SDP techniques however work at coarse-grained units, such as a module or a class, putting some burden on the developers to locate the fault. To address this issue, we propose a new technique called as Statement-Level software defect prediction using Deep-learning model (SLDeep). The significance of SLDeep for intelligent and expert systems is that it demonstrates a novel use of deep-learning models to the solution of a practical problem faced by software developers. To reify our proposal, we defined a suite of 32 statement-level metrics, such as the number of binary and unary operators used in a statement. Then, we applied as learning model, long short-term memory (LSTM). We conducted experiments using 119,989 C/C++ programs within Code4Bench. The programs comprise 2,356,458 lines of code of which 292,064 lines are faulty. The benchmark comprises a diverse set of programs and versions, written by thousands of developers. Therefore, it tends to give a model that can be used for cross-project SDP. In the experiments, our trained model could successfully classify the unseen data (that is, fault-proneness of new statements) with average performance measures 0.979, 0.570, and 0.702 in terms of recall, precision, and accuracy, respectively. These experimental results suggest that SLDeep is effective for statement-level SDP. The impact of this work is twofold. Working at statement-level further alleviates developer's burden in pinpointing the fault locations. Second, cross-project feature of SLDeep helps defect prediction research become more industrially-viable.

1. Introduction

Today, humans are increasingly relying on software for every task. Thus, the current software has become more complex and costly to develop. Meanwhile, we are still in trouble with low-quality software; it has imposed large economic costs (Bird et al., 2009). To improve software quality and alleviate the underlying costs, an approach has been to construct effective software defect prediction (SDP) techniques. By using fault historical data along with static code features gathered from previous software versions, SDP techniques seek to develop prediction models. These models are intended to shed light on problematic portions of the code in the next release of software. This is achieved by predicting (estimating) fault-prone code fragments. Once fault-prone portions are identified, testing and code review activities can focus on them. The SDP research is therefore of great practical importance as it makes software quality assurance activities more targeted and productive. As a result, the high-quality software can be obtained (Hall, Beecham, Bowes, Gray & Counsell, 2011).

To date, we are witnessing the advent of a proliferation of SDP techniques. They have used different metrics such as code complexity (Radjenović, Heričko, Torkar & Živkovič, 2013), code micro-interactions (Lee, Nam, Han, Kim & In, 2016), and smelly statements (Palomba, Zanoni, Fontana, De Lucia & Oliveto, 2017). In addition, they have been constructed based on different learning models (Malhorta, 2015; Bowes, Hall and Petrić, 2018; Rathore, 2017) such as decision trees, support vector machines and Naïve Bayes. The current techniques are either used to predict within-project (Hall et al., 2011) or cross-project (Hosseini, Turhan & Gunarathna, 2017) software defects. Often, SDP is used at code level prior to software release (Catal, 2011). However, SDP techniques exist (Özakıncı, 2018) that can be used in early stages of

* Corresponding author.
E-mail addresses: a.majd@mail.sbu.ac.ir (A. Majd), mo_vahidi@sbu.ac.ir (M. Vahidi-Asl), khalilian@eng.ui.ac.ir (A. Khalilian), h_haghighi@sbu.ac.ir (H. Haghighi).

https://doi.org/10.1016/j.eswa.2019.113156
0957-4174/© 2020 Elsevier Ltd. All rights reserved.

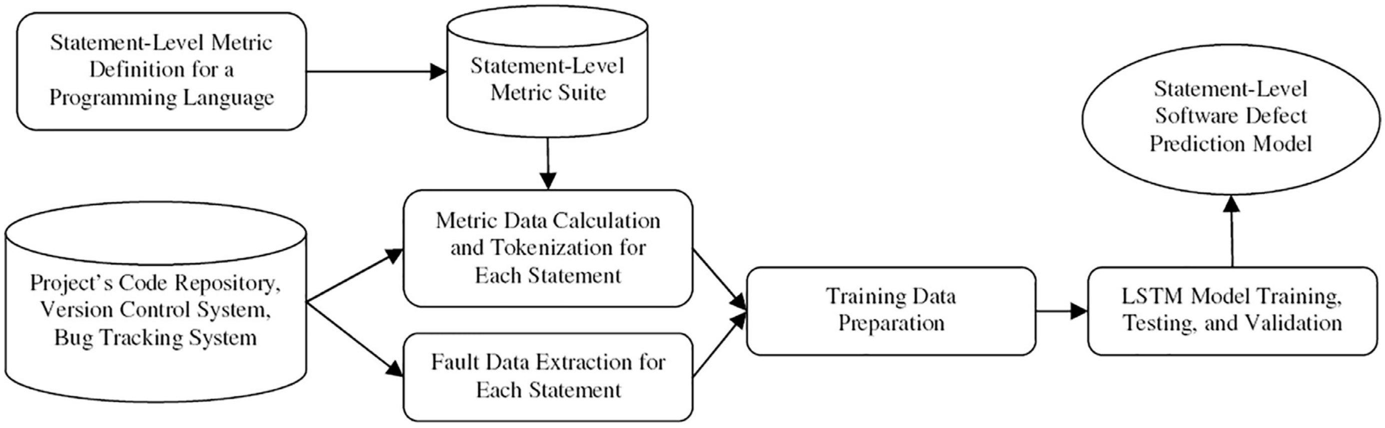

Statement-Level Metric Definition for a Programming Language
Statement-Level Metric Suite
Statement-Level Software Defect Prediction Model
Metric Data Calculation and Tokenization for Each Statement
Project's Code Repository, Version Control System, Bug Tracking System
Training Data Preparation
LSTM Model Training, Testing, and Validation
Fault Data Extraction for Each Statement

Fig. 1. The high-level overview of the SLDeep architecture.

software development such as requirements engineering, analysis, and design. Finally, the current techniques are developed to work at coarse-grained code areas such as a class, module, file, function, or plug-in binaries (Hall et al., 2011; Hosseini et al., 2017). We observe that SDP techniques that work at finer granularities such as statements have received little or no attention.

The wealth of current SDP techniques has profoundly helped for improving software quality. In fact, the research community acknowledges the benefits of SDP in vitro; however, limited real-world applications have been a major criticism to SDP research (Lanza, Mocci & Ponzanelli, 2016). One reason seems to lie in the fact that with the current techniques, which rely on defect prediction at higher granularities, developers have to spend non-trivial time to detect and localize the faults within a module labeled as fault-prone. Fine-grained defect prediction analysis can result in acceleration of identifying fault locations (Kamei et al., 2016). Our hypothesis is that statement-level SDP has much potential to pinpoint fault-prone locations. Consequently, faults would be detected with less time and effort.

To reify our hypothesis, we propose a new technique, which we call Statement-Level defect prediction using Deep-learning models (SLDeep). Our goal is to construct a prediction model that can be used to classify a program statement as fault-prone or fault-free. Fig. 1 depicts the high-level architecture of SLDeep. It works on source code defect prediction at the statement level. Also, it is designed neither for usage at early stages of the software development lifecycle nor for online defect prediction. For statement-level SDP, we first introduce a suite of 32 new code metrics that measure some features of a statement. An example of such metrics is the number of binary operators used in a line of code. The metric suite is intended to extract static information out of a statement that can be used to effectively distinguish it with another statement. The metrics are also intended to estimate how much a statement is likely to be faulty. As its learning component, SLDeep applies long short-term memory (LSTM) (Goodfellow, Bengio & Courville, 2016) because it lends itself to the nature of our data. LSTM is an effective deep-learning model in widespread usage (Sutskever, Vinyals and Le, 2014). The choice of LSTM is in contrast to the existing literature, in which other models such as logistic regression or Naïve Bayes have been successfully applied (Radjenović et al., 2013). The volume and nature of our data necessitate a model with both high scalability and memorization features, respectively. Model scalability is essentially needed as we produce very large amounts of data due to our fine-grained analysis. Memorization is also needed to help the model learn code structure as a sequence of statements and make more informative decisions based on previously-observed statements. We decided to preserve every extracted metric per each statement. This design decision exempts us from any feature selection preprocessing and is a step towards a fully-automated technique. We give more arguments on this decision in Section 5.

SLDeep can be used for both within-project and cross-project defect prediction.[1] This capability offers practical utility because the lack of fault data or difficulty in their collection have been among the reasons that hamper companies to apply within-project SDP techniques in practice (Turhan, Menzies, Bener and Di Stefano, 2009). Cross-project defect prediction can be a viable case for companies as it significantly reduces the effort in data collection. In addition, the existing within-project data might no longer be indicative of the current practices (Kitchenham, Mendes & Travassos, 2007), since practices change over time. As yet, a large number of real-world projects have made their full source code and fault data available at public repositories. In presence of such data, cross-project defect prediction can provide significant potential in vivo Zimmermann, Nagappan, Gall, Giger & Murphy, 2009).

To evaluate SLDeep, we have conducted experiments on more than 100,000 C/C++ programs within Code4Bench (Majd, Vahidi-Asl, Khalilian, Baraani-Dastjerdi & Zamani, 2019). This benchmark comprises all versions of each program. For each statement of each subject program, we measured our introduced metrics. Then, we tokenized each statement to its lexemes. For fault data, we used the corresponding ready-made information, available in Code4Bench. Next, we made a matrix per program, in which each row is devoted to a statement. For each statement, the corresponding columns are used for metric data, the including lexemes, the fault data, and a label indicating fault-prone or fault-free. Finally, we trained an LSTM model using the matrix of all programs.

The programs we used in our experiments are developed by thousands of developers for thousands of problems. Hence, the learned model is not unique to a single project. It has been trained using a diverse set of data along four dimensions: programs (projects, problems), versions, statements, and developers. This property is a step towards a universal SDP model for a particular programming language. This viewpoint provides a novel formulation for the SDP problem.

In our experiments, we considered an accuracy neighborhood. That is, for a fault-free statement labeled as fault-prone by the model, we take the prediction as true if the really faulty statement is at most n lines before or after that statement. For our accuracy neighborhood set to four, in the experiments, we could train a model with average performance measures 0.988, 0.581, and 0.715 in terms of recall, precision, and accuracy, respectively. For the un-

[1] See Section 2 for concepts and definitions.

seen data at the test stage,[2] our trained model successfully classified them with average performance measures 0.979, 0.570, and 0.702 in terms of recall, precision, and accuracy, respectively. We have also conducted an experiment in which we used Random-Forest model instead of LSTM as the leaner component. For accuracy neighborhood set to four, the Random-Forest model could classify the unseen data with average performance measures 0.182, 0.580, and 0.627 for recall, precision, and accuracy, respectively. In the context of SDP, recall is the key measure because it shows the percent of really faulty statements that are labeled as faulty by the model. A high recall implies that the model could effectively estimate the fault-prone statements of the code, which is in line with ultimate goal of SDP research. Therefore, based on our experimental results, we concluded that SLDeep with LSTM is effective for statement-level SDP and can be successfully adopted.

We provided every methodological, contextual, and performance information regarding our method and experiments, according to 14 criteria suggested by Hall et al. (Hall et al., 2011). This information makes the usage of SLDeep clear and straightforward for researchers and practitioners. The information also allows other researchers to qualitatively and quantitatively synthesize results in future systematic studies. We made our codes publicly available at: https://github.com/sldeep/SLDeep. In addition, we made our training data publicly available at http://doi.org/10.5281/zenodo.3268512, with DOI 10.5281/zenodo.3268512.

For researchers, SLDeep can open a new avenue to leverage and optimize other deep-learning models. It can also help to define additional fine-grained statement-level code metrics for other programming languages. For practitioners, SLDeep can assist to spend less time in finding faulty statements. As a result, more faults can be found and fixed. Therefore, we can release higher quality software in presence of limited resources.

To sum up, the main contributions of this paper are as follows:

- A survey of the literature, highlighting gaps, strengths, weaknesses, and lessons-learned
- A new formulation for software defect prediction
- The introduction of 32 novel source code metrics at statement-level
- SLDeep, a new LSTM-based system for statement-level software defect prediction
- Empirical studies of applying SLDeep on more than 100,000 C/C++ programs in Code4Bench
- Empirical studies of applying Random Forest on the subject programs and comparison with SLDeep
- A discussion on the considerations, implications, limitations, and threats to the validity

The remainder of this paper is organized as follows: In Section 2, we present the required background and survey the related work. We give the details of the methodology of our technique in Section 3. Empirical studies and experimental results are elaborated in Section 4. We provide a discussion on different aspects of SLDeep and the results in Section 5. Finally, we conclude the paper in Section 6.

2. Background and related work

In this section, we briefly explain the fundamental concepts, which are required for the rest of the paper. Then, we present a review of the main body of the SDP literature. We conclude this section with lessons-learned and the specific gaps that we address.

2.1. Foundations

Software defect (fault) prediction techniques leverage software metrics along with fault data to construct a predictive model (Catal, 2011). The fault data may come from the previous versions of the current software project or they may be taken from other similar software projects (Malhorta, 2015). The resulting model is used to predict the fault-prone portions of the unseen software, for example the next release of the same software (Hosseini et al., 2017). When the training data of the prediction model are taken from the same project, we would achieve a within-project defect prediction (WPDP) system. In contrast, when the whole or majority of the training data are taken from other projects, we would achieve a cross-project defect prediction (CPDP) system (Hosseini et al., 2017). The presence of indicative software data and an appropriate learning model significantly influence the efficacy of SDP techniques.

In general, the aim of SDP techniques is an optimized usage of resources in the software team and to deliver high-quality software. In particular, SDP techniques seek to identify faulty areas of the code, to estimate the number of remaining defects, and to track and localize the faulty modifications to the code.

Three components are often required to construct a defect prediction system (Catal, 2009). First, a version control system such as Subversion[3] is required to store source code data. Second, a change management system such as Bugzilla[4] is required to capture the fault-related data. Finally, a helper tool is needed to gather software metrics out of the version control system. For prediction, software metrics are used as independent variables and fault data are used as dependent variables. The prediction models are used to identify fault-prone modules often before system testing. Menzies, Greenwald and Frank (2006) showed that with a robust model, the likelihood of fault detection can be 71%, which is higher than the likelihood of fault detection with code reviews, which is 60%. Catal and Diri (Catal, 2009) enumerate several advantages in using the SDP techniques:

- Improving testing process by focusing on fault-prone code areas
- Achieving high-quality software through better testing
- Identifying refactoring candidates labeled as fault-prone
- Reduction in the underlying costs
- Achieving more robust software

Next, the question arises as to the advances and the current challenges in SDP techniques. We will explore this question in the next subsection.

2.2. Literature review

As of today, a myriad of SDP techniques has appeared, presenting novel methods and rich empirical studies. An important body of literature can be found in the existing systematic studies, which comprise only those studies that have passed certain quality assessment criteria. In this section, first we review several systematic studies to highlight major advances and challenges they have reported. Then, we review a number of individual studies to obtain additional lessons and to help to place our own work in the current body of knowledge. We note that the list of research we investigate here is not comprehensive. Therefore, any missing work should not imply to be undervalued.

2.2.1. Defect prediction in general

There exist a number of systematic studies (Hall et al., 2011; Hosseini et al., 2017; Malhorta, 2015; Radjenović et al., 2013) that

[2] We mean predicting the fault-proneness of new statements.

[3] https://subversion.apache.org/.

[4] https://www.bugzilla.org/.

review defect prediction techniques from different aspects such as learning model and code metrics.

Kamei et al. (2016) conducted an empirical study, in which they investigated the performance of just-in-time models[5] in the context of cross-project SDP. They used 11 open-source projects in their study. They found that the constructed prediction models are more effective when used with process metrics because they are built at finer granularities. They have also found that just-in-time models tend to perform better in cross-project context when: the models are trained on similar projects, the data of several projects are combined and used, and ensembles are developed from several models built on several projects.

The ensemble of different learning models in the context of SDP was explored by Rathore and Kumar (Rathore, 2017). They evaluated the resulting model on several datasets and found that the resulting ensemble led to higher accuracy.

In a study on five common classification models, Gao et al. (Gao, Khoshgoftaar, Wang & Seliya, 2011) investigated the effect of feature selection techniques in the context of the SDP problem. They conducted the experiments on a large software system and reported several key findings. Particularly, they observed that for the code metrics and models they used the reduction in the number of code metrics up to 85% did not adversely impact on the effectiveness of prediction models. This is of paramount importance for practitioners who want to collect as few metrics as possible.

Multi-objective formulation of the SDP problem is the key contribution made by Canfora et al. (2015). In single-objective formulation, the aim is to maximize precision and recall of the model. However, this goal can be insufficient for an effective model. In fact, for a developer who is responsible to test fault-prone classes, the required time and effort plays a major role; the larger class would need more time. As a result, the researchers suggested measuring the goodness of a model based on the time it incurs on the developer to pinpoint the fault. Then, this knowledge is incorporated into multi-objective SDP.

Lee et al. (2016) proposed micro-interaction metrics for the SDP context. These metrics capture interaction information of developers such as file editing and browsing events. In the experiments, they could construct models with higher accuracy and cost-effectiveness. The new metrics also provide novel insights for the developers regarding their ineffective actions.

The experiments by Bowes et al. (Bowes, 2016) sought to understand the nature of faults that can be detected by each prediction model. They used four classifiers on NASA dataset and found that a different set of defects can be detected by each model. Boucher and Badri (Boucher, 2018) conducted an empirical study to find appropriate techniques for threshold computation in the SDP context. They found that receiver operating characteristic (ROC) outperforms other techniques for threshold calculation. Since ROC is a supervised technique, it can be used in situations where fault data are available. Based on the existing findings on fault-proneness, Palomba et al. (2017) developed specific SDP models for smelly[6] classes. They considered measuring the smelly intensity by a new code metric and constructed a model based on process metrics in addition to the new metric. In the experiments, researchers achieved higher accuracy using the resulting model. They also found that the new code metric is relevant and could reduce the model entropy.

Choudhary et al. (Choudhary, Kumar, Kumar, Mishra & Catal, 2018) conducted experiments to investigate the change metrics together with code metrics to enhance the precision of SDP models. The researchers used different versions of Eclipse projects as experimental subjects. In addition to utilizing the existing change metrics, they introduced several new ones. They observed that SDP models with new change metrics outperform SDP models with the existing metrics. In addition, change metrics provide advantages in construction of high-performance SDP models.

To improve the precision of SDP models, Shippey et al. (Shippey, Bowes & Hall, 2019) considered extracting features of faulty Java code. To achieve this, they used a bottom-up analysis on abstract syntax tree (AST) n-grams. They leveraged non-parametric testing to identify the association between the defects of software and AST n-grams. They used open-source and commercial software systems as their subjects. Finally, they built SDP models on three machine-learning classifiers, namely J48, Random Forest, and Naïve Bayes.

2.2.2. *Deep learning for defect prediction*

Yang et al. (Yang, Lo, Xia, Zhang & Sun, 2015) used deep-learning models for the prediction of defect-prone changes. Particularly, they used a deep belief network to extract a set of metrics (features) for code change measurement. The metrics were extracted from fourteen initial features related to change measurement. Then, they trained a classifier on the extracted metrics to predict the defect-prone code changes. They evaluated the approach on six large open-source projects.

Qiao and Wang (Qiao, 2019) proposed to apply deep learning for effort-aware just-in-time SDP. The researchers used ten metrics that measure code changes from various aspects such as the size of the changed code and developers' experience. Then, they trained a deep-learning model to predict the number of potential defects. They evaluated the proposed approach on six open-source projects. As the researchers mentioned, deep learning fits to SDP context because it is capable of selecting effective input features particularly when the relationship between the inputs and outputs are complex.

Fan et al. (Fan, Diao, Yu, Yang & Chen, 2019) applied deep-learning on the vector-encoded structure of the programs for the SDP problem. Particularly, they used attention-based recurrent neural network as deep-learning model. To extract the code structure, they built the AST of a program and mapped it to a numerical vector. Their model also applies the attention mechanism to generate effective features for SDP. They evaluated the proposed approach on seven open-source Java projects.

Manjula and Florence (Manjula, 2018) addressed the SDP problem using a combination of genetic programming and deep learning. Particularly, they applied genetic programming to optimize features and employed deep learning for classification. They have leveraged improved versions of both genetic programming and deep learning customized to their application. The researchers evaluated their hybrid approach on five datasets of the PROMISE repository of software defects.

Pan et al. (Pan, Lu, Xu & Gao, 2019) applied an improved version of convolutional neural network (CNN) for within-project SDP. The researchers built a dataset of source codes, namely Simplified PROMISE to promote datasets in CNN-related studies. Then, they evaluated their approach on different versions of 12 projects within the generated dataset. Finally, they compared the approach with baseline deep-learning based SDP approaches.

Dong et al. (Dong, Wang, Li, Xu & Zhang, 2018) focused on addressing the SDP problem for Android APKs. They proposed a technique to generate customized metrics (features) to capture the properties of decompiled APK files. Their technique extracts both syntactical and semantic metrics of the decompiled APK files. Then, they applied deep-learning techniques to train an SDP model. Finally, the researchers evaluated their approach on more than 90,000 files that are decompiled from 50 Android APKs.

[5] They are models that are applied to identify fault-introducing changes.

[6] Code smells are the sign of poor design and design decisions.

Dam et al. (Dam et al., 2018) proposed to build SDP models using tree-structured LSTM network. As they mentioned, LSTM fits best to the structure of AST, which are an appropriate intermediate representation of the source code. The researchers evaluated their approach on two datasets; one is an open-source project and the other is PROMISE repository. The experiments showed the effectiveness of the proposed approach both for within-project as well as cross-project SDP.

2.2.3. Statement-Level software code metrics

There exists a plethora of software code metrics that have been used in different software quality assurance studies, in general, and in software defect prediction studies, in particular. The metrics are categorized into various groups, such as object-oriented, function-level, and statement-level. From the generality point of view, each group of the existing source code metrics can be divided into two categories, namely general-purpose and special-purpose. The former category includes software code metrics that are commonly used in studies. The latter category includes software code metrics that are defined for a certain study; they are often customized for the specific context in which they are applied.

Fenton and Bieman (Fenton, 2014) provide a comprehensive introduction to software metrics including statement-level ones. Radjenović et al. (Radjenović et al., 2013) elaborate on different software code metrics including statement-level ones in the context of software defect prediction. Moreover, Nuñez-Varela et al. (Nuñez-Varela, Pérez-Gonzalez, Martínez-Perez & Soubervielle-Montalvo, 2017) investigate source code metrics used in different fields of software engineering.

2.3. Conclusions

A number of lessons can be drawn from the systematic studies. We learn that more robust and richer methodologies are required to build prediction models (Hall et al., 2011). In addition, we need to consider defining different code metrics that are more relevant for real-world, industrial applications (Radjenović et al., 2013). Machine-learning models largely contribute to the success of SDP techniques. However, they have been applied in a limited manner and much more potential remain untouched (Malhorta, 2015). Finally, we learn that cross-project SDP is significantly promising. It has merits that can be effectively adopted by companies. However, several important challenges should be addressed, such as the capability to deal with heterogeneous source code data (Hosseini et al., 2017).

From other studies, additional lessons can be drawn. First, we learn that fine-grained code metrics increase the success rate of SDP techniques. In addition, cross-project SDP can significantly benefit from datasets that include several other projects, particularly those that are similar to the project at hand (Kamei et al., 2016). We learn that using more automated mechanisms in model construction would lead to elimination of developer's burden. In this case it is highly likely that SDP techniques are adopted in industry. An implication we can draw from multi-objective SDP is that better prediction models are those that minimize the time to find the faulty statement (Canfora et al., 2015). We also learn that to meet specific design goals, one may need to define new metrics. The existing code metrics may lose effectiveness in certain new circumstances (Lee et al., 2016). Furthermore, we can learn than relevant metrics can reduce the entropy of the prediction models. Ensemble models tend to improve the overall performance of the prediction system (Palomba et al., 2017).

Based on the identified challenges, the high-level goals of this paper are twofold. First, it aims at further alleviating the developer's burden in fault detection and localization. Second, it aims at making SDP more industrially-viable. On the grounding of the lessons-learned, we consider working at finer granularity, particularly statement-level. This implies that we need to define new relevant metrics. We should also leverage better model that can handle our fine-grained data. We accomplish this requirement through exploiting the potentials of deep-learning models, particularly LSTM. To reach industrial scale and adoption, we consider developing cross-project models through as much an automated process as possible. The concrete details of the methodology in the construction and integration of the mentioned components are articulated in the next section.

3. Methodology

To build an effective SDP system, we propose to work at statement-level. For this purpose, we should define relevant code metrics to estimate fault-proneness of each statement. We present the list of the introduced metrics in Section 3.1. The training data we produce are different from the typical data produced in the literature, in terms of the nature and size. Hence, we should apply a relevant learning model. We propose to use LSTM. The details of applying this model are elaborated in Section 3.2. Our new SDP technique, namely SLDeep, is developed by integrating the statement-level metrics with learning model. We describe the resulting SLDeep in Section 3.3.

3.1. Metric suite

Metrics in the context of SDP are considered as independent variables. We introduce 32 statement-level metrics to capture the complexity of a statement, which may adversely influence to its fault-proneness. In particular, we define 10 *external-linear* and 22 *internal-linear* metrics, which are listed in Tables 1 and 2, respectively. External-linear metrics capture external, contextual characteristics that may influence on the complexity of a statement. Internal-linear metrics estimate the complexity of a statement based on the existing properties in that statement itself. Note that we take each line as the unit of our computation. As an example, the occurrence of a statement in a recursive function is an external-linear metric. In contrast, the number of binary operators used in a statement is an instance of an internal-linear metric.

To achieve effective metrics, we leverage the insights used in the literature in defining coarse-grained metrics. Then, we adapt them to obtain fine-grained, relevant metrics. The metric suite we provided is intended to have the following properties:

- They capture different static features of a statement
- The metrics are expressive enough to distinguish as much similar statements as possible
- They reflect some complexity aspects of a statement that relates to fault-proneness

It is worth mentioning that the list of the statement-level metrics presented here is not intended to be comprehensive; many other metrics can be defined. However, we consider only the existing metrics on the assumption that they are representative of any other metrics that can be defined. We note also that the current metric suite can be used for C-based programming languages, particularly for C and *C*++. However, this is not a limitation to our technique. One can easily redefine the metrics to adapt new target language. We give more arguments regarding these points in Section 5.

3.2. Learning model: long short-term memory

Most of the current SDP research relies on machine-learning techniques to construct a learning model. The reason is that

Table 1
External-linear statement-level metrics introduced for the SLDeep.

ID	Metric	Description
1	Function	Is the line located in a function
2	Recursive Function	Is the line located in a recursive function
3	Blocks Count	The number of nested blocks in which the line is located
4	Recursive Blocks Count	The number of nested recursive blocks in which the line is located
5	FOR Block	The number of nested `FOR` blocks in which the line is located
6	DO Block	The number of nested `DO WHILE` blocks in which the line is located
7	WHILE Block	The number of nested `WHILE` blocks in which the line is located
8	IF Block	The number of nested `IF` blocks in which the line is located
9	SWITCH Block	The number of nested `SWITCH` blocks in which the line is located
10	Conditional Count	The number of single conditions checked to reach a line. This includes the number of components in a compound condition as well as nested conditionals

Table 2
Internal-linear statement-level metrics introduced for the SLDeep.

ID	Metric	Description
11	Literal String	The number of string literals in a line
12	Integer Literal	The number of integer literals in a line
13	Literal Count	The total number of literals in a line
14	Variable Count	The number of variables in a line
15	IF Statement	The number of `IF` conditions in a line
16	FOR Statement	The number of `FOR` loops in a line
17	WHILE Statement	The number of `WHILE` loops in a line
18	DO Statement	The number of `DO WHILE` loops in a line
19	SWITCH Statement	The number of `SWITCH` in a line
20	Conditional and Loop Count	The number of loops and conditionals in a line
21	Variable Declaration	The number of declared variables in a line
22	Function Declaration Count	The number of declared functions in a line
23	Variable Declaration Statement	The number of statements in which a variable is declared in a line
24	Declaration Count	The number of declaration statements in a line
25	Pointer Count	The number of pointers in a line
26	User-Defined Function Count	The number of non-library functions called in a line
27	Function Call Count	The number of called functions in a line
28	Binary Operator	The number of binary operators used in a line
29	Unary Operator	The number of unary operators used in a line
30	Compound Assignment Count	The number of compound assignments in a line
31	Operator Count	The total number of operators in a line
32	Array Usage	The number of arrays used in a line

machine-learning models are tolerant to the imprecise and partially incorrect data (Malhorta, 2015). For our SLDeep, we use an effective machine-learning technique, namely long short-term memory (LSTM), which belongs to the deep-learning sub-category. We first give some required descriptions regarding the LSTM.

LSTM was invented in 1997 to address some of the mathematical issues introduced when modeling long sequences. Today, it is also used to model the relationships between a sequence and other sequences. LSTM belongs to recurrent neural networks (RNN) family. A feed-forward neural network, which is extended to support feedback connections, is called an RNN. An LSTM uses self-loops to establish paths in which the gradient can flow for long durations. In addition, the weights of self-loops are not fixed; they are determined dependent upon the context. Self-loops serve as internal recurrence within units called LSTM cells. These cells have the same input/outputs as ordinary RNNs. However, they have more parameters and several gating units that control the information flows (Goodfellow et al., 2016).

Several studies (Graves, Mohamed & Hinton, 2013; Sutskever, Vinyals and Le, 2014; Graves, 2012) showed that LSTM networks are more effective in learning long-term dependencies than simple recurrent networks. Now, LSTM is increasingly used at Google to model sequences in natural language processing. LSTMs are also widely used for handwritten recognition, machine translation, and parsing (Goodfellow et al., 2016).

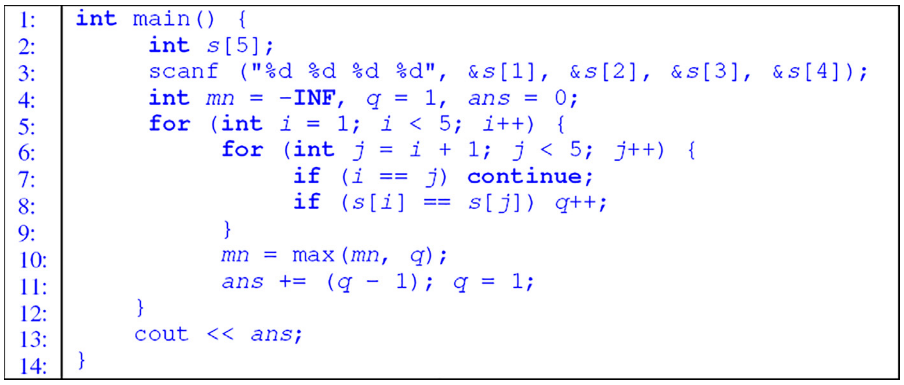
In the following, we describe and justify the usage of LSTM in SLDeep. For SDP problem, we need a binary classifier that is able to classify any statement as either fault-free or fault-prone. Working at statement-level implies that we take every statement into consideration. This is why the amount of produced data is increased by orders of magnitude. Hence, the scalability of the model essentially matters, particularly for adoption of SLDeep in industry. The nature of our data is partially different from typical data in the literature. In addition to metrics values, we augment the corresponding row data of a statement, every token of that statement (see Section 3.3). Finally, we need a model with memorization to consider the code structure as a sequence of statements, not each statement in isolation. The reason lies in the fact that we seek to apply a model that can learn based on what has been observed so far and to make more informative decisions. As a result of these requirements, we found that LSTM could be a relevant model.

3.3. SLDeep

Now, we integrate the introduced metric suite with LSTM to develop our proposed SDP system. Fig. 1 demonstrates the overall structure of SLDeep. To train an LSTM model, a matrix so called M_P should be established per each program P. The i^{th} row of M_P is devoted to the i^{th} line (statement) of P. The columns of M_P are divided into two broad categories. The first category is devoted to 32 metrics introduced in Section 3.1 and is fixed for every program. The second category of columns is devoted to the individual tokens present in each line. This category is of paramount importance as it serves us as a means to capture the code structure itself. The second category can also break any tie introduced using the code metrics in the first category.

Table 3
The structure of the matrix constructed for each program to be given to LSTM model.

Rows/Columns	Metric$_1$	Metric$_2$	…	Metric$_{32}$	Token$_1$	Token$_2$	…	Token$_{max}$	Class
Line$_1$	$m_{1,1}$	$m_{1,2}$	…	$m_{1,32}$	$(t_1, v_1)_1$	$(t_2, v_2)_1$	…	$(t_{max}, v_{max})_1$	fault-free/fault-prone
…	…	…	…	…	…	…	…	…	…
Line$_n$	$m_{n,1}$	$m_{n,2}$	…	$m_{n,32}$	$(t_1, v_1)_n$	$(t_2, v_2)_n$	…	$(t_{max}, v_{max})_n$	fault-free/fault-prone



```
1:  int main() {
2:       int s[5];
3:       scanf ("%d %d %d %d", &s[1], &s[2], &s[3], &s[4]);
4:       int mn = -INF, q = 1, ans = 0;
5:       for (int i = 1; i < 5; i++) {
6:            for (int j = i + 1; j < 5; j++) {
7:                 if (i == j) continue;
8:                 if (s[i] == s[j]) q++;
9:            }
10:           mn = max(mn, q);
11:           ans += (q - 1); q = 1;
12:      }
13:      cout << ans;
14: }
```

Fig. 2. A sample code snippet from Code4Bench.

A token is a sequence of one or more characters that comprise a lexical unit within the source program (Aho, Lam, Sethi & Ullman, 2007). Keywords, operators, numbers, delimiters, and identifiers are examples of tokens. Particularly, we tokenize each line to obtain the comprising tokens. Each token is stored as a pair of (t, v), in which t is the token type and v is the token value. For tokens of the type IDENTIFIER, we consider a number, which is unique only across a program, not all programs. The numbers are assigned according to the order they are visited and are considered as token values for identifiers. We compute the maximum number of tokens across all lines of all programs as max($NT_{i,j}$). Therefore, we would have:

$$\text{The number of columns} = 32 + max\left(NT_{i,j}\right) \tag{1}$$

NT: number of tokens for the j^{th} row of program P_i,

$1 \le i \le P$, $1 \le j \le r_i$, P: the number of programs; r_i: the number of rows of program P_i.

The tokens of each line are then padded alongside the metrics values. For rows whose tokens are less than max(t), we add sufficient pairs of (0, 0). Table 3 shows the overall structure of the resulting matrix. The last column of the matrix is devoted to whether each line is fault-free or fault-prone.

The overall metric and token information are intended to discriminate very similar sequences of statements. For instance, consider a sequence of statements as $a = 0$; $c = 2 / a$; and a second sequence of statements as $a = 1$; $c = 2 / a$; in which they differ only in an integer literal at the first statement. The metric suite together with the extracted tokens would discriminate these two sequences.

To use SLDeep, one needs to compute metric data, extract tokens and construct a matrix for a given program (project). Then an LSTM model is trained and used to predict whether a new unseen statement is either fault-free or fault-prone. As an example of how we extract the presented metrics, consider the following code snippet in Fig. 2, which we have adopted from Code4Bench.

In Table 4, we have depicted the value of each metric per statement. The row numbers in the leftmost column correspond to the identifiers of statements in Fig. 2. The column numbers in the first row correspond to the identifiers of metrics in Tables 1 and 2.

SLDeep can be used in several scenarios, for instance:

- Train SLDeep using the current or several version(s) of a program and apply it on the next release of the same project
- Train SLDeep using a project and apply it on another project
- Train SLDeep using the versions of several projects and apply it on the next versions of the same or other projects

So far, we provided the details of SLDeep and its components. Then the question arises as to how well SLDeep performs on real programs. We investigate this question in the next section.

4. Empirical studies

To evaluate SLDeep, we have conducted experiments. In this section, we explain the process we followed to evaluate SLDeep. We provide every detail in the process to enable replicating the experiments and reproducing the results. First, we explain the experimental setup and the methodology in conducting the experiments. Then, we describe the performance measures we computed. Finally, we give the obtained results and interpret them.

4.1. Experimental setup

Our experiment is guided by the following research questions:

- **RQ1:** How effective does SLDeep perform as measured in terms of accuracy, precision, recall, and f-measure?
- **RQ2:** How scalable is SLDeep when applied on a large set of real-world programs?
- **RQ3:** How is SLDeep compared with the alternative models such as tree-based ones?
- **RQ4:** How is SLDeep compared with the similar related work to interpret the competitiveness of the study?

To answer the research questions, we used a methodology, which is outlined in Fig. 3. We explain the details of each step in the following.

As subject programs, we used C/C++ codes from the Code4Bench (Majd et al., 2019). This benchmark comprises programs written by different users for different problems. For each pair of problem and user, there exist several versions including faulty and correct ones. The large number of users, problems, and versions of programs introduces much diversity in the available

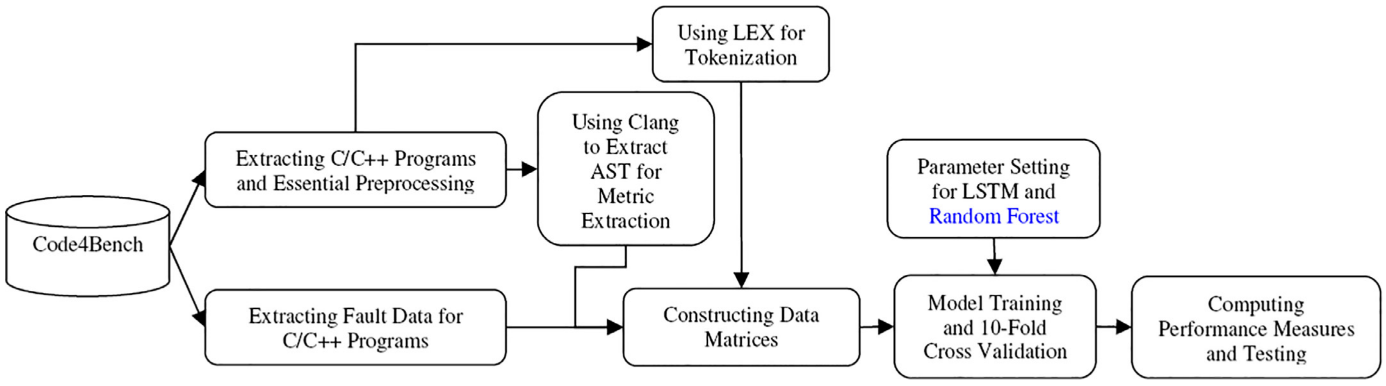

Code4Bench
Extracting C/C++ Programs and Essential Preprocessing
Extracting Fault Data for C/C++ Programs
Using LEX for Tokenization
Using Clang to Extract AST for Metric Extraction
Constructing Data Matrices
Parameter Setting for LSTM and Random Forest
Model Training and 10-Fold Cross Validation
Computing Performance Measures and Testing

Fig. 3. The experimental methodology we followed to evaluate and compare SLDeep.

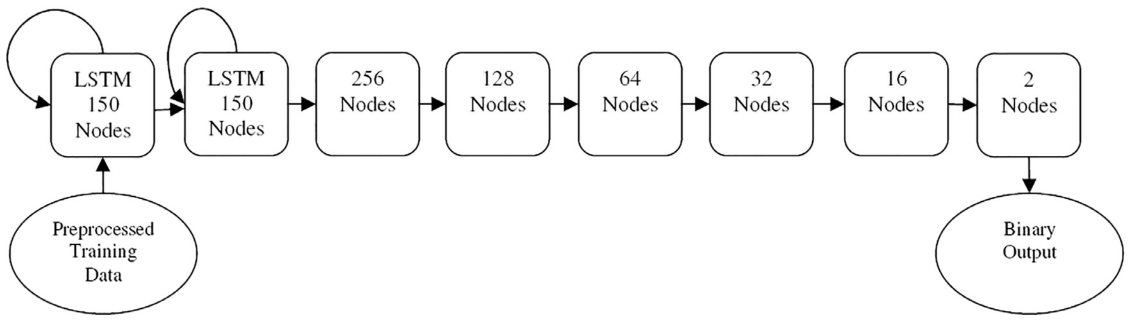


Fig. 4. The high-level topology of the SLDeep learning model.

programs. In the extracted dataset, there exist 119,989 subject programs totaling 2356,458 lines of code with 292,064 faulty lines. For all subject programs, the fault data are present in the benchmark. More precisely, Code4Bench provides tables, in which the correct and faulty versions for C/C++ programs are specified. In addition, the exact faulty lines of defective programs are specified.

A major component of SLDeep is the learning model, which is constructed using LSTM. To achieve the appropriate number of levels and nodes, we made several trials, which is conventional in machine-learning research. Therefore, we note that the parameters of the LSTM we report here are not the best, optimized ones. However, they work well for our experimental data.

Our neural-network model comprises eight layers. The first two layers are LSTMs, each with 150 nodes. The remaining parts of the model are typical neural networks that comprise 256, 128, 64, 32, 16, and 2 nodes for the third to the eighth layers, respectively. Fig. 4 depicts a high-level topology of our model.

The loops in the first two layers denote feedbacks. The first LSTM layer takes as input the original data and a reverse version of data. That is, the first layer reads the first row of each matrix through the last row. Then, it reads the data inversely from the last row of each matrix through the first row. This is meant to give the model better inference of the code structure.

Further details of the layers are as follows:

- Bidirectional LSTM (150 nodes), with dropout (chance of each node dropping out) on recurrent node set to 0.1
- LSTM (150 nodes), with dropout set to 0.2
- A dense (fully connected) layer of 256 nodes with ReLU[7] as activation function and dropout set to 0.2, and batch normalization on its output and another dropout on the output of the batch normalization set to 0.2
- A dense layer of 128 nodes with ReLU as activation function and batch normalization on its output and dropout after the batch normalization with dropout set to 0.25
- A dense layer of 64 nodes with ReLU as activation function and batch normalization on its output and dropout after the batch normalization with dropout set to 0.3
- A dense layer of 32 nodes with ReLU as activation function and batch normalization on its output and dropout after the batch normalization with dropout set to 0.4
- A dense layer of 16 nodes with ReLU as activation function and batch normalization on its output and dropout after the batch normalization with dropout set to 0.4
- A dense layer of 2 nodes with ReLU as activation function. This is the last layer which predicts the output of each line.

In order to demonstrate the effect of LSTM on SLDeep, we conduct an additional experiment in which we employ another classifier rather than LSTM. We chose Random Forest (RF) because in the context of CPDP, decision tree-based models have led to better results (Hosseini et al., 2017). SLDeep is also a CPDP system and an instance of transfer learning.

RF is an ensemble method that comprises a collection of classifiers (Han, Pei & Kamber, 2011). The aim of an ensemble is to improve the accuracy of the overall model. Each classifier in a RF is a decision tree. However, RF is intended to offer a finer resolution than a single decision tree. RF has led to impressive results in many applications such as computer security (Canfora, Iannaccone & Visaggio, 2014). It is more robust to outliers and errors and is not subject to overfitting.

[7] The rectified linear unit (ReLU) is an activation function, which is a default choice in modern neural networks (Goodfellow, 2016).

Table 4
The values of metrics presented in Tables 1 and 2 for the statements in the code snippet presented in Fig. 2.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	2	0	0	0	0	0	0	0	0
3	1	0	1	0	0	0	0	0	0	0	1	4	5	4	0	0	0	0	0	0	0	5	0	5	4	0	1	0	4	0	4	4
4	1	0	1	0	0	0	0	0	0	0	0	2	2	1	0	0	0	0	0	0	3	1	1	5	0	0	0	0	1	0	1	0
5	1	0	2	0	1	0	0	0	0	1	0	2	2	2	0	1	0	0	0	1	1	2	1	4	0	0	0	1	1	0	2	0
6	1	0	3	0	2	0	0	0	0	2	0	2	2	3	0	1	0	0	0	1	1	3	1	5	0	0	0	2	1	0	3	0
7	1	0	4	0	2	0	0	1	0	3	0	0	0	2	1	0	0	0	0	0	0	2	0	2	0	0	0	1	0	0	1	0
8	1	0	4	0	2	0	0	1	0	3	0	0	0	5	1	0	0	0	0	0	0	5	0	5	0	0	0	1	1	0	2	2
9	1	0	3	0	1	0	0	0	0	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
10	1	0	2	0	1	0	0	0	0	1	0	0	0	3	0	0	0	0	0	0	0	4	0	4	0	0	1	1	0	0	1	0
11	1	0	2	0	1	0	0	0	0	1	0	2	2	3	0	0	0	0	0	0	0	3	0	3	0	0	0	2	0	1	3	0
12	1	0	2	0	1	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
13	1	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	2	0	2	0	0	0	0	0	0	0	0
14	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

We used Clang 5.0.1[8] tool to build AST of subject programs. Then, we used the ASTs to compute the metrics on each subject program. To run LSTM and RF, we used Keras,[9] TensorFlow,[10] and scikit-learn.[11] For tokenization of statements, we used the LEX and YACC tools.[12] We carried out the experiment on a system with CPU Core i7-6800k @ 3.40, 64GB of RAM, and graphics NVIDIA GeForce GTX 1070 Ti. Note that machine-learning tools often leverage both CPU and GPU to achieve better efficiency.

As an essential preprocessing on programs, the empty lines and comments were eliminated. In some cases, Clang was unable to build the program's AST. Thus, we excluded these programs of our dataset. In the resulting dataset, there might be some noise or outlier data. However, we did not investigate or exclude them on the assumption that our model handles them. Except for these, no other preprocessing was necessary.

To construct the prediction model, we run the tuned neural network with the matrices of all programs as input. Each matrix corresponds to a program. For our experimental data, we run the model for about one week using 10-fold cross-validation. The long-lasting execution of our model can be credited to two reasons. First, our training data is significantly voluminous. Second, we have used 10-fold cross-validation for model evaluation. Nevertheless, this long-lasting execution can be justified by considering two advantages. First, the resulting model tends to show universal properties, as it has learnt data from several projects and can be used on several other projects. Second, the training is one-time processing whose costs are amortized to its usage in numerous applications.

4.2. Performance measures

To evaluate the effectiveness of SLDeep, we computed four performance measures, which are the most common measures in the SDP literature (Malhorta, 2015). The measures are Accuracy, Precision, Recall, and F-Measure, which are computed using Eqs. (2) to 5, respectively. In the equations, true positive (TP) refers to the number of positive tuples (rows, lines of code) that were correctly labeled by the classifier as positive. True negative (TN) refers to the number of negative tuples that were correctly labeled by the classifier as negative. False positive (FP) refers to the number of negative tuples that were incorrectly labeled as positive. Finally, false negative (FN) refers to the number of positive tuples that were incorrectly labeled as negative. In our context, positive refers to the fault-prone line. We report the raw values of TP, TN, FP, and FN to enable computing other desirable measures. They help to facilitate the quantitative synthesis of the results for future systematic studies and meta-analysis.

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{2}$$

$$\Pr ecision = \frac{TP}{TP + FP} \tag{3}$$

$$\mathrm{Re}\, call = \frac{TP}{TP + FN} \tag{4}$$

$$F - measure = 2 \times \frac{precision \times recall}{precision + recall} \tag{5}$$

Accuracy measures the recognition rate. That is, it refers to the percent of tuples that were correctly labeled. Precision measures

[8] https://clang.llvm.org/index.html.
[9] https://keras.io/.
[10] https://www.tensorflow.org/.
[11] https://scikit-learn.org/stable/.
[12] http://dinosaur.compilertools.net/.

Table 5
The experimental results of training SLDeep on subject programs when using different learning models.

Fold	Model	Accuracy Neighborhood	Recall	Precision	Accuracy	F-Measure	TP	FN	FP	TN
1	LSTM	2	0.984978892	0.43127532	0.605148625	0.599888188	607,337	9262	800,898	634,313
		4	0.987425214	0.581132626	0.716556601	0.731659353	792,859	10,097	571,475	677,379
	RF	2	0	0	0.69949	0	0	616,599	0	1,435,211
		4	0.207931692	0.578506956	0.630745049	0.305910526	166,960	635,996	121,645	1,127,209
2	LSTM	2	0.982750624	0.432426376	0.607182926	0.600585356	605,966	10,636	795,350	639,858
		4	0.98853014	0.578351014	0.712971474	0.729752103	795,143	9226	579,702	667,739
	RF	2	0	0	0.69948	0	0	616,602	0	1,435,208
		4	0.195329507	0.582395831	0.629638222	0.292543099	157,117	647,252	112,660	1,134,781
3	LSTM	2	0.983132095	0.428904672	0.602984682	0.597250665	603,999	10,363	804,237	633,211
		4	0.983146712	0.590579938	0.726890891	0.737900352	788,814	13,522	546,846	702,628
	RF	2	0	0	0.70058	0	0	614,362	0	1,437,448
		4	0.179345062	0.576398486	0.627552746	0.27356958	143,895	658,441	105,750	1,143,724
4	LSTM	2	0.989569282	0.423145905	0.591815519	0.592804825	609,639	6426	831,091	604,654
		4	0.991646332	0.570326224	0.703819554	0.724163694	797,717	6720	600,986	646,387
	RF	2	0	0	0.69975	0	0	616,065	0	1,435,745
		4	0.18049	0.57849	0.6157	0.27514	16,628	75,497	12,116	123,740
5	LSTM	2	0.982394989	0.431728399	0.606522046	0.599845557	605,117	10,844	796,498	639,351
		4	0.986830983	0.583162788	0.718277033	0.733102408	793,870	10,594	567,448	679,898
	RF	2	0	0	0.6998	0	0	615,961	0	1,435,849
		4	0.144328398	0.576854667	0.623003105	0.230888699	116,107	688,357	85,169	1,162,177
6	LSTM	2	0.986392563	0.425779062	0.595627275	0.594807732	608,982	8401	821,295	613,132
		4	0.983296593	0.590351031	0.725060313	0.73776384	793,541	13,480	550,644	694,145
	RF	2	0	0	0.6991	0	0	617,383	0	1,434,427
		4	0.174066102	0.575031417	0.624545158	0.267237509	140,475	666,546	103,816	1,140,973
7	LSTM	2	0.989352518	0.420698738	0.588418031	0.590360604	608,526	6549	837,939	598,796
		4	0.989584227	0.573254314	0.707019168	0.725965494	796,264	8381	592,760	654,405
	RF	2	0	0	0.70023	0	0	615,075	0	1,436,735
		4	0.205115299	0.57464521	0.628734142	0.302319809	165,045	639,600	122,167	1,124,998
8	LSTM	2	0.98950596	0.420599382	0.587125514	0.590290077	610,259	6472	840,668	594,411
		4	0.990092333	0.573723858	0.707410043	0.726478721	797,257	7978	592,361	654,214
	RF	2	0	0	0.69942	0	0	616,731	0	1,435,079
		4	0.194082473	0.574462689	0.627294925	0.290140761	156,282	648,953	115,767	1,130,808
9	LSTM	2	0.986381317	0.428541051	0.600356271	0.597495517	608,617	8403	811,590	623,200
		4	0.988530153	0.579906696	0.714088536	0.730989271	797,040	9248	577,388	668,134
	RF	2	0	0	0.69928	0	0	617,020	0	1,434,790
		4	0.20095797	0.574933292	0.627620491	0.297818607	162,030	644,258	119,794	1,125,728
10	LSTM	2	0.981714363	0.43421284	0.607080804	0.602111437	610,000	11,362	794,841	635,626
		4	0.984355478	0.585824357	0.719645253	0.734513846	795,751	12,647	562,593	680,838
	RF	2	0	0	0.69717	0	0	621,362	0	1,430,467
		4	0.133296965	0.578697786	0.620294869	0.216683223	107,757	700,641	78,449	1,164,982
Average	LSTM	2	0.98561726	0.427731175	0.599226169	0.596543996				
		4	0.987343817	0.580661285	0.715173887	0.731228908				
	RF	2	0	0	0.69943	0				
		4	0.181494347	0.577041633	0.625512871	0.275225181				

the exactness of the model. That is, it states what percent of the tuples that were labeled as positive, are really positive. Recall measures completeness of the model. It states what percent of positive tuples are labeled as positive. Recall is an important measure for SDP context because it refers to the power of the model in predicting fault-prone lines of code. Finally, F-Measure is the harmonic mean between precision and recall (Han et al., 2011).

4.3. The results

Tables 5 and 6 present the experimental results of applying SLDeep on the mentioned subject programs. The results are reported based on the performance measures described in the last section. Table 5 shows the performance measures of the trained model while the results in Table 6 indicate the performance measures of the tested model. The rows in Table 5 correspond to the rows in Table 6. That is, they show the training and testing measures of the same model in the same fold and in the same iteration. Each part of both tables corresponds to a certain fold in a 10-fold cross-validation. In the tables, we have reported the results of applying SLDeep as well as applying a RF model on the subject programs. In our experiments, we considered an *accuracy neighborhood*. That is, for a fault-free statement labeled as fault-prone by the model, we take the prediction as true if the really faulty statement is at most *n* lines before or after that statement. The accuracy neighborhood is considered due to our fined-grained statement-level analysis. We repeated our experiments for two iterations where *n* is set to four and two.

In Addition to Tables 5 and 6, Fig. 5 demonstrates the average performance measures for SLDeep and RF at training and testing stages. Particularly, the left and right charts correspond to SLDeep and RF, respectively. Also, the top and bottom charts correspond to training and testing stages. In each chart, two groups of bars are shown. The left and right groups are devoted to accuracy neighborhoods two and four, respectively. In each group, the values of four performance measures, namely recall, precision, accuracy, and f-measure, averaged over 10-folds are depicted from left to right, respectively.

The results in Table 5 indicate how well the model fits on the training data. At the first iteration, the accuracy neighborhood is set to two. This means that for a fault-free statement labeled as fault-prone by the model, we take the prediction as true if the really faulty statement is at most two lines before or after that statement. By this assumption, at the test stage SLDeep could achieve average performance measures 0.967, 0.415, 0.580, and 0.580 in terms of recall, precision, accuracy, and f-measure, respectively. At the next iteration, we set accuracy neighborhood to four. At the test stage, SLDeep could achieve average performance measures 0.979, 0.570, 0.702, and 0.721 in terms of recall, precision, accuracy, and f-measure, respectively.

Table 6
The experimental results of testing SLDeep on subject programs when using different learning models.

Fold	Model	Accuracy Neighborhood	Recall	Precision	Accuracy	F-Measure	TP	FN	FP	TN
1	LSTM	2	0.955813581	0.398706792	0.552695181	0.562692705	65,608	3033	98,944	60,396
		4	0.979574887	0.554338617	0.675709818	0.708014091	89,636	1869	72,063	64,413
	RF	2	0	0	0.69892	0	0	68,641	0	159,340
		4	0.187443309	0.550855895	0.612520342	0.27970842	17,152	74,353	13,985	122,491
2	LSTM	2	0.960284391	0.406192225	0.565393607	0.570898981	65,912	2726	96,356	62,987
		4	0.97878835	0.553751185	0.679916309	0.707329496	88,181	1911	71,062	66,827
	RF	2	0	0	0.69893	0	0	68,638	0	159,343
		4	0.185099676	0.551327404	0.618446274	0.277150383	16,676	73,416	13,571	124,318
3	LSTM	2	0.962174441	0.420447469	0.575907641	0.585183564	68,197	2681	94,004	63,099
		4	0.971137042	0.579277926	0.703321768	0.725687333	89,466	2659	64,978	70,878
	RF	2	0	0	0.68911	0	0	70,878	0	157,103
		4	0.180493894	0.578485945	0.615700431	0.275140855	16,628	75,497	12,116	123,740
4	LSTM	2	0.972157571	0.407542527	0.562735491	0.574321266	67,249	1926	97,762	61,044
		4	0.985603839	0.554259014	0.681324321	0.709517488	88,728	1296	71,356	6660
	RF	2	0	0	0.69658	0	0	69,175	0	158,806
		4	0.1798	0.58014	0.62741	0.27452	144,638	659,799	104,676	1,142,697
5	LSTM	2	0.957216473	0.41807728	0.582123072	0.581970882	66,315	2964	92,304	66,398
		4	0.975454737	0.570148207	0.69999693	0.719659304	87,788	2209	66,186	71,798
	RF	2	0	0	0.69612	0	0	69,279	0	158,702
		4	0.142860318	0.576779866	0.620257829	0.229000427	12,857	77,140	9434	128,550
6	LSTM	2	0.97120415	0.420053285	0.59232129	0.586458672	65,903	1954	90,989	69,135
		4	0.972118024	0.582904166	0.72251635	0.728802006	85,002	2438	60,823	79,718
	RF	2	0	0	0.70236	0	0	67,857	0	160,124
		4	0.17694419	0.584378305	0.636057391	0.271638751	15,472	71,968	11,004	129,537
7	LSTM	2	0.977894962	0.433508555	0.599909642	0.600715283	68,614	1551	89,662	68,154
		4	0.981272824	0.5881246	0.721889105	0.735455122	88,134	1682	61,722	76,443
	RF	2	0	0	0.69223	0	0	70,165	0	157,816
		4	0.220406164	0.608939063	0.637105724	0.323662375	19,796	70,020	12,713	125,452
8	LSTM	2	0.982162927	0.417270782	0.582465205	0.585705332	67,287	1222	93,968	65,504
		4	0.985082823	0.573670985	0.70764669	0.725083629	87,895	1331	65,320	73,435
	RF	2	0	0	0.6995	0	0	68,509	0	159,472
		4	0.215901195	0.595247659	0.635667007	0.31687077	19,264	69,962	13,099	125,656
9	LSTM	2	0.967736734	0.418602144	0.588145503	0.584412193	66,019	2201	91,694	68,067
		4	0.979199982	0.570169124	0.706457994	0.720692821	86,339	1834	65,088	74,720
	RF	2	0	0	0.70076	0	0	68,220	0	159,761
		4	0.20916834	0.580662427	0.635719643	0.307549923	18,443	69,730	13,319	126,489
10	LSTM	2	0.959250446	0.404872344	0.593476106	0.569412049	61,275	2603	90,069	74,015
		4	0.977830194	0.576084501	0.719979646	0.725024123	84,155	1908	61,926	79,973
	RF	2	0	0	0.71979	0	0	63,878	0	164,084
		4	0.122143081	0.585855208	0.63598319	0.202142184	10,512	75,551	7431	134,468
Average	LSTM	2	0.966589568	0.41452734	0.579517274	0.580177093				
		4	0.97860627	0.570272833	0.701875893	0.720526541				
	RF	2	0	0	0.69943	0				
		4	0.182026017	0.579267177	0.627486783	0.275738409				

Although we have achieved significantly high recall in the experiments on SLDeep, however, the precision is not so good. We investigate this result from two aspects. First, this result is not uncommon in the SDP literature. In fact, the best models have typically achieved high recall at the cost of precision (Hosseini et al., 2017). In addition, in the context of software defect prediction, recall is the key, determinant measure because it shows the percent of the really faulty statements that are labeled as faulty by the model. A high recall means that there is little likelihood for our model to miss a faulty statement. Second, we attribute this result to the level of granularity in our analysis which is extremely fine-grained. Statements in the programs are appeared in much diverse forms. Many different statements can be semantically equivalent. There are numerous degrees of freedom in the ways a statement can be written. A same algorithm can be implemented in many different ways in a programming language. The judgment on the faultiness of a statement is based on the program it appears and the specifications that the program must fulfill. That is, the faultiness of a statement is a context-sensitive property. A same statement appearing in two different programs may be fault-free in the first program and faulty in the second one. This makes the learning model confused. Therefore, the sequence of all statements in a program should be considered when learning the faultiness of a statement. Fortunately, LSTM has been designed to learn such sequential data. However, in addition to the diversity in the forms a statement can appear, another important factor contributes to the low precision in our experiments. This factor is a specific property of the Code4Bench programs that we used in our experiments. Code4Bench includes vast number of program groups where each group comprises structurally-redundant programs. Each group corresponds to the different versions of a program that a developer has submitted for a problem. The versions are nearly the same except for few changed statements. As a result, there exist many identical statements in different versions with very similar sequences, in which one statement is labeled as fault-free and another is labeled as faulty. This situation makes distinguishing of the statements challenging even for an LSTM. Basically, the issue does not arise due to some limitations in the LSTM; but the issue relates to the lack of extra distinguishing information in the mentioned situation. The adverse effect of the described issues causes the model to identify many of the fault-free statements as fault-prone which in turn leads to the low precision.

The cases in which accuracy neighborhood is greater than zero imply that SLDeep may sometimes predict at quasi statement-level. Nevertheless, the amount of accuracy neighborhood is small enough to offer significant benefits for the developer over the traditional module/file/class/function levels.

These results can answer our first and second research questions mentioned in Section 4.1. SLDeep seems to be effective in classifying statements as fault-free or fault-prone. In addition, the

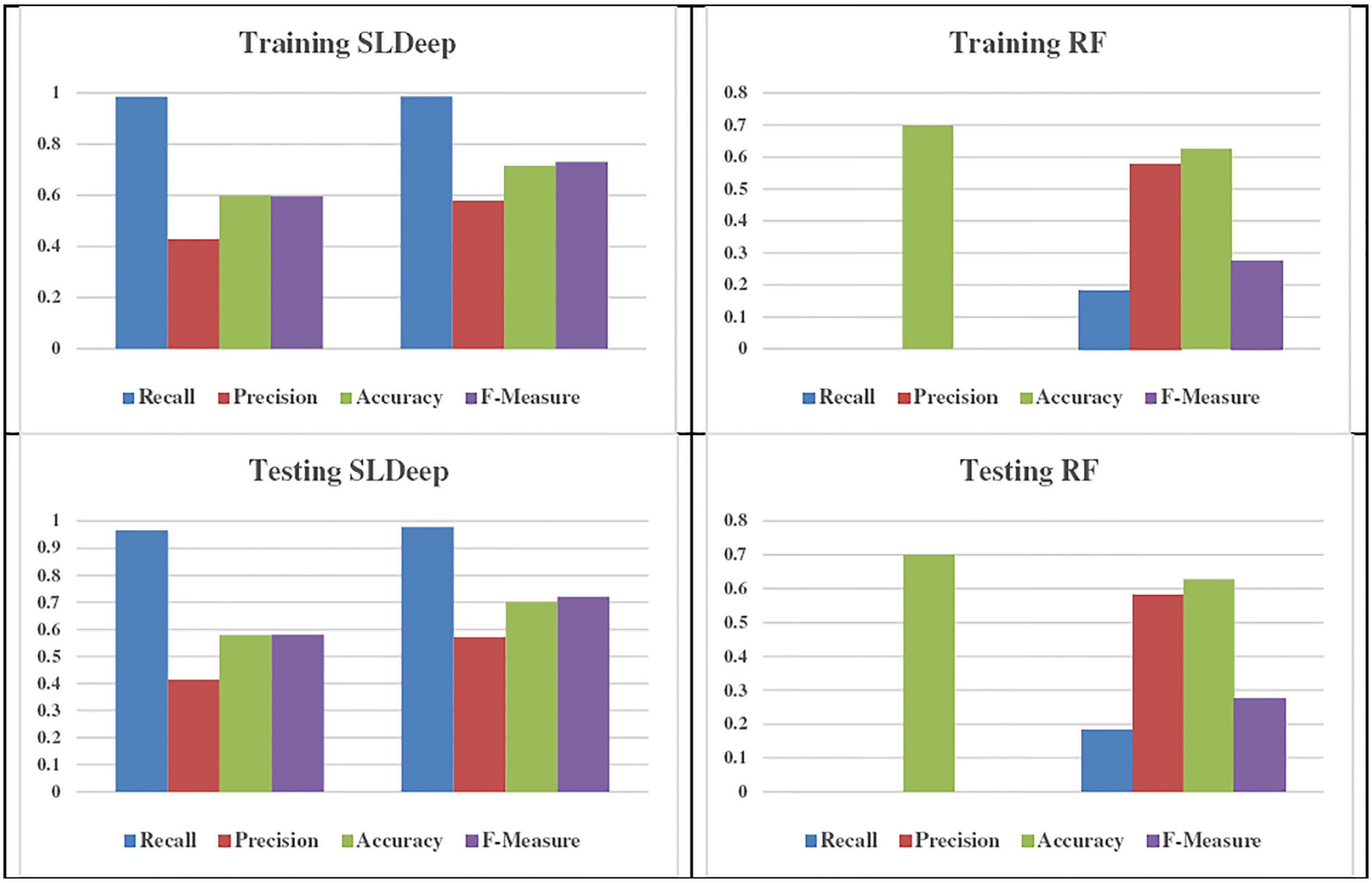


Fig. 5. Comparison of SLDeep and RF at the training and testing phases. In each chart, two groups of bars are shown. The left and right groups are devoted to accuracy neighborhoods two and four, respectively. Each group comprises four bars. The bars are dedicated to recall, precision, accuracy, and f-measure, from left to right, respectively.

results suggest that SLDeep is scalable since it could work on more than 2300,000 training data (lines of code) and yet it has achieved very high effectiveness.

At the test stage of the experiments on RF, we could achieve a recall of 0.182 and 0 for accuracy neighborhood set to 4 and 2, respectively. These results can answer our third research question and suggest that RF is quite ineffective for statement-level defect prediction. In order to evaluate the magnitude of statistical results, we have managed for a Cohen *d*'s test on the mean values of obtained recalls when applying LSTM and RF for the accuracy neighborhood 4 at the testing stage. For our calculated *d* value 38.069975 and based on the Cohen's suggestion (Kampenes, Dybå, Hannay & Sjøberg, 2007), it turns out that the mean recall values differ by a large effect size, when applying LSTM and RF as the learning model.

We have also managed for another experiment, in which we used K-Nearest-Neighbor (KNN) as learning model. After nearly three days of model execution, we obtained the results only for the first fold. Particularly, we have achieved 0.647 and 0.634 for the recall measure at the training and testing stages, respectively. Then, we stopped running the model due to our limitations in computational resources. Although the results of the first fold are not enough to draw conclusive results, however, based on the results in Tables 5 and 6, we conjecture that the remaining folds produce similar results.

Our fourth research question is associated with how well SLDeep is compared to the related work. Most of the studies in the SDP literature are focused on the prediction at file, class, module, or function granularities, which are substantially coarser than our statement-level granularity. In fact, a major novelty of our work lies in its fine-grained granularity, for which we had to employ specific statement-level code metrics. This feature hinders a fair comparison to any of the studies in the literature. The competitiveness of our study to the body of literature can be interpreted considering that it helps the developers to pinpoint the faulty locations more quickly. With existing techniques, a coarse-grained unit such as a function is labeled to be fault-prone. Then, the developer should employ a fault localization technique to find a list of statements that are highly likely to be faulty. Using SLDeep, however, exempts us from the fault localization step since it specifies the approximate locations of the faulty statements.

Overall, the results suggest that accounting for statement-level analysis, together with leveraging LSTM model as learner, has much potential to yield an effective, scalable SDP system. It can lead to software with higher quality in less time and effort. Therefore, SLDeep proved relevant; it can be successfully adopted at industrial scale for both WPDP and CPDP. After quantitative evaluation of SLDeep, we should discuss around different aspects of its design and performance. The next section is dedicated to this goal.

5. Discussion

So far, we provided the details of design and implementation of SLDeep and reported the results of evaluation. Yet, some high-level viewpoints, considerations, and implications are still unexplored. We should also discuss around the limitations and threats to the validity of the results. These aspects are covered in this section.

Most of the SDP research assumes that sufficient fault data are available (Catal, 2011). However, this is not always the case. For example, a company may deal with a new project or fault data have not been collected for some parts or the whole project of the previous version. Even, the deployment of appropriate tools and collection of fault data might be too costly to be used by a company (Hall, Bowes, Liebchen & Wernick, 2010). Similarly, collecting metric data may be difficult or they may be unavailable at the beginning of a project (Radjenović et al., 2013). In such cases, applying cross-project SDP systems such as SLDeep can be beneficial.

For SLDeep, we have not used any feature selection or correlation analysis on the metrics, despite the studies (Hall et al., 2011) that have shown the advantages of feature selection on SDP techniques. The reason behind this design decision is twofold. First, we assume the LSTM model can partly address the redundancies in the data and does not incorporate them in its further computations. Second, we see this point as a benefit that exempts us from a non-trivial preprocessing and moves SLDeep one step forward to become automated.

In the literature, there exist numerous deep learning models. However, we chose to adopt LSTM for our SDP problem. This is because LSTMs are generally used to model the sequence data. That is, one can model dependency with an LSTM model. This property fits best to our application. However, this is not the case for other deep learning models. As an instance, consider stacked autoencoders, which have been widely used in many applications such as image processing and classification. Autoencoders are trained to retain as much input data as possible (Goodfellow et al., 2016). They try to approximate the representation of the original, input data and are trained to produce a new representation with various good properties. During the training, the network tries to minimize the difference between the approximate representation and the original data. These characteristics do not satisfy the requirements of processing source code sequence data.

The quality of the data essentially matters for research in SDP, particularly cross-project defect prediction (Hosseini et al., 2017). The data should be up to data, indicating the current practices in software development. To meet this requirement, we used Code4Bench (Majd et al., 2019), which is the most recent benchmark comprising a rich set of programs developed in recent years.

As suggested by Hosseini et al. (Hosseini et al., 2017), we provided several performance measures to give better grasp of the nature and capabilities of the applied model. They reported that most of the SDP research has used the default parameters of the learning models they utilized. As our model is a complex one with numerous components, the default parameter setting led to a model, which was ineffective. Therefore, we had to manage for some trials to reach parameter settings for which SLDeep shows effectiveness. This is not abnormal since most of the current classifiers involve at least a parameter that needs to be carefully tuned (Tantithamthavorn, McIntosh, Hassan & Matsumoto, 2016).

The programs we used in the experiments involve different versions of the programs developed by many developers for many problems. As a result, the training data is diverse along four dimensions: statements, developers, problems (projects), and versions. This property seems to result in a universal model. In principle, universality implies that the model can be used on any version of any program. In practice, however, this is not the case for SLDeep and its training data. That said, diversity among four dimensions implies that SLDeep can be used for both within-project and cross-project defect prediction of multiple versions of the programs, at least at the context of the programs we used. Although we have not provided explicit evidence for this claim, for the specific programs of Code4Bench, this claim is more likely to be aptly stated. The reason is that Code4Bench contains of many versions with little difference. In addition, it comprises many programs that are developed for the same problem. We note that universality may lead to reduction in the precision of the model. This point manifests an important trade-off between universality and effectiveness.

The metrics suite and the experiments focus on C/C++ languages. However, this choice has been an implementation necessity and should not imply as a limitation for SLDeep. Newer programming languages such as C# or Java have been designed based on the structures and experiences in C/C++ (Sebesta, 2016). As a result, it seems that the metric suite would work on C# and Java with little changes. The metrics are designed to capture static features of the codes. Therefore, for a language with different grammar and structure, one needs only to redefine the metrics to capture the intended feature. Hence, we expect the SLDeep shows effectiveness when implemented on other programming languages. Future work will explore this hypothesis.

In real-world contexts, faults are relatively stochastic and fault data are highly imbalanced (Radjenović et al., 2013). Therefore, the number of faults and their distribution may differ significantly between the two versions. 10-fold cross validation may not consider all the factors present in real-world environments. However, such evaluations are common in the literature. Additional evaluations that take the mentioned factors into account remain as future work.

Most of the SDP studies including our own have not incorporated fault severity into the model. Although some faults are more severe than others, severity is hard to measure (Hall et al., 2011). Researchers (Böhme, 2014) proposed to measure the severity of a fault based on the patch that fixes that fault. Considering that Code4Bench contains correct versions of the corresponding fault versions, some research can be conducted to train a model incorporating fault severities. This is an avenue for further research.

In the experiments, we did not explicitly investigate whether a trained model on the i^{th} version of a project, will be effective on the $(i + n)^{\text{th}}$ version where $n > 1$. Note that this question has been implicitly confirmed to hold partially true because we train on multiple versions of multiple programs and test on some other versions of some programs. For a developer whose coding style and programming practices may remain unchanged through a project, the model may be effective through consecutive regression versions. Nonetheless, concrete evidence necessitates curated experiments and remains as future work.

A possible threat to the external validity is the representative of the programs. The programs in Code4Bench are not industrial projects and the resulting model may not be effective for such projects. We mitigate this threat by using more than 100,000 programs totaling 2356,458 lines of code, developed by thousands of users for thousands of problems. Since we are working at statement-level, the very large amounts of statements seem to provide sufficient diversity into the training data. A threat concerning the construct validity is the performance measures we used. However, the measures we used are common in the context of SDP research. The threats to the internal validity can be errors introduced when data collection or the tools we used. For data collection, another co-author checked a sample of data. As to the helper tools, we note that they are developed by well-known institutes or companies and are widely-used in code analysis and machine-learning research.

6. Conclusions and future work

The goal of this paper is to alleviate the developer's burden in pinpointing the locations of faults, hence provide high-quality software with less time and effort. To achieve this goal, we proposed SLDeep, a technique for statement-level software defect prediction. To realize SLDeep, we defined 32 statement-level metrics and leveraged long short-term memory as learning model. We evaluated SLDeep on more than 100,000 C/C++ programs from the

Code4Bench. In the experiments, SLDeep could successfully predict fault-prone statements with average performance measures 0.979, 0.570, and 0.702 in terms of recall, precision, and accuracy, respectively. Based on these results, SLDeep seems to be effective at statement-level software defect prediction and can be adopted. The significance of SLDeep lies in its novel usage of LSTM for a practical problem in the context of software engineering. It can lead to save software development resources and more reliable software.

This work can be extended in several ways. First, we can explore to define additional metrics for C-based languages. Second, we can adapt the current metrics for other common programming languages such as Python and Kotlin. Third, new variants of LSTM can be used to exploit their potentials. Fourth, Empirical studies can be conducted to investigate the effects of layers and nodes of LSTM in the overall performance of the model. Fifth, some experiments can be carried out on programs in other application domains, for example Android applications. Finally, empirical studies can be conducted to understand the success factors of SLDeep in different contexts. This is particularly valuable for practitioners to know the cases that SLDeep should be preferred to other techniques.

Contribution statement

All authors have contributed to the scientific and write-up contents of the paper.

Declaration of Competing Interest

None.

Acknowledgements

We thank the anonymous reviewers of the journal whose comments helped us to make the revised version of paper stronger. In addition, we are grateful to the cloud-computing center at Iran University of Science and Technology for providing the access to their system.

References

Aho, A. V., Lam, M. S., Sethi, R., & Ullman, J. D. (2007). *Compilers, principles, techniques. second edition*. Pearson Education Inc.

Bird, C., Bachmann, A., Aune, E., Duffy, J., Bernstein, A., Filkov, V., et al. (2009). Fair and balanced?: Bias in bug-fix datasets. in *proceedings of the the 7th joint meeting of the European software engineering conference and the acm sigsoft symposium on the foundations of software engineering. ACM*, 121–130.

Böhme, M., & Roychoudhury, A. (2014). Corebench: Studying complexity of regression errors. in *proceedings of the 2014 international symposium on software testing and analysis. ACM*, 105–115.

Boucher, A., & Badri, M. (2018). Software metrics thresholds calculation techniques to predict fault-proneness: An empirical comparison. *Information and Software Technology, 96*, 38–67.

Bowes, D., Hall, T., & Petrić, J. (2018). Software defect prediction: Do different classifiers find the same defects? *Software Quality Journal, 26*(2), 525–552.

Canfora, G., Iannaccone, A. N., & Visaggio, C. A. (2014). Static analysis for the detection of metamorphic computer viruses using repeated-instructions counting heuristics. *Journal of Computer Virology and Hacking Techniques, 10*(1), 11–27.

Canfora, G., Lucia, A. D., Penta, M. D., Oliveto, R., Panichella, A., & Panichella, S. (2015). Defect prediction as a multiobjective optimization problem. *Software Testing, Verification and Reliability, 25*(4), 426–459.

Catal, C. (2011). Software fault prediction: A literature review and current trends. *Expert systems with applications, 38*(4), 4626–4636.

Catal, C., & Diri, B. (2009). Investigating the effect of dataset size, metrics sets, and feature selection techniques on software fault prediction problem. *Information Sciences, 179*(8), 1040–1058.

Choudhary, G. R., Kumar, S., Kumar, K., Mishra, A., & Catal, C. (2018). Empirical analysis of change metrics for software fault prediction. *Computers & Electrical Engineering, 67*, 15–24.

Dam, H.K., .Pham, T., Ng, S.W., .Tran, T., Grundy, J., Ghose, A. et al. (2018). A deep tree-based model for software defect prediction. arXiv preprint arXiv:1802.00921.

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

Fan, G., Diao, X., Yu, H., Yang, K., & Chen, L. (2019). *Software defect prediction via attention-based recurrent neural network*. Scientific Programming 2019.

Fenton, N., & Bieman, J. (2014). *Software metrics: A rigorous and practical approach*. CRC press.

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

Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep learning*. MIT press.

Graves, A. (2012). *Supervised sequence labelling with recurrent neural networks*. Berlin Heidelberg: Springer-Verlag.

Graves, A., Mohamed, A. R., & Hinton, G. (2013). Speech recognition with deep recurrent neural networks. In*2013 IEEE international conference on acoustics, speech and signal processing. IEEE*, 6645–6649.

Hall, T., Beecham, S., Bowes, D., Gray, D., & Counsell, S. (2011). A systematic literature review on fault prediction performance in software engineering. *IEEE Transactions on Software Engineering, 38*(6), 1276–1304.

Hall, T., Bowes, D., Liebchen, G., & Wernick, P. (2010). *Evaluating three approaches to extracting fault data from software change repositories. in* international conference on product focused software process improvement (pp. 107–115). Berlin, Heidelberg: Springer.

Han, J., Pei, J., & Kamber, M. (2011). *Data mining: Concepts and techniques*. Elsevier.

Hosseini, S., Turhan, B., & Gunarathna, D. (2017). A systematic literature review and meta-analysis on cross project defect prediction. *IEEE Transactions on Software Engineering*.

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

Kampenes, V. B., Dybå, T., Hannay, J. E., & Sjøberg, D. I. (2007). A systematic review of effect size in software engineering experiments. *Information and Software Technology, 49*(11–12), 1073–1086.

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

Lanza, M., Mocci, A., & Ponzanelli, L. (2016). The tragedy of defect prediction, prince of empirical software engineering research. *IEEE Software, 33*(6), 102–105.

Lee, T., Nam, J., Han, D., Kim, S., & In, H. P. (2016). Developer micro interaction metrics for software defect prediction. *IEEE Transactions on Software Engineering, 42*(11), 1015–1035.

Majd, A., Vahidi-Asl, M., Khalilian, A., Baraani-Dastjerdi, A., & Zamani, B. (2019). Code4Bench: A multidimensional benchmark of codeforces data for different program analysis techniques. *Journal of Computer Languages*.

Malhotra, R. (2015). A systematic review of machine learning techniques for software fault prediction. *Applied Soft Computing, 27*, 504–518.

Manjula, C., & Florence, L. (2018). Deep neural network based hybrid approach for software defect prediction using software metrics. *Cluster Computing*, 1–17.

Menzies, T., Greenwald, J., & Frank, A. (2006). Data mining static code attributes to learn defect predictors. *IEEE Transactions on Software Engineering, 33*(1), 2–13.

Nuñez-Varela, A. S., Pérez-Gonzalez, H. G., Martínez-Perez, F. E., & Soubervielle-Montalvo, C. (2017). Source code metrics: A systematic mapping study. *Journal of Systems and Software, 128*, 164–197.

Özakıncı, R., & Tarhan, A. (2018). Early software defect prediction: A systematic map and review. *Journal of Systems and Software, 144*, 216–239.

Palomba, F., Zanoni, M., Fontana, F. A., De Lucia, A., & Oliveto, R. (2017). Toward a smell-aware bug prediction model. *IEEE Transactions on Software Engineering*.

Pan, C., Lu, M., Xu, B., & Gao, H. (2019). An improved CNN model for within-project software defect prediction. *Applied Sciences, 9*(10), 2138.

Qiao, L., & Wang, Y. (2019). Effort-aware and just-in-time defect prediction with neural network. *PloS one, 14*(2), e0211359.

Radjenović, D., Heričko, M., Torkar, R., & Živkovič, A. (2013). Software fault prediction metrics: A systematic literature review. *Information and Software Technology, 55*(8), 1397–1418.

Rathore, S. S., & Kumar, S. (2017). Towards an ensemble based system for predicting the number of software faults. *Expert Systems with Applications, 82*, 357–382.

Sebesta, R. W. (2016). *Concepts of programming languages* (11 th Ed.). Pearson.

Shippey, T., Bowes, D., & Hall, T. (2019). Automatically identifying code features for software defect prediction: Using AST N-grams. *Information and Software Technology, 106*, 142–160.

Sutskever, I., Vinyals, O., & Le, Q. V. (2014). Sequence to sequence learning with neural networks. In *Advances in neural information processing systems* (pp. 3104–3112).

Tantithamthavorn, C., McIntosh, S., Hassan, A. E., & Matsumoto, K. (2016). Automated parameter optimization of classification techniques for defect prediction models. in *2016 IEEE/ACM 38th international conference on software engineering (ICSE). IEEE*, 321–332.

Turhan, B., Menzies, T., Bener, A. B., & Di Stefano, J. (2009). On the relative value of cross-company and within-company data for defect prediction. *Empirical Software Engineering, 14*(5), 540–578.

Yang, X., Lo, D., Xia, X., Zhang, Y., & Sun, J. (2015). Deep learning for just-in-time defect prediction. in 2015 ieee international conference on software quality, reliability and security. *IEEE*, 17–26.

Zimmermann, T., Nagappan, N., Gall, H., Giger, E., & Murphy, B. (2009). Cross-project defect prediction: A large scale experiment on data vs. domain vs. process. in *proceedings of the the 7th joint meeting of the European software engineering conference and the ACM sigsoft symposium on the foundations of software engineering. ACM*, 91–100.