The Journal of Systems & Software 195 (2023) 111537

Contents lists available at ScienceDirect

The Journal of Systems & Software

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

On the use of deep learning in software defect prediction ☆

Görkem Giray [a,*], Kwabena Ebo Bennin [b], Ömer Köksal [c], Önder Babur [b,d], Bedir Tekinerdogan [b]

[a] Independent Researcher, Izmir, Turkey
[b] Information Technology Group, Wageningen University & Research, Wageningen, The Netherlands
[c] ASELSAN, Ankara, Turkey
[d] Department of Mathematics and Computer Science, Eindhoven University of Technology, Eindhoven, The Netherlands

ARTICLE INFO

Article history:
Received 24 June 2022
Received in revised form 3 October 2022
Accepted 5 October 2022
Available online 12 October 2022

Keywords:
Software defect prediction
Deep learning
Quality assurance
Systematic literature review

ABSTRACT

Context: Automated software defect prediction (SDP) methods are increasingly applied, often with the use of machine learning (ML) techniques. Yet, the existing ML-based approaches require manually extracted features, which are cumbersome, time consuming and hardly capture the semantic information reported in bug reporting tools. Deep learning (DL) techniques provide practitioners with the opportunities to automatically extract and learn from more complex and high-dimensional data.
Objective: The purpose of this study is to systematically identify, analyze, summarize, and synthesize the current state of the utilization of DL algorithms for SDP in the literature.
Method: We systematically selected a pool of 102 peer-reviewed studies and then conducted a quantitative and qualitative analysis using the data extracted from these studies.
Results: Main highlights include: (1) most studies applied supervised DL; (2) two third of the studies used metrics as an input to DL algorithms; (3) Convolutional Neural Network is the most frequently used DL algorithm.
Conclusion: Based on our findings, we propose to (1) develop more comprehensive DL approaches that automatically capture the needed features; (2) use diverse software artifacts other than source code; (3) adopt data augmentation techniques to tackle the class imbalance problem; (4) publish replication packages.

1. Introduction

Software reliability and quality mainly depend on removing faults or defects in software. Although some defects might arise from causes unrelated to code (such as compilers or byte code representations), the main source of software faults is software code. The traditional way of finding software defects is by testing and conducting reviews. However, these activities may require extensive time and effort. On the other hand, automatic prediction of defective software modules at early stages may guide developers in improving code quality at a reduced cost compared to a fully manual approach (Wahono, 2015). To this end, software defect prediction (SDP) aims to promptly identify potential faults in the software and is a promising approach to improving software quality (Lessmann et al., 2008; Menzies et al., 2006). Therefore, SDP has become an important research topic in software engineering and testing in recent years.

Predicting defect-prone parts of software before discovering faults by performing substantial efforts is a challenging task. The main challenge of SDP is identifying the faulty parts of source code with better fault prediction performance. To this end, diverse methods and techniques have been proposed and reported in the literature for many years. Many researchers use learning-based algorithms to have better accuracies in SDP; on the other hand, some research has focused on the semantic representation of the source code. Researchers have been using machine learning (ML) and, more recently, deep learning (DL) algorithms to develop efficient SDP models. ML-based SDP techniques require manual extraction of features mainly based on software metrics. Although software metrics are effective indicators of defective portions of software (Rodríguez et al., 2012), manually extracted features are time-consuming to construct in the first place, and

Abbreviations: SE, Software Engineering; ML, Machine Learning; DL, Deep Learning; SDP, Software Defect Prediction; WPDP, Within Project Defect Prediction; CPDP, Cross Project Defect Prediction; HDP, Heterogeneous Defect Prediction; SMS, Systematic Mapping Study; SLR, Systematic Literature Review; RQ, Research Question

☆ Editor: Aldeida Aleti.

* Corresponding author.
E-mail addresses: gorkemgiray@gmail.com (G. Giray), kwabena.bennin@wur.nl (K.E. Bennin), koksal@aselsan.com.tr (Ö. Köksal), onder.babur@wur.nl (Ö. Babur), bedir.tekinerdogan@wur.nl (B. Tekinerdogan).

https://doi.org/10.1016/j.jss.2022.111537

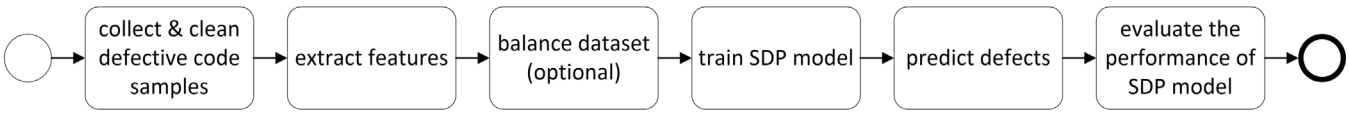

Fig. 1. Software defect prediction process.

they hardly capture semantic information reported in bug reporting tools. On the other hand, DL-based techniques automatically extract higher-level features and learn from more complex and high-dimensional data (LeCun et al., 2015). Therefore, many researchers have recently focused on developing SDP models using DL-based techniques.

DL covers extensive state-of-the-art techniques and algorithms. Therefore, many studies applied these algorithms and techniques to the SDP domain comparing their findings with other studies. Hence, this paper systematically identifies, analyzes, summarizes, and synthesizes the current state of developing SDP models using DL algorithms and techniques. We conducted our survey using systematic literature review (SLR), a well-defined method introduced by Kitchenham and Charters (2007). Further, we indicate and investigate the recent research trends and point out future research directions on SDP. To this end, we selected a pool of 102 peer-reviewed studies and then conducted a quantitative and qualitative analysis using the data extracted from these studies. Researchers and practitioners may benefit from this survey to understand the state-of-the-art DL usage for SDP and shape their efforts to build more effective and efficient SDP models.

The rest of the paper is organized as follows: Section 2 provides a background and the related work. Section 3 explains the research methodology. Section 4 presents the results. Section 5 discusses our findings and reports the threats to validity. Section 6 concludes the paper.

2. Background and related work

2.1. Software defect prediction

SDP mainly involves prediction models that are built to predict faulty parts of software. Although diverse techniques and algorithms have been applied in order to have better performing (e.g., more accurate) SDP models, the main steps of SDP can be summarized as in Fig. 1: (1) collect clean and defective code samples from software repositories, (2) extract features to create a dataset, (3) balance the dataset if it is imbalanced, (4) train a prediction model on the dataset, (5) predict the faulty parts for a dataset extracted from a new software (different version of trained dataset or new software project), and finally, (6) evaluate the performance of the SDP model. This process is iterative; Fig. 1 ignores iteration steps for the sake of simplicity.

Fig. 1 shows that the process starts with collecting clean and defective code samples. Software data can be found in different formats comprising of source codes, commit messages, bug reports and other software artifacts. These data are usually extracted from archives and repositories.

The second step in SDP is feature extraction. During this phase, the software artifacts/source codes/commit logs and messages among others are converted to metrics which is used as input data for model training. The input data type, ranging from McCabe metrics (McCabe, 1976), CK metrics (Chidamber and Kemerer, 1994), change histories, assembly code, to source code, as well as the representation of the data are crucial in the feature extraction step. Besides metric-based data, nowadays, several DL techniques provide automatic extraction of features and learn from more complex and high dimensional data. In many studies in the literature, defect data from popular public defect repositories, such as the NASA (Shepperd et al., 2013) and PROMISE (Jureczko and Madeyski, 2010) datasets, have been utilized.

The next step is usually an optional step. This phase involves balancing the data since defect datasets typically include much fewer faulty parts than non-faulty. Unfortunately, most SDP techniques suffer from this class imbalance problem as several metrics for evaluating SDP performance result in misleading results due to the imbalanced structure of classes (Bennin et al., 2016). A variety of techniques, such as oversampling, can be used to tackle this issue and increase SDP performance.

The fourth step of SDP is determining the faulty parts of the software. The main concern in this step is the selection of DL algorithms and techniques, which can involve a wide range of architectures (e.g., Convolutional Neural Networks) and machine learning categories (e.g., supervised or not). In addition, the granularity of the faulty parts to be detected is an important issue at this step: these can be e.g., at module, file, class, function, or sentence level.

The next step is predicting the faulty parts of new (test) data using the trained model in the previous step. The prediction performed here provides the input for the last step of the SDP process.

The final step of the SDP process is evaluating the model developed. The SDP model can be evaluated utilizing various metrics such as F-measure or area under curve. One or more of such metrics are used to evaluate the prediction models and compare them with other related studies.

Orthogonal to the variety of choices in several steps of the process as outlined above, SDP studies can also be categorized with respect to their scenarios. Traditionally, two main SDP scenarios are used in the literature: Within-Project Defect Prediction (WPDP) and Cross-Project Defect Prediction (CPDP). In WPDP, the historical data of a project (i.e., different versions) is used to predict the faulty parts (Omri and Sinz, 2020), i.e., WPDP focuses on fault predictions within the same software project on which it is trained (Ni et al., 2017). Hence, both the training and test sets belong to the same project. On the other hand, CPDP uses data from other projects (source projects) to train an SDP model and use this model to predict the faulty parts of another project (target project) (Chen et al., 2019). This approach originated from transfer learning and has particular importance where the target project may have inadequate labeled data for training. However, the main complication of this approach is to minimize the feature distribution difference between source projects and the target project.

The main obstacle in CPDP is that all projects used in the CPDP scenario must use the same metrics. In contrast, Heterogeneous Defect Prediction (HDP) enables defect prediction across projects with different metrics, mapping data from source and target projects into a common metric space (Chen et al., 2022; Nam et al., 2018).

In addition to these SDP scenarios, Just in Time Software Defect Prediction (JIT-SDP) is another popular approach (Kamei et al., 2013) aiming to predict software defects at the software change level (Cabral et al., 2019). JIT-SDP (also called change level defect prediction) enables developers identify and fix defects on time ensuring software quality. It has particular importance in SDP since it provides on time guidance to developers at a

finer granularity, i.e., change level. Using JIT-SDP, developers can immediately review and test their changes without time consuming code reviews and extensive tests (Kamei et al., 2013; Zheng et al., 2022). A final novel SDP approach is Cross-Version Defect Prediction (CVDP) which utilizes the fault data in prior versions of the same project to predict the current version of the software project (Zhang et al., 2020b).

2.2. Deep learning in SDP

Deep Learning is a subfield of ML based on Artificial Neural Networks (ANNs), simply called Neural Networks (NNs) with multiple layers (Arar and Ayan, 2015). The neural network model yields a prediction for each input in supervised, unsupervised, or semi-supervised training (Jorayeva et al., 2022). First, the error between these predictions and the actual results is calculated according to a previously simplified loss function. Later, the gradients of this function concerning the model's parameters are computed in a process called backpropagation and used for updating them in the next step with the help of an optimizer (Goodfellow et al., 2016). This section briefly introduces the most used DL models in SDP.

Multi-Layer Perceptron (MLP) is the fundamental structure of feedforward neural networks and has multiple layers (Goodfellow et al., 2016). The input layer contains the vectorized input data; hidden layers of interconnected nodes allow the structure to learn transformations on the data. The MLP model learns weights and biases using the backpropagation mechanism and nonlinear activation functions, extracting more advanced features. Finally, the output layer produces output vectors that correspond to the model's prediction of the input's class.

Deep Neural Networks (DNNs) are particular ANNs devised to learn by multi-connection layers (Montavon et al., 2018). The architecture of DNNs includes one input layer, one output layer, and one or more hidden layers between them. The input feature space of the data constitutes the input layer of the DNN. The input can be constructed with feature extraction methods. The output layer has one node in binary classification and has nodes as many as the number of classes in multi-class classification. DNN uses a standard backpropagation algorithm with a nonlinear activation function like sigmoid or Relu (Apicella et al., 2021). With the help of the defined architecture, DNNs extract features from the input data. Then, the model is trained to optimize weight and bias values in the neural network structure. Finally, the trained model is used to predict the class of the new input.

Inspired by the human visual system, Convolutional Neural Network (CNN) uses convolution operations to extract input features. This process is implemented through the multiple sliding kernels, matrices of specified shape and size, also called filters, and the elementwise multiplication of these kernels with the corresponding image data. This operation yields information about various features in the input. The CNN structure is very commonly used in image processing. For example, 1D convolution operations are applicable in diverse areas, including examining sequential data, text, or time-series data to find the patterns in the data (Rao and McMahan, 2019).

Recurrent Neural Networks (RNNs) have feedback loops in their architecture, allowing information to be memorized in short terms. Due to this property, RNN can analyze sequential input, such as speech, audio, weather, and financial data. However, an RNN's output at a stage relies on the previous output and the current input. While CNN shares unknown parameters across space, RNN shares them across time. Nevertheless, RNNs' memory is short-termed, their computation can be slow, and they suffer from the vanishing or exploding gradients problem (Hochreiter and Schmidhuber, 1997). Hochreiter and Schmidhuber (1997) developed the Long Short-Term Memory (LSTM) model to solve the problems with RNN structure. To overcome these issues, additional neural networks, called gates, were introduced, which handle the information stream in the network. Another type of RNN, Gated Recurrent Units (GRUs), were proposed by Chung et al. (2014). In GRUs, the gated approach solves RNN's information flow problems in long sequences with a simpler architecture introducing two gates: the update and the reset gates. Since GRUs have fewer gate structures than LSTMs', they have fewer parameters to change during the training, which leads them to being faster.

Deep Belief Neural Networks (DBNs) are feedforward Neural Networks (NNs) with many layers (Golovko et al., 2014). A DBN is not the same as the traditional types of DNNs discussed so far. Instead, a DBN is a particular DNN with undirected connections between some layers. These undirected layers are Restricted Boltzmann Machines and can be trained using unsupervised learning algorithms.

Encoder–Decoder models (also known as Sequence to Sequence or Seq2Seq models) are commonly used DNN architectures to convert input data in a particular domain into output data in another domain via a two-stage network (Cho et al., 2014; Chollampatt and Ng, 2018). First, the encoder takes a variable-length sequence in a specific domain and compresses it to a fixed-length representation. Then, the decoder maps the encoded data to a variable-length output in another domain. Due to these features, encoder–decoder models are widely used in many application areas, such as machine translation (Cho et al., 2014; Chollampatt and Ng, 2018).

Autoencoders might be considered as specific types of encoder–decoder models (Zhu et al., 2020). Autoencoder is an unsupervised ANN that learns efficient encoding of unlabeled data. First, it learns how to efficiently compress and encode input data. Next, autoencoders learn how to ignore the noise in the data and reduce data dimensions. Then, using the encoded representation, it learns how to reconstruct the data as close as to the original input. Autoencoders are used in many deep learning tasks such as anomaly and face detection. In addition, modified versions of autoencoders are used for specific tasks in deep learning. For example, sparse and denoising autoencoders are used in learning representations for subsequent classification tasks. Variational autoencoders are used in generative tasks to produce similar outputs to the input data. In SDP, autoencoders' main use is to extract features of input data automatically (Tong et al., 2018; Zhu et al., 2020; Wu et al., 2021; Zhang et al., 2021b).

Extreme learning machines (ELMs) are special feedforward neural networks invented by Huang et al. (2006). ELM architecture includes single or multiple layers of hidden nodes whose parameters need not be tuned. The hidden nodes of ELMs can be assigned randomly. No update operation is performed for these nodes, or they can be inherited from their antecessors without being changed. Generally, these nodes' weights are learned in a single step converging to a linear model. Hence, these models might be much faster than backpropagation-based neural networks. Moreover, these models might produce comparable results with SVM in classification and regression tasks (Liu et al., 2005, 2012).

Generative Adversarial Networks (GANs) are another approach used in generative modeling, designed by Goodfellow et al. (2014). Generative modeling is an unsupervised task. GANs use deep learning methods, such as CNN, to produce new outputs similar to the input data acquired from the original dataset. GANs use two neural networks named generator and discriminator. The generator is a CNN, and the discriminator is a de-convolutional NN. These networks compete in a game where one agent's gain is another agent's loss to predict more accurately. In this game, the

generator produces artificial data similar to the real data, and the discriminator tries to distinguish the artificially generated data from the original data. The generator produces better artificial outputs as the game continues, and the discriminator will detect them better. In this way, GANs learn to generate new data with the same statistics as the training set.

Siamese Neural Networks (SNNs) are NNs that contain two or more subnetworks whose configurations, parameters, and weights are the same. Moreover, parameters are updated in both networks in the same way. In this way, Siamese NN compares its feature vectors and finds the similarity of the inputs by learning a similarity function. Hence it can be trained to check whether two inputs (for example, images of a person) are the same. Hence, SNN's architecture enables new data classification without retraining the network and making them suitable for one-shot learning problems. Furthermore, Siamese NNs are robust to class imbalance and learn semantic similarity. So, SNNs were used in several types of research in the SDP domain, although they require more training time than NNs (Zhao et al., 2018, 2019).

Hierarchical Neural Network (HNN) is a special NN that consists of multiple loosely coupled subnets defined in the form of an acyclic graph. The subnets of the graph can be single neurons or complex NNs. Each subnet tries to acquire a specific figure of the input data (Mavrovouniotis and Chang, 1992). They are being used in various deep learning-based tasks such as classification (Wang et al., 2012) and image interpretation (Behnke, 2003). Further, HNNs have been used in SDP to provide better fault predictions (Wang et al., 2021; Yu et al., 2021a).

Graph Neural Networks (GNNs) are NNs designed to leverage the structure and properties of graphs. GNNs perform inference on data described by graphs by using deep learning methods. Hence, GNNs can be used in graph operations performing node-level, edge-level, and graph-level predictions. GNNs are active research topics in many domains such as social networks, knowledge graphs, and recommender systems (Chen et al., 2021; Kumar et al., 2022). GNNs are also used in the SDP domain to take full advantage of the tree structure of the source code. To this end, GNNs are exploited to acquire the inherent defect information of faulty subtrees, which are excluded based on a fix-inducing change (Xu et al., 2021a).

2.3. Related work

Researchers use ML models obtained from SE data (source code, requirement specifications, test cases, etc.) to effectively and efficiently engineer software (Giray, 2021). Watson et al. (2020), Yang et al. (2020), and Ferreira et al. (2021) surveyed how DL has been used to solve SE problems in general. Yang et al. (2020) identified that SDP is the most popular sub-problem under testing and debugging problems for which DL is applied. Ferreira et al. (2021) found out that SDP is one of the top three problems SE researchers are dealing with. Although a few papers that use DL for SDP are included in these reviews, their number is very low compared to our study due to their search strings involving terms that are more generic (like "software engineering"). In addition, these reviews do not include a detailed analysis and synthesis of SDP studies.

About a decade ago, researchers started to synthesize the results of the studies to understand the progress in SDP. Catal and Diri (2009) analyzed 74 studies published between 1990 and 2007 to present a consolidated view of the use of ML and statistical techniques for SDP. They observed a significant increase in the number of primary studies in 2007 compared to previous years. Hall et al. (2011) synthesized the quantitative and qualitative results of 36 results published from January 2000 to December 2010. Malhotra (2015) analyzed 64 studies to understand the use of ML for SDP for the period of 1991 and 2013. In contrast to these studies, this study focuses on the synthesis of the studies that used DL algorithms for SDP.

Some review studies targeted a specific subarea of SDP. Hosseini et al. (2017) and Goel et al. (2017) focused on CPDP. Özakıncı and Tarhan (2018) reviewed the studies on early SDP, which utilized the metrics gathered earlier in the software development life cycle, such as metrics on requirements, design artifacts, and source code. Radjenović et al. (2013) identified software metrics and assessed their applicability in SDP. Li et al. (2020) investigated the use and performance of unsupervised learning techniques in SDP. Matloob et al. (2021) examined ensemble learning techniques for SDP. Different from these review studies, this study focuses on the use of DL for SDP in general.

Table 1 lists the related work on the use of DL for SDP. Eight of these studies did not follow a systematic review research method. Son et al. (2019) conducted a systematic mapping study on the use of ML and DL for SDP by examining 156 studies. Malhotra et al. (2020) analyzed 20 primary studies to explore the use of DL for software quality prediction. Pandey et al. (2021) recently published a systematic review study on the use of ML and DL for SDP. Their study covers the primary studies published until June 2019 of which 36 are addressing DL for SDP.

Recently, Batool and Khan (2022) and Pachouly et al. (2022) published two SLR papers on the use of ML and DL for SDP. Both include studies published between 2010 and 2021 (both papers were submitted to a journal before the end of 2021). In addition, both studies included traditional ML and data mining techniques besides DL in their review. The SLR by Batool and Khan (2022) covers 11 primary studies focusing on DL. Pachouly et al. (2022) did not report the number of primary studies particularly using DL; they rather mention that they include 146 primary studies in total. On the other hand, their analysis covers only DBN, CNN, RNN/LSTM, and MLP excluding other types of DL approaches such as encoder–decoder architectures, GAN, and hybrid DL models. Compared to these two SLRs, this study reflects the state-of-the-art of the use of DL for SDP by analyzing 102 primary studies. Batool and Khan (2022) and Pachouly et al. (2022) provided information on the datasets, evaluation metrics, and ML/DL approaches used in SDP. Additionally, Pachouly et al. (2022) provided an analysis on the tools/frameworks and challenges related to datasets, such as class imbalance. This study includes analyses on these items except tools and frameworks and additionally presents information on the representation of source code, granularity level of prediction, validation approaches, and reproducibility package. Finally, this study includes a qualitative analysis on the challenges and proposed solutions on the whole aspects of SDP unlike the analysis of Pachouly et al. (2022) focusing on the challenges related to datasets.

This study differs from the related studies by involving a unique combination of the following characteristics: (1) focusing particularly on the use of DL for SDP, (2) with substantial level of depth on several aspects of DL-based SDP, (3) achieving a good coverage of the literature including 102 primary studies published until the end of 2021, and (4) following a systematic literature review research method.

3. Research objectives and method

This section describes the research objectives and the method used in this study. We adopted a systematic literature review (SLR) approach to synthesize the knowledge on the use of DL algorithms for SDP. The research method is based on established guidelines (Kitchenham and Charters, 2007; Wohlin, 2014), some previous good examples of SLRs on SDP (Hall et al., 2011; Hosseini et al., 2017), and our previous experience in conducting SLRs

Table 1
Summary of related work.

Reference	Year	Type of review	# of primary studies	Time period covered	Scope
Li et al. (2018)	2018	Non-systematic	70	Jan 2014–Apr 2017	ML & DL for SDP
Kalaivani and Beena (2018)	2018	Non-systematic	Not reported	Not reported	ML & DL for SDP
Prasad and Sasikala (2019)	2019	Non-systematic	Not reported	Not reported	ML & DL for SDP
Rathore and Kumar (2019)	2019	Non-systematic	Not reported	1993–2017	ML & DL for SDP
Son et al. (2019)	2019	Systematic mapping	156	1995–2018	ML & DL for SDP
Omri and Sinz (2020)	2020	Non-systematic	Not reported	Not reported	ML & DL for SDP
Guan et al. (2020)	2020	Non-systematic	Not reported	Not reported	ML & DL for SDP
Malhotra et al. (2020)	2020	Systematic mapping	20	Jan 1990–Jan 2019	DL for software quality prediction
Akimova et al. (2021)	2021	Non-systematic	Not reported	2019–2021	DL for SDP
Atif et al. (2021)	2021	Non-systematic	Not reported	Not reported	Statistics, ML & DL for SDP
Pandey et al. (2021)	2021	Systematic literature review	154 out of which 36 are on DL	1990–June 2019	ML & DL for SDP
Batool and Khan (2022)	2022	Systematic literature review	68 out of which 11 involve DL	2010–2021 (partial)	Data mining, ML & DL for SDP
Pachouly et al. (2022)	2022	Systematic literature review	146 (number of studies involving DL not reported)	2010–2021 (partial)	ML & DL
This study	2022	Systematic literature review	102	Until the end of 2021	DL for SDP

(Garousi et al., 2019; Giray and Tüzün, 2018; Tarhan and Giray, 2017). Table 2 summarizes the SLR protocol used in this study using the format adopted from Motta et al. (2018). We performed four main activities: (1) defining the goal and the research questions (RQs), (2) selecting relevant primary studies, (3) extracting data, and (4) synthesizing data and reporting the results. The details are described in the following subsections.

3.1. Goal and research questions

The scope and goal of this study were formulated using the Goal-Question-Metric approach (Basili et al., 1994) as follows.

Analyze *the state-of-the-art in software defect prediction (SDP)*

for the purpose of *exploration and analysis*

with respect to *the SDP scenarios, ML categories, datasets, representation of source code, granularity of prediction, techniques for dealing with the class imbalance problem, DL algorithms, evaluation metrics and validation approaches, presence of a reproducibility package, and reported challenges and proposed solutions*

from the point of view of *machine learning researchers*

in the context of *deep learning (DL).*

The goal of this study is to systematically classify, review, and synthesize the body of knowledge and evidence on the use of DL algorithms for SDP. As Kitchenham et al. (2015) pointed out, RQs must embody secondary studies' goals. Accordingly, we raised the following RQs to achieve our goal:

RQ1. Which SDP scenarios (e.g., WPDP, CPDP or HDP) were applied?

RQ2. Which ML categories (i.e., supervised/unsupervised/semi-supervised learning) were applied in DL-based SDP studies?

RQ3. Which public datasets were used for the development and testing of ML/DL models for SDP?

RQ4. How did researchers represent source code to develop DL models for SDP?

RQ5. At which granularity levels did researchers perform SDP?

RQ6. What approaches did researchers follow to cope with class imbalance challenge for SDP?

RQ7. Which DL algorithms (e.g., CNN, LSTM) were applied?

RQ8. What kind of evaluation metrics and validation approaches were used?

RQ9. How often did researchers provide reproduction packages to support the reproducibility of DL models for SDP?

RQ10. What were the challenges and proposed solutions in the use of DL for SDP?

Fig. 2 shows how the RQs are mapped to the ML model life cycle proposed by Amershi et al. (2019). The feedback loops and the iterations throughout the life cycle were omitted for the sake of simplicity. In the model requirements stage, researchers decide on the SDP scenario(s) they will focus and select the most appropriate type of ML category for the SDP problem. During the data engineering stages, teams look for available datasets or construct new ones, clean data if required, and prepare labeled datasets for supervised learning if labels are not already present. Two basic decisions affect the data engineering phase, i.e., how to represent source code (such as via metrics, abstract syntax trees) and the granularity level of prediction (such as file-level, function-level). A very common problem researchers deal with in data engineering phase is class imbalance. Software defect datasets generally have fewer buggy modules than non-buggy ones (Bennin et al., 2017a,b). Unfortunately, such datasets consisting of imbalanced data typically decrease the prediction performance of ML/DL models developed for SDP (Bennin et al., 2017a,b). Feature engineering refers to the activities for extracting and selecting informative features for ML models (Amershi et al., 2019). For some approaches using DL algorithms, the feature engineering stage is less explicit and combined with the model training stage (Amershi et al., 2019). Generally, researchers use more than one ML/DL algorithm to develop models for SDP. In model evaluation, teams evaluate output models using evaluation metrics and approaches to choose the best-performing model. To enable other researchers to obtain the reported experimental results with the same experimental setup, researchers must share source code and datasets (Liu et al., 2021a). During this model life cycle, teams may face some challenges. After model development, the chosen model is deployed and monitored in a production environment. We excluded these two stages since our primary studies did not include any information on these stages.

3.2. Primary study selection

Fig. 3 depicts the process we used for primary study selection. In the first step, we used five widely used online databases,

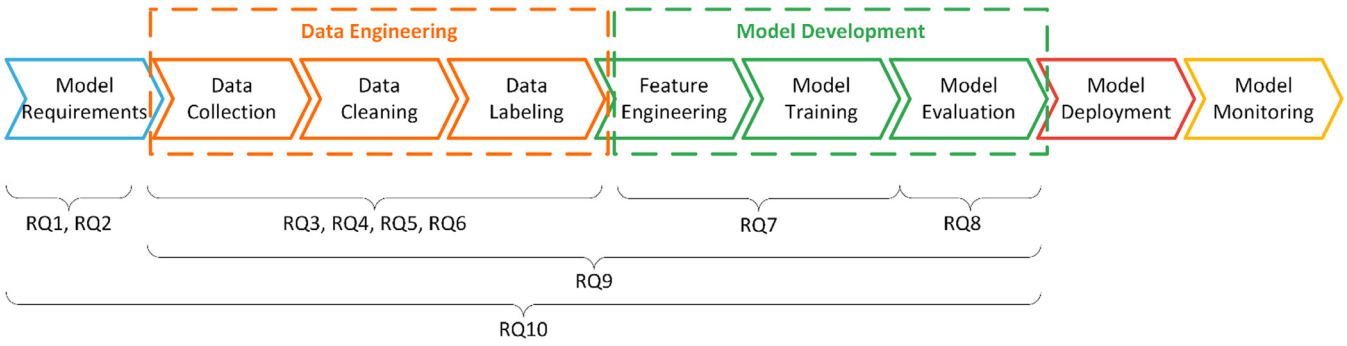

Fig. 2. RQs mapped to the ML model life cycle.
Source: Adapted from Amershi et al. (2019).

Table 2
Protocol summary.

Themes addressed by RQs	**RQ1.** SDP scenarios (e.g., WPDP, CPDP, HDP) **RQ2.** ML categories (i.e., supervised/unsupervised/semi-supervised learning) **RQ3.** Training and testing datasets **RQ4.** Representation of source code (e.g., metrics, Abstract Syntax Tree) **RQ5.** Granularity level of prediction (e.g., file, change, class) **RQ6.** Techniques for dealing with the class imbalance problem (e.g., over-sampling, under-sampling) **RQ7.** DL approaches (e.g., CNN, LSTM) **RQ8.** Evaluation metrics (e.g., F-measure, recall, AUC; etc.) and validation approaches (i.e., cross-validation and hold-out) **RQ9.** Reproducibility package **RQ10.** Challenges and proposed solutions
Search string	Population: software defect/fault/bug/quality prediction/estimation Intervention: deep learning *(software) AND (fault OR defect OR quality OR bug) AND (predict* OR estimat*) AND ("deep learning")*
Search strategy	DB search: ACM, IEEE, ScienceDirect, Springer, Wiley Manual search on the primary studies included in the secondary studies listed in Table 1 Forward snowballing using Google Scholar
Inclusion and exclusion criteria	Inclusion: • The paper is written in English • The paper is published in a scholarly journal or conference/workshop/symposium proceedings. • The paper involves at least one DL algorithm applied to SDP problem and reported empirical results. Exclusion: • The paper's full text is not available. • The paper is an editorial, issue introduction or secondary study (literature review, SMS, SLR). • The paper involves only traditional ML algorithms or statistical techniques applied to SDP problem.
Study type	Primary studies

i.e., ACM, IEEE Xplore, ScienceDirect, Springer, and Wiley as the source of the potentially relevant primary studies. We used the following search string to query these databases (for details see Appendix A.1): *(software) AND (fault OR defect OR quality OR bug) AND (predict* OR estimat*) AND ("deep learning")*. We used paper title, abstract, and keywords as the search fields. We searched each of the five online databases two times in June 2021 and January 2022. In January 2022, we searched the databases to obtain the papers published only in the second half of 2021 and added the new candidate papers to our pool. By doing so, we aimed to involve all primary studies published until the end of 2021. As shown in Table 3, we obtained 296 primary studies in total (238 in June 2021 and additional 58 papers in January 2022) to apply inclusion and exclusion criteria.

To identify the relevant set of papers to answer our RQs, we specified our inclusion and exclusion criteria, as presented in Table 2. Each primary study was assigned to two authors for the application of inclusion and exclusion criteria. Initially, two authors voted on the candidate papers individually whether to include by reading the title, abstract, keywords, and by checking the full text if needed. When the voting results were compared, there was 86% agreement between two authors. In the case of disagreements, two other authors were assigned to investigate whether to include a paper and the conflict was resolved. As the result of the first step, we obtained 72 primary studies for quality assessment.

Table 3
Database search results.

Database	Search in June 2021	Search in January 2022
ACM	22	3
IEEE Xplore	89	23
ScienceDirect	29	6
Springer	83	24
Wiley	22	2
Duplicates removed	7	0
Total for inclusion/exclusion	238	58

In the second step, we checked the primary studies used in the related reviews listed in Table 1 to enrich our paper pool. We applied our inclusion and exclusion criteria to all studies that used DL for SDP in these reviews. This manual search led to the addition of eight primary studies to our pool.

In the third step, we conducted forward snowballing, as recommended by systematic review guidelines (Wohlin, 2014), to

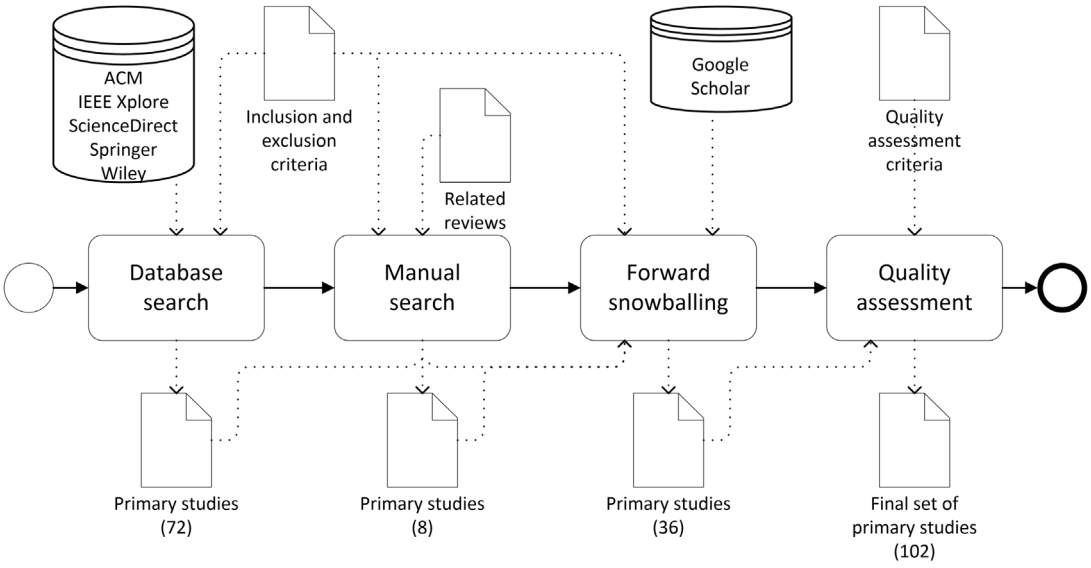

Fig. 3. The primary study selection process.

Table 4
Quality assessment criteria.

#	Question
Q1	Are the aims of the study clearly stated?
Q2	Are the scope and context and experimental design of the study clearly defined?
Q3	Are the variables in the study likely to be valid and reliable?
Q4	Is the research process documented adequately?
Q5	Are all the study questions answered?
Q6	Are the negative findings presented?
Q7	Are the main findings stated clearly (regarding creditability, validity, and reliability)?
Q8	Do the conclusions relate to the aim of the purpose of the study, and are they reliable?

ensure the inclusion of as many relevant primary studies as possible. We opted for forward snowballing rather than backward snowballing since their efficiencies are similar, with forward snowballing finding slightly lower number of non-relevant papers (Badampudi et al., 2015), and not both to manage the manual effort needed. We checked the citations listed on Google Scholar to each primary study found in the first two steps, i.e., database and manual search, against the inclusion and exclusion criteria. Forward snowballing provided additional 36 primary studies.

In the last step, the authors conducted a quality assessment for the primary studies assigned to them before extracting data, as proposed in the literature (Hassler et al., 2014). Each primary study was assessed by one of the authors. Table 4 lists the criteria used for quality assessment. We derived these criteria from Kitchenham et al. (2009) and our earlier SLRs, such as Catal et al. (2021). We scored each paper using a 3-point Likert scale (yes = 1, somewhat = 0.5, no = 0) for each criterion. For instance, we scored for Q1 as 1 if the aim of the study was stated clearly in the introduction (expected place); as 0.5 if the aim was vaguely stated, or not at the expected place, and as 0 if the aim was not stated in the paper. We decided to include the papers with a score higher than four points to maintain a high-quality input of primary studies. We excluded 14 studies (listed in Appendix A.2: Ahmed et al., 2021, Al Qasem and Akour, 2019, Alazawi and Salam, 2021, Ardimento et al., 2020, Chaubey and Arora, 2020, Kumar and Singh, 2021, Lin et al., 2020, Liu et al., 2021b, Manjula and Florence, 2018, Thapa et al., 2020, Wongpheng and Visutsak, 2020, Zhang and Wu, 2020, Zheng et al., 2019, Zheng et al., 2021) with a score under our threshold.

3.3. Data extraction

After primary study selection, we started with the data extraction phase. We formed an initial data extraction form (Table 5) based on our RQs. The first six rows constitute the metadata of the papers. The first author formed an initial list of categories using previous SLRs (Catal et al., 2021) and conducted a pilot data extraction on a few randomly selected primary studies. Afterwards each author extracted data from the primary studies assigned to him. Whenever an author was undecided about the data to be extracted, he recorded that case, and these cases were resolved via discussions among the authors. During data extraction phase, we continuously refined the categories iteratively and incrementally during data extraction. We recorded the reported challenges and proposed solutions as free text for further analysis and synthesis.

3.4. Data synthesis and reporting

We extracted quantitative data using categories for the RQs between one and nine. Thus, we reported the frequencies and percentages of each identified category to answer these RQs.

The only RQ that required qualitative analysis was RQ10, i.e., the challenges and proposed solutions. 50 out of 102 primary studies reported one or more challenges and some of them proposed solutions to cope with these challenges. We recorded the challenges and the proposed solutions in Google sheets during data extraction. We conducted open coding (Miles et al., 2019) to analyze the challenges. A code symbolically assigns a summative or evocative attribute for a portion of qualitative data (Miles et al., 2019). We performed open coding in cycles. In the first cycle, we identified any emerging patterns of similarity or contradiction. In

Table 5
Data extraction form.

Field	Input type/Categories	Relevant RQ
Paper ID	Auto incremented number	–
Paper title	Free text	–
Abstract	Free text	–
Keywords	Free text	–
Publication year	Number	Demographics
Venue/Journal/Conference	Free text	Demographics
SDP scenario	Multiple selection	RQ1
ML category	Multiple selection	RQ2
Dataset	Multiple selection	RQ3
Representation of source code	Multiple selection	RQ4
Granularity level of prediction	Multiple selection	RQ5
Dealing with class imbalance problem	Multiple selection	RQ6
DL algorithms used	Multiple selection	RQ7
Evaluation metrics	Multiple selection	RQ8
Validation approach	Multiple selection	RQ8
Reproducibility package	Multiple selection	RQ9
Challenges and proposed solutions	Free text	RQ10

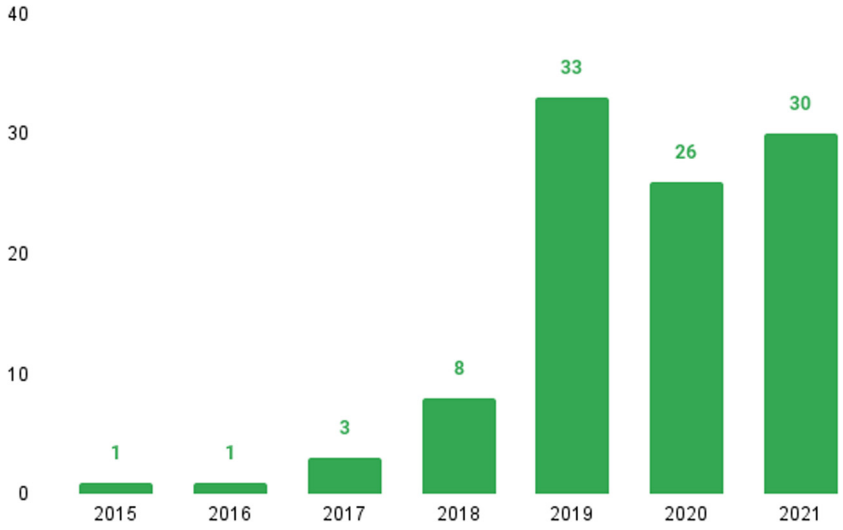

Fig. 4. Number of primary studies over the years.

the second cycle, we collapsed and expanded to understand any patterns. After extracting the main themes and codes, we revised the codes and assigned them to each challenge.

4. Results

This section presents the responses to the RQs defined at the beginning of this research study. Before presenting the responses, we provide additional information about the identified primary studies, e.g., the yearly distribution and distribution of the studies per venue. As Fig. 4 shows, there is an increasing trend in the number of primary studies. This indicates that the application of DL algorithms for SDP is a recent trend among researchers, especially ongoing as of 2019. 54% of the studies (55 primary studies) were published in journals. The journals with the most papers are IEEE Access (10 papers) and IET Software (five papers). 46% of the studies (47 primary studies) were presented in conferences and workshops. The conference with the most papers (four papers) is the International Conference on Software Quality, Reliability and Security (QRS). Appendix A.2 includes the list of the venues in which the primary studies were presented and published.

4.1. SDP scenarios

Fig. 5 shows the number of studies per SDP scenario. While 82 studies include WPDP, 42 studies involve CPDP. 22 studies encompass experiments for both WPDP and CPDP.

Ten studies in total focus on cross-version defect prediction. Six of these are classified under WPDP scenario. In one of these studies, Zhang et al. (2021a) conducted experiments on total 32 cross-version pairs derived from 45 versions of 13 software projects obtained from PROMISE (Jureczko and Madeyski, 2010), NASA (Shepperd et al., 2013), and SOFTLAB (Turhan et al., 2009) repositories. For instance, in three of the experiments, they trained a model using Ant versions 1.3, 1.4, and 1.5 and tried to predict bugs in Ant versions 1.4, 1.5, and 1.6, respectively (Zhang et al., 2021a). Li et al. (2019c) examined the ability of a model trained on all the existing versions of a project X and other projects to detect bugs on an unseen version of the project X, i.e., CPDP. Shi et al. (2021) conducted experiments on cross-version defect prediction in both WPDP and CPDP settings. For instance, they built a model using Camel 1.4 and tried to predict bugs for Camel 1.6 (WPDP) and Jedit 4.1 (CPDP).

Young et al. (2018) conducted experiments on just-in-time defect prediction, a.k.a. change level defect prediction, using six open-source projects. For each project, they built models via a training set obtained from that project and tried to predict defect-prone changes for the same project. Xu et al. (2021b) emphasized the difficulty of collecting sufficient labeled bug data for some mobile applications. Hence, they proposed to learn a high-level feature representation from a bug dataset consisting of 19 mobile applications for JIT defect prediction. Zeng et al. (2021) used DL

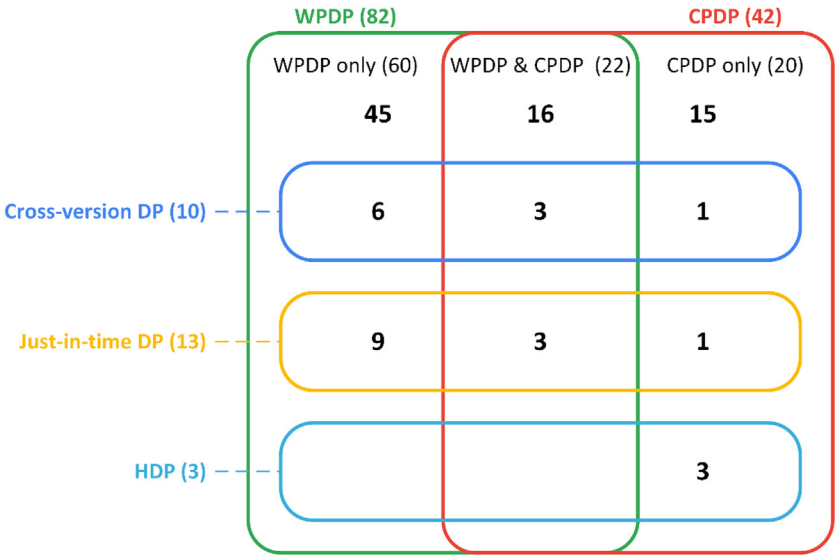

Fig. 5. Number of primary studies per SDP scenarios.

to build models for identifying defective commits in both WPDP and CPDP settings.

Three studies addressed HDP. Gong et al. (2019) designed a neural network to deal with heterogeneous metric sets for defect prediction. Sun et al. (2021) proposed a deep adversarial learning based HDP method. Wu et al. (2021) proposed a method for multi-source heterogeneous cross-project defect prediction. They used an autoencoder to extract the intermediate features from the original datasets instead of simply removing redundant and unrelated features (Wu et al., 2021).

Researchers started to show interest in cross-version defect prediction using DL in last three years. Our paper pool includes two studies published in 2019 and four in 2020 and 2021. The first study on JIT defect prediction with DL dates back 2015. One study published in 2018 and the rest 11 studies were published in last three years. One study on HDP was published in 2019 and the other two in 2021.

4.2. ML categories

The second RQ is related to the ML categories (i.e., supervised learning, unsupervised learning, and semi-supervised learning). As Fig. 6 shows, 94% of the primary studies (96 studies) apply supervised DL. 77 of these do not include any other ML category. 23 studies involve unsupervised DL learning. While three of them do not include any other ML category, 19 of them involve supervised and one of them involves semi-supervised learning.

Shi et al. (2021) conducted experiments using unsupervised and semi-supervised learning. Two studies, i.e., Sun et al. (2020a) and Xu et al. (2021b) involve only semi-supervised learning.

All the studies, except one (Wang et al., 2016), published before 2018 includes only supervised learning. The first three studies involving unsupervised learning were published in 2018 (Bhandari and Gupta, 2018; Sun et al., 2018; Tong et al., 2018). 10 of 23 studies including unsupervised learning were published in 2019. Afterwards, we see a downward trend in the use of unsupervised learning, four study in 2020 and five in 2021. Semi-supervised learning for SDP with DL started to be used after 2020 (Sun et al., 2020a; Xu et al., 2021b; Shi et al., 2021).

4.3. Training and testing datasets

To answer this RQ, we extracted the datasets used in each primary study. Researchers used more than eight datasets on

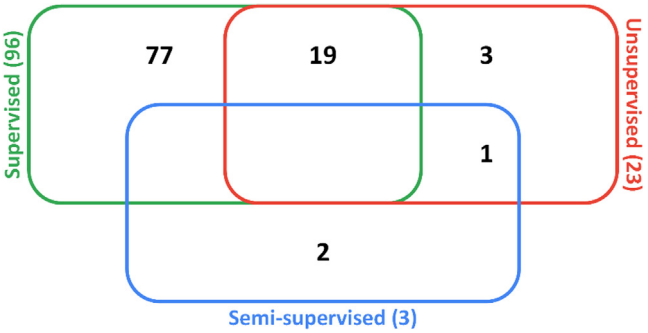

Fig. 6. Number of primary studies per ML category.

average per study. Table 6 lists the datasets that were used in ten or more primary studies. All these datasets were developed in one of three programming languages, i.e., Java, C, and C++. The top 11 most frequently used datasets are from PROMISE (Jureczko and Madeyski, 2010) repository and were developed using Java.

Some studies, e.g., Zeng et al. (2021), conducted experiments on the projects developed via C++ (QT and OpenStack) and Java (JDT, Platform, and Gerrit) as well as the projects like Go developed with a modern programming language, like Golang. Dong et al. (2018) focused on predicting bugs in Android binary executables called "apk"s. They obtained Android projects, such as Wikipedia and Chess apps, from GitHub and constructed datasets to build models for defect prediction. Xu et al. (2021b) and Zhao et al. (2021a,b) also used Android projects to build and test bug prediction models.

Ferenc et al. (2020) used a public unified bug dataset for Java (Ferenc et al., 2018), which is an amalgamation of three repositories, i.e., PROMISE (Jureczko and Madeyski, 2010), the Bug Prediction Dataset (D'Ambros et al., 2010), and the GitHub Bug Dataset (Tóth et al., 2016). Xu et al. (2021a) and Zhang et al. (2020) crawled GitHub and Codeforces, respectively to build datasets. Only one study, i.e., Albahli (2019), included commercial projects in their experiments. They tried to build models and predict bugs using six open source and five commercial projects (Albahli, 2019).

Table 6
The datasets used in ten or more primary studies.

Project	# of primary studies	Programming language	Description
Camel	55	Java	Enterprise integration framework
Xalan	53	Java	A library for transforming XML files
Xerces	49	Java	XML parser
Poi	48	Java	Java library to access Microsoft format files
Log4j	44	Java	Logging library for Java
Lucene	44	Java	Text search engine library
Synapse	44	Java	Data transport adapters
Jedit	43	Java	Text editor designed for programmers
Ant	39	Java	Java based build tool
Ivy	32	Java	Dependency management library
Velocity	22	Java	A Java-based template engine
Eclipse JDT	18	Java	Eclipse Java Development Tools
PC1	18	C	A flight software for earth orbiting satellite
CM1	16	C	A NASA spacecraft instrument
KC1	16	C++	A system implementing storage management for receiving and processing ground data
PC3	16	C	NASA orbiting project
PC4	16	C	NASA satellites project
MW1	15	C	Zero gravity experiment
JM1	14	C	A real-time predictive ground system
KC2	13	C++	Data from C++ functions. Science data processing; another part of the same project as KC1; different personnel than KC1. Shared some third-party software libraries with KC1, but no other software overlap.
MC1	13	C & C++	NASA combustion project
PC2	11	C	NASA for earth project
MC2	10	C	One of the NASA Metrics Data Program defect data sets

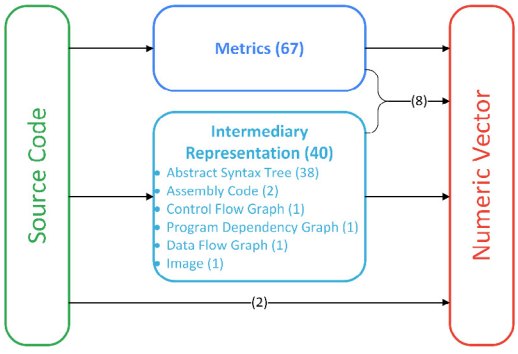

Fig. 7. Source code representation approaches used in the primary studies.

4.4. Representation of source code

While source code comprises of textual data, DL algorithms work on numeric vectors (Gousios, 2021). Source code representation refers to converting source code to a form that can be processed by DL algorithms. This conversion process should consider and optimize the loss of information during conversion. Fig. 7 shows source code representation approaches used in the primary studies in our pool. Different kinds of metrics, i.e., software size and structure, process, and product metrics, are the most frequently used representation technique for SDP. 40 studies used an intermediary representation (some of them involve more than one), i.e., Abstract Syntax Tree (AST), assembly code, Control Flow Graph, Program Dependency Graph, Data Flow Graph, and image, to form a numeric vector to be fed to a DL algorithm. Eight studies combined metrics with an intermediary representation. Two studies converted source code directly to a numeric vector.

67 studies used a set of metrics to represent source code. Some of the studies (such as Xu et al., 2019) used a tool like CKJM (Spinellis, 2005) to extract size and structure software metrics by processing the bytecode of compiled Java files. These metrics include weighted methods per class, coupling between object classes and McCabe's cyclomatic complexity. Process features or change features obtained from the change history of a software project are also an indicator for defect prediction (Rahman and Devanbu, 2013). For instance, Yang et al. (2015) utilized change metrics, such as number of modified files, number of developers that changed the modified files, lines of codes added and deleted for JIT defect prediction. Ardimento et al. (2021) used some metrics, such as commit frequency, developer seniority, owned commit, mean time between commits, to represent the development process. Product metrics, describing the source code internal structure quality, are another type of metrics used for defect prediction. Such metrics include number of attributes inherited, depth of inheritance tree, number of methods, and number of static methods. Ardimento et al. (2021) used product metrics along with process metrics. Some researchers, such as Tong et al. (2018) and Zhao et al. (2019), normalized the values of metrics before forming a numeric vector.

Abstract Syntax Tree (AST) is a tree representation of the abstract syntactic structure of source code (Mou et al., 2016). 38 studies involved AST as an intermediary representation to construct a numeric vector. Liang et al. (2019) converted source code to AST and extracted tokens from AST nodes to generate token sequences. These sequences were mapped to fixed-length vectors and a Continuous Bag of Words (CBOW) model was built using all datasets to be fed to an LSTM network. Chen et al. (2019) used a simplified version of AST, by considering node types, which are project-independent, and ignoring method and variable names, which are project-specific. Some researchers, such as Li et al. (2017), Dam et al. (2019), and Liu et al. (2020), used word embeddings to obtain numeric vectors from ASTs. Shi et al. (2020) built embedding vectors using an AST path pair-based source code representation method named PathPair2Vec.

Li et al. (2019b) modeled and analyzed the relations among paths of ASTs from different methods using Program Dependency Graph (PDG) and Data Flow Graph (DFG). While the local context of buggy code is represented by buggy paths in AST, the global context of buggy code is represented by the relations among

buggy methods modeled via program and data flow dependencies (Li et al., 2019b).

Phan and Nguyen (2017) preferred assembly instruction sequences over ASTs since they may simulate program behavior better due to its closeness to machine code and reflect program structure. Phan et al. (2017) leveraged control flow graphs constructed from assembly instructions obtained by compiling source code.

Chen et al. (2020) proposed source code visualization for SDP, in other words, they represented source code as images and trained image classification models that predict defects. Each source file was converted into a vector of 8-bit unsigned integers corresponding to the ASCII decimal values of the characters in the source code. Then, an image is generated from that vector to be fed to ImageNet's pre-trained AlexNet model for classification (Chen et al., 2020).

Eight of the studies combined an AST-based input with a set of metrics. Fan et al. (2019b), Li et al. (2017), Lin and Lu (2021), Qiu et al. (2019b), Shi et al. (2021) and Wang et al. (2021) combined word embeddings of ASTs with metrics. Fiore et al. (2021) obtained a vector from the nodes of AST. Afterwards, they combined these vectors with the vectors involving metrics. Huo et al. (2018) extracted five types of metrics (authorship, change type, change interval, code churn, and co-change) by analyzing change logs and textual contents generated by version control systems. In addition, they analyzed the differences between ASTs and identified change semantic types, like insertion of an expression statement or change of an infix expression. Afterwards, they built change sequences out of these metrics and semantic information of changes and used them for training DL models. For instance, a sequence for authorship can be represented as <developer1, developer2, developer1, developer3, developer2>.

Two studies did not use any intermediary representation and converted source code directly into numeric vector. Hoang et al. (2019) parsed commit messages and code changes using NLTK (Loper and Bird, 2002) and represented each word in the commit messages and code changes as a n-dimensional vector. Tian and Tian (2020) converted source code into fixed length vectors using Word2vec (Mikolov et al., 2013).

4.5. Granularity level of prediction

ML/DL models were constructed to predict defects at various levels of granularity (Nam, 2014), i.e., file, module, change, class, function, procedure, and statement. Previous research found out that level of granularity has an impact both on model prediction performance and effort required to localize defects (Koru and Liu, 2005; Calikli et al., 2009). 37 studies involved a model predicting defects at file level and 32 studies at module level. 12 studies built models to identify buggy changes. Four studies included class-level predictions for a software system developed using an object-oriented programming language. More fine-grained levels of granularity level predictions, i.e., function/procedure and statement/line level, were addressed by three primary studies each. 16 studies did not report any granularity level of prediction (see Fig. 8).

In line with the observation of Kamei and Shihab (2016), researchers started to show more interest in using DL for SDP at more fine-grained levels. Since 2019, nine studies (two in 2019, four in 2020, three in 2021) reported the results of their experiments on class, statement, and procedure-level predictions. In addition, starting with one study in 2018, researchers published papers on change-level defect prediction using DL algorithms. 11 studies followed these in the last three years, i.e., three papers in 2020 and four papers in 2019 and 2021.

Three studies (Zhou et al., 2019; Wang et al., 2020; Zhu et al., 2021a) reported experiment results at more than one granularity level. Wang et al. (2020) deployed a DBN to learn semantic features from ASTs for file-level defect prediction models and source code changes for change-level defect prediction models automatically. DBN-based semantic features helped in improving prediction performance by varying percentages (from 2.9% to 13.3%) at file and change levels in WPDP and CPDP scenarios (Wang et al., 2020). Zhou et al. (2019) used different datasets to train and test DL models for file-, module-, and class-level defect prediction. Zhu et al. (2021a) used five datasets in their experiments, i.e., PROMISE (Jureczko and Madeyski, 2010) and AEEEM (D'Ambros et al., 2012) for class-level, NASA (Shepperd et al., 2013) and SOFTLAB (Turhan et al., 2009) at function-level, and ReLink (Wu et al., 2011) at file-level defect prediction.

4.6. Techniques for dealing with the class imbalance problem

Class imbalance problem arises when there is a severe skew in the class distribution in a dataset (Brownlee, 2020). In SDP, datasets generally involve much fewer defective samples compared to non-defective ones. Having this bias, i.e., defective samples as the minority class, influences the prediction performance of DL models, sometimes leading to ignore the minority class entirely. This is a serious problem since it is important to predict defective instances. There are techniques to address class imbalance problem at data-level and algorithm level (Tong et al., 2018).

Fig. 9 shows the frequencies of the techniques used by researchers to address class imbalance problem. Since six studies involved more than one technique, the total number of data points in the figure is more than 102. 63 of the studies used a technique at the data-level. They applied a kind of over-sampling or under-sampling or tried to create data to balance minority and majority classes. Nine studies tried to cope with the imbalanced datasets at the algorithm-level using either cost-sensitive or ensemble learning techniques. 42 of the studies did not report any technique to address class imbalance problem.

The most frequently used technique used at the data-level is oversampling (42 studies). Oversampling techniques duplicate or create new synthetic instances in the minority class (Brownlee, 2020), i.e., creating new defective samples. 21 studies used Synthetic Minority Oversampling TEchnique, or SMOTE for short (Chawla et al., 2002) for oversampling. Some studies (Eivazpour and Keyvanpour, 2019; Xu et al., 2019) used ADAptive SYNthetic (ADASYN) sampling, which is an extension of SMOTE. 20 studies applied a kind of random oversampling. One study, i.e., Yedida and Menzies (2021), proposed a fuzzy sampling technique, which is a variation of oversampling. Eight studies used other techniques to create new instances of defective samples. Xu et al. (2021a) identified 3026 bug fixes in 307 Java projects on GitHub. They constructed a dataset by combining the defective and fixed versions of the source files of these 3026 bug fixes and ended up with a balanced dataset. Bhandari and Gupta (2020) increased the number of defective instances by injecting defects into the source files.

Zhang et al. (2021a) leveraged WGAN-GP (Wasserstein GAN with Gradient Penalty) to generate more defective training instances. Similarly, Sun et al. (2018) and Eivazpour and Keyvanpour (2019) utilized Variational Autoencoder (VAE) and Zhu et al. (2021a) used GAN (Generative Adversarial Networks) to generate defective instances. Sun et al. (2020b) used both VAE and GAN along with SMOTE in their experiments.

Nine studies included algorithm-level techniques to address class imbalance problem. Six of them utilized cost-sensitive techniques. Li et al. (2019a) and Xu et al. (2019) proposed to assign different misclassification costs to the different classes in the model building stage to learn defective instances better. Similarly,

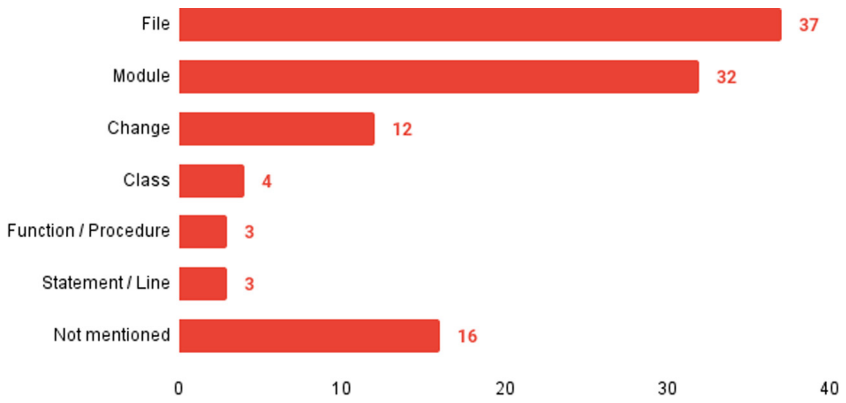

Fig. 8. Granularity levels of prediction reported in the primary studies.

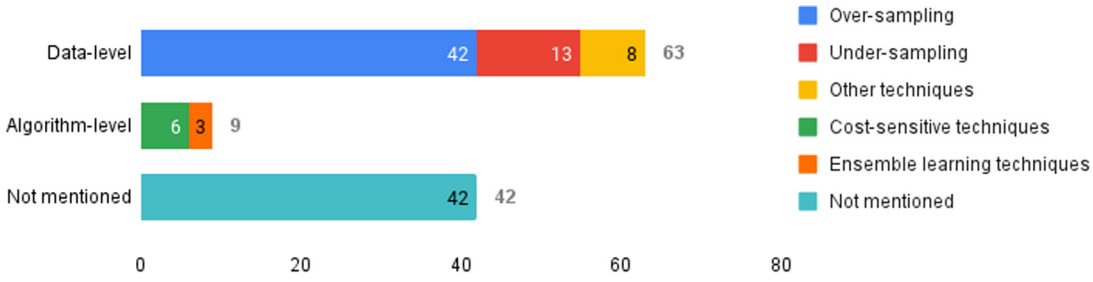

Fig. 9. Distribution of the techniques used for coping with class imbalance problem.

Hoang et al. (2019) imposed a higher cost on misclassification of minority class, i.e., buggy commits, than they do on misclassification of the majority class, i.e., non-buggy commits to increase the performance of their JIT defect prediction model. Zhao et al. (2019) introduced a cost-sensitive cross-entropy loss function into DNN for JIT defect prediction in Android applications. Thus, they considered the prior probability of minority and majority classes, i.e., defective and clean commits respectively, when calculating cross-entropy loss to compensate class imbalance. Gong et al. (2019) assigned different misclassification costs to defective and non-defective instances to increase the performance of the neural network for HDP.

The authors of the three of the nine studies preferred an ensemble learning technique. Tran et al. (2019) used a two-stage ensemble learning method in the training stage. Tong et al. (2018) also reported that they used two-stage ensemble learning to cope with imbalanced datasets as well as eliminating overfitting problem. Xu et al. (2019) included Bagging, Balanced Bagging, AdaBoost, RUS with AdaBoost, EasyEnsemble, and BalanceCascade in their experiments.

Six studies out of 102 used more than one technique to cope with class imbalance problem and reported their findings. Yedida and Menzies (2021) found out that oversampling is effective and necessary to applying DL for SDP. They applied oversampling, SMOTE, and fuzzy sampling on 24 datasets. Based on the F1 scores obtained from 240 data points (10 repeats over the 24 datasets), most significant improvements to DL model performance came from fuzzy sampling approach. Eivazpour and Keyvanpour (2019) applied various oversampling techniques, such as SMOTE, ADASYN, Borderline-SMOTE, VAE, to ten imbalanced datasets. Based on the results, the generation of synthetic samples using VAE yielded better performance. Sun et al. (2020b) reported that VAE performed better than GAN and SMOTE; GAN had better performance compared to SMOTE on some of the datasets. Zhang et al. (2020) compared random undersampling with Self Organizing Maps (SOM) clustering based undersampling (Vannucci and Colla, 2018). Vannucci and Colla (2018) cluster rare and frequent samples in datasets remove frequent samples to have a more balanced dataset. Zhang et al. (2020) proposed to use SOM clustering-based undersampling instead of random undersampling. Xu et al. (2019) compared sampling, cost-sensitive, and ensemble learning techniques and found our that cost-sensitive techniques are more effective in improving DL model performance. The experiments conducted by Gong et al. (2019) favored the use of cost-sensitive learning technique over SMOTE for HDP.

4.7. DL approaches

As seen in Fig. 10, the most frequently used DL algorithm is CNN. The other widely used algorithms are RNN/LSTM/GRU, MLP, and DBN. 13 studies used encode-decoder architecture most of which specifically an Autoencoder.

Like we can observe in some other domains, such as malware detection (Catal et al., 2021) and phishing detection (Catal et al., 2022), CNN, RNN/LSTM/GRU, and MLP are the top three most frequently used DL approaches. The overall reason may be that these algorithms performed well in many tasks, and they are well-known among researchers and practitioners; this fact is indeed mentioned in numerous works covered in this study. CNNs, in particular, are reported to work well with high dimensional data and capture local patterns (Li et al., 2017; Pan et al., 2019). In turn, RNN/LSTM/GRU architectures can, for instance, capture long-distance dependencies and semantics (Wang et al., 2021; Liang et al., 2019). DBNs are used with regards to their ability to learn a representation for reconstructing the training data with a high probability (Wang et al., 2016). Finally, autoencoders help learning semantic information and eliminating noise (Zhang et al., 2021a,b).

Less frequently used DL algorithms include Generative Adversarial Networks (GAN), Hybrid DL Model, Hierarchical NN, Siamese NN, Extreme Learning Machines, and Graph NN. Various reasons have been reported for using such approaches, e.g., using

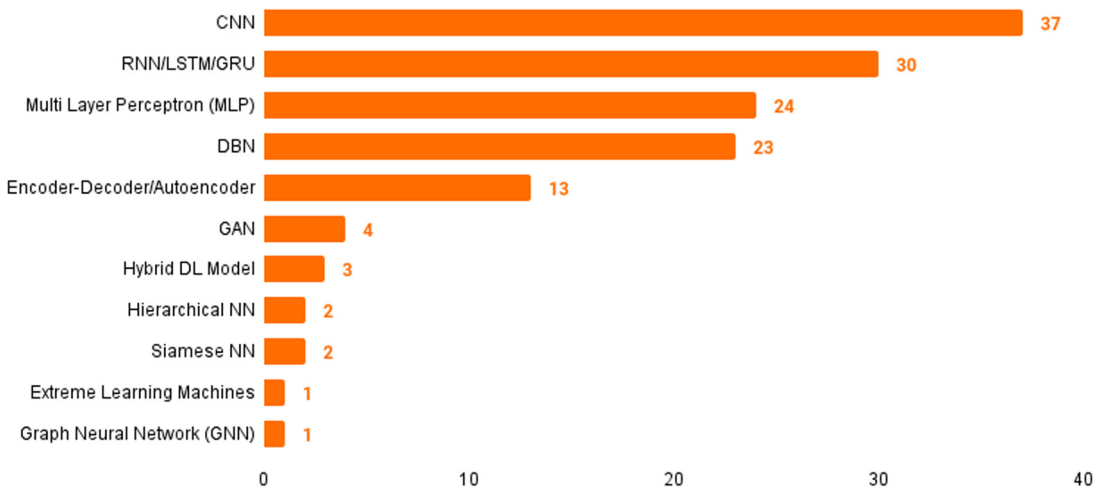

Fig. 10. Distribution of DL approaches.

GAN for creating synthetic data (Sun et al., 2020a,b), combining supervised and unsupervised techniques in a hybrid architecture (Albahli, 2019), using a hierarchical architecture to ensure full extraction of all the semantic features (Wang et al., 2021), using Siamese networks to work with limited data (Zhao et al., 2018), and capturing source code ASTs in graph neural networks (Xu et al., 2021a,b).

4.8. Evaluation metrics and validation approaches

Researchers evaluated their defect prediction models using different evaluation metrics and validation approaches. Fig. 11 presents the distribution of the top 10 most frequently used evaluation metrics based on our paper pool. 68% of the studies (69) used "F-measure" for evaluation. 48 studies used "Recall", also known as "True Positive Rate" or "Sensitivity". Recall refers to the fraction of the successfully predicted defects. "Area Under the Curve (AUC)" measures a classifier's ability to distinguish between classes. When AUC approaches to one, this means that the prediction model can distinguish positive and negative classes, i.e., buggy and non-buggy in our case. A predictor with an AUC value close to zero tend to classify buggy cases as non-buggy and vice-versa. 40 studies in our paper pool used AUC as an evaluation metric. 40 studies used "Precision", also known as "Positive Predictive Value". Precision refers to the number of correctly predicted defects divided by the number of predictions. The "Accuracy" evaluation metric was used in 28 studies. This metric is easy to understand and easily suits for binary and multi-class classification problems. On the other hand, accuracy metric works well when there is no class imbalance. A defect prediction dataset is qualified as imbalanced when the number of buggy samples much lower than the number of non-buggy ones and vice versa. 20 studies used "Matthews Correlation Coefficient (MCC)" as an effective solution overcoming the class imbalance issue. MCC produces a high score only if the prediction obtained good results in all the four confusion matrix categories (true positives, false negatives, true negatives, and false positives).

Effort-aware defect prediction models consider the differences in the cost of applying quality assurance activities for each piece of code (file, function, etc.) (Mende and Koschke, 2010). Although it may be hard to quantify effort, researchers proposed effort-aware performance measures. 13 studies in our paper pool used an effort-aware performance measure. Six studies, i.e., Albahli (2019), Chen et al. (2019), Qiao and Wang (2019), Qiu et al. (2019c), Wang et al. (2021), and Yedida and Menzies (2021), used "P_{opt}" evaluation metric. A larger P_{opt} value refers to a smaller difference between the optimal and predicted models and thus better performing model (Bennin et al., 2016). Four studies, i.e., Sheng et al. (2020), Wang et al. (2020), Zhu et al. (2020), and Wang et al. (2021), used "PofB20" (Jiang et al., 2013) to measure the percentage of defects that a developer can identify by inspecting the top 20% lines of code. Four studies, i.e., Qiao and Wang (2019), Xu et al. (2019, 2021b), and Zhao et al. (2021b), Effort-Aware recall (EARecall), which is defined as the percent of reviewed defective commit instances to the whole defective commit instances. Three studies, i.e., Xu et al. (2019, 2021b), and Zhao et al. (2021b), Effort-Aware F-measure (EAF-measure), which is defined as the weighted harmonic average between EARecall and EAPrecision. Xu et al. (2019) used EAPrecision in addition to EARecall and EAF-measure. Zhao et al. (2019) used "Normalized Expected Cost of Misclassification (NECM)" proposed by Khoshgoftaar and Seliya (2004) to handle the different misclassification costs. To cope with the imbalanced datasets, they used the reciprocal of NECM to strengthen the punishment for the weight of the majority class and reduce the suppression of the weight of the minority class (Zhao et al., 2019). Wang et al. (2021) used IFA, which counts the number of initial false alarms encountered before the first defect is found (Majumder et al., 2022). Since developers will ignore the suggestions if too many false alarms are offered before reporting a defect. Therefore, smaller IFA values are preferred.

Nine studies used "G-measure", which is a trade-off measure that balances "Possibility of detection (PD)" and "Possibility of false alarm (PF)" (Yu et al., 2018). Higher PD and lower PF, thus higher G-measure denote a better prediction model (Yu et al., 2018). Six studies used "ROC curve", also known as "Receiver Operating Characteristics Curve". ROC curve compares the True Positive Rate and the False Positive Rate and helps to determine the trade of between these two characteristics.

Four studies used "Mean Squared Error (MSE)", which measures the average of the squares of the errors. Another set of four studies used "True Negative Rate (TNR)/Specificity", which refers to the ratio of genuinely negative cases that is predicted as negative by model.

The researchers of 60 studies reported a hold-out approach in which the dataset is split into training and test datasets. Another set of 43 studies preferred cross-validation as a validation approach. Only one study, i.e., Turabieh et al. (2019), reported that both approaches were applied for validation.

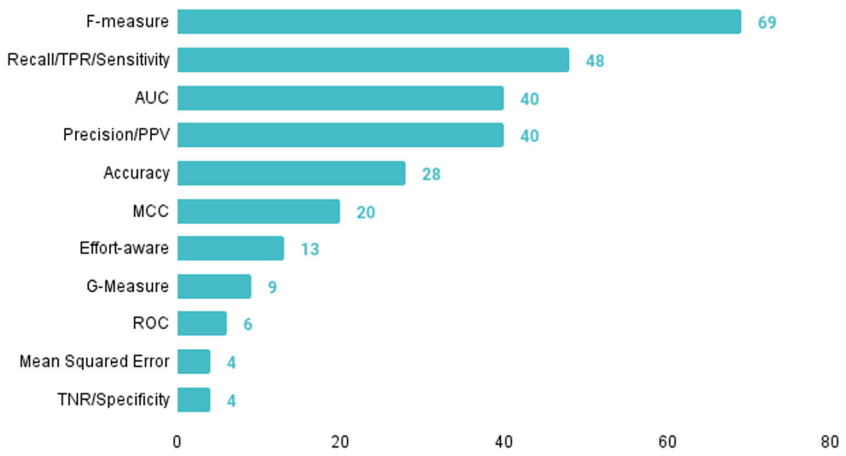

Fig. 11. Distribution of evaluation parameters.

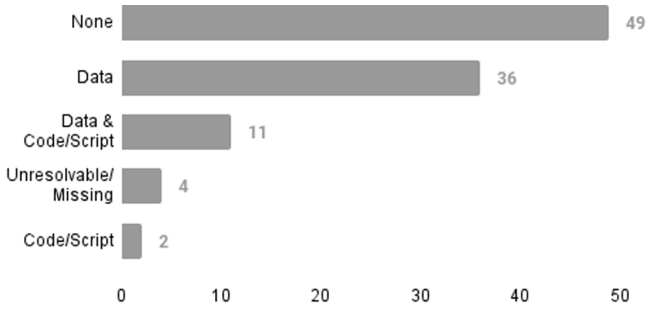

Fig. 12. Presence of reproducibility packages in the primary studies.

4.9. Reproducibility package

The reproducibility of a study is one of the essential characteristics of scientific studies (González-Barahona and Robles, 2012). To be qualified as a reproducible scientific study, the reported experimental results of a study should be obtained by other researchers using authors' artifacts (i.e., source code and datasets) with the same experimental setup (Liu et al., 2021b). Some researchers pointed out the reproducibility issues in SE (Lewowski and Madeyski, 2022). Recently Liu et al. (2021a) analyzed some studies on the use of DL models in solving a SE problem, like defect prediction or code clone detection. They reported that more than half of the studies do not share high-quality source code or complete data to support the reproducibility of their DL models. Thus, we examined whether the authors of primary studies on SDP using DL publish reproduction packages for their studies. We used the categories used by Lewowski and Madeyski (2022) during data extraction. Fig. 12 shows the results on the presence of a reproducibility package in the primary studies in our paper pool. 49 publications (48% of the primary studies) do not mention any sort of package to reproduce the experimental results. Four studies claim to provide a reproducibility package; however, the link in the study is either missing or unresolvable. 36 studies contain only data, and two studies involve only scripts for reproduction. 11 studies (11% of the primary studies) include a reproducibility package including data and scripts.

In addition, we do not observe an upward trend over the years in sharing a reproducibility package. Out of three and eight studies, only one shared a reproducibility package in 2017 and 2018. In the last three years, out of 33, 26, and 30 studies, only three shared a package for reproduction.

4.10. Challenges and proposed solutions

In this section, we summarize the challenges and proposed solutions we extracted from 50 primary studies. The rest of the studies do not mention a challenge related to the use of DL for SDP; besides the common challenges we already address explicitly in our research questions (e.g., class imbalance addressed in RQ6). In our qualitative analysis, we mapped the challenges to three categories derived from the ML model life cycle (as introduced in Fig. 2):

1. Data Engineering,
2. Model Development, and
3. General (i.e., related to the whole life cycle)

Within each category, we further classified the challenges into subcategories. For each subcategory, we described the challenge, and summarized the solutions offered by the primary studies in the scope of our review. We believe this elaborate analysis, which has not been done with this level of detail in any of the related work (see Section 2.3), can provide researchers with directions to focus their efforts in the coming years.

A short summary of the challenges and their solutions is as follows, while the extensive list can be found in Appendix A.3.

4.10.1. Data engineering

Heterogeneous data: The fact that a large variety of different projects, versions and features is used in SDP leads to highly heterogeneous data, in particular when using different source and target for prediction. Such data degrade the performance of the classifier (Albahli, 2019; Gong et al., 2019; Li et al., 2019a; Qiu et al., 2019a,b; Sheng et al., 2020; Wang and Lu, 2020; Sun et al., 2020a; Huang et al., 2021; Sun et al., 2021; Wu et al., 2021). Some researchers have tackled this challenge using different DL architectures which take this difference into account, while others have introduced normalization and transformation steps in data preprocessing as well as in feature extraction.

Insufficient training data: Having limited training data, either from a quality or quantity point of view, makes it difficult to perform SDP in the first place (Zhao et al., 2018, 2019; Pandey and Tripathi, 2021). Potential solutions include using DL architectures capable of learning with limited data and adding more labeled data into the training dataset.

Training data formation: The training data can contain invalid instances of defects, e.g., incorrectly reported bugs (Li et al., 2019b). A solution to this involves manual validation of the training data, however this has the disadvantage of introducing manual bias in the validation process.

Incomplete code snippets: It is a challenge working around incomplete code snippets in change-level SDP, particularly when the approach relies on ASTs of the code (Wang et al., 2020). Heuristic approaches can be used for extracting relevant information from incomplete code.

AST node granularity and distribution: Different granularity and distribution for the ASTs used in CPDP might affect SDP negatively (Deng et al., 2020b). A particular multi-kernel transfer CNN, which considers these factors can be used for tackling the challenge.

4.10.2. Model development

Semantic features: Traditional features such as source code metrics are not sensitive to the semantics of programs, and can lead to poor prediction performance (Wang et al., 2016; Li et al., 2017; Huo et al., 2018; Cai et al., 2019; Chen et al., 2019; Dam et al., 2019; Fan et al., 2019a,b; Liang et al., 2019; Qiu et al., 2019c; Xu et al., 2019; Wang et al., 2020; Zhang et al., 2020; Bahaweres et al., 2021; Chatterjee et al., 2021; Xu et al., 2021a; Zhang et al., 2018, 2021a). A wide range of solutions have been offered to tackle this. These include using richer features ranging from code comments, embeddings, AST, and structural features, to feeding the source code itself to deep learners capable of capturing semantic information and using hybrid and ensemble techniques.

Feature redundancy: Highly correlating features may reduce prediction performance (Tran et al., 2019; Turabieh et al., 2019; Wei et al., 2019; Zhao et al., 2021b; Zhu et al., 2021b). Solutions include using deep learning architectures and ensemble techniques which can cope with this and using meta-heuristic approaches for feature selection.

Manual feature selection: Chen et al. point out the potential bias due to manual feature selection and rely on an image representation of source code from which image features are automatically selected by a self-attention mechanism (Chen et al., 2020).

Context-dependence: Different feature sets may provide the best prediction performance in different contexts, which might be overcome by using a particular LSTM architecture to optimize the combination of input features in each context (Wang et al., 2021).

Random initialization of parameters: Random selection of parameters of the learner may lower performance; meta-heuristic approaches can help compute the optimal values (Zhang et al., 2021b).

Fixed-length feature vectors: Conventional classification algorithms such as Naïve Bayes, assume the same length for all feature vectors, which can be tackled by using latent features and varying-length features (Wen et al., 2018).

Sequential networks: Sequential networks cannot capture the tree syntax and semantics of AST representations and their dependencies (Zhou and Lu, 2020; Yu et al., 2021a). Non-sequential networks such as bi-directional LSTM and HNN can solve the problem.

Hyperparameter sensitivity: Hyperparameters are very sensitive for DL models and different settings lead to very different performance results for SDP. Particularly given that not in all cases the hyperparameters are entirely reported, this is a serious challenge in transparency and reproducibility (Pan et al., 2019).

Model overfitting: Overfit models over the training data is a potential challenge and can be overcome using suitable techniques such as dropout regularization (Pandey and Tripathi, 2020).

Performance degradation: The performance of a prediction model may degrade, which can be avoided by updating the model via user feedback (Albahli, 2019).

4.10.3. General

Early SDP: Early defect prediction is a challenge; Manjula and Florence suggest combining DNN with generic algorithms for feature optimization (Manjula and Florence, 2019).

Granularity: Prediction of too coarse-grained levels of errors burden the developers for bug localization; therefore, keeping SDP at statement-level is a solution (Majd et al., 2020; Munir et al., 2021).

5. Discussion

In the following sub-sections, we discuss the results of our study. In Section 5.1, we provide a critical reflection on the results. Section 5.2, we discuss the threats to the validity of the present study and how we addressed them.

5.1. General discussion

We summarize the key findings to date in terms of current DL approaches and limitations in the SE literature. Additionally, we draw on the findings to provide key recommendations for future research in the application of DL for SDP.

There is a lack of consensus on the evaluation criteria for SDP. Several evaluation criteria have been proposed and each evaluation criteria works very well in specific scenarios. With DL being a subset of ML, it is not unusual that studies evaluate DL models with evaluation indicators used in ML. Popular indicators such as accuracy and precision have been considered as not very feasible in assessing defect prediction models because they are unstable for highly unbalanced datasets (Menzies et al., 2006; Joshi et al., 2001). Menzies et al. (2006) argued that recall and probability of false alarms (*pf*) are good indicators for determining the performance of a SDP model since defect prediction has a challenge, which is highly imbalanced datasets. Additionally, MCC has been recommended to be a better evaluation indicator for highly imbalanced datasets. Our analysis revealed that majority of the studies focused on using F-measure, which computes the harmonic mean between recall and precision and few considered indicators that include *pf*. Recall and AUC were the next most used indicators implying researchers consider the class imbalance issue when assessing the performance of deep learning models. However, additional indicators such as G-measure and MCC are amongst the least used. This could be due in part to the fact that most studies considered less than three evaluation indicators per study and usually used indicators that are easier to compute from a confusion matrix. Although few studies use more than three evaluation indicators in assessing their DL models, the feasibility, understanding and interpretability of these evaluation methods need clearer formalization and empirical studies under different testing scenarios.

We observe that although DL models improved prediction performance, the improvement was not significant when compared to traditional ML models. This may be due in part to the limited amount of data. Our analysis revealed that researchers used the same amount of data they used for conventional ML based defect prediction. Most software projects are not too large in terms of source code files but the data extracted from these projects could be large depending on the metric being used. Metrics computed on function level could generate more data instances compared to metrics computed on file level. Few studies (6%) computed metrics on function or statement level whereas 36% of the studies used file-based metrics. A potential approach to obtaining more metrics, data and improving prediction performance will be to leverage other sources of defect data and not only source code data. Heterogeneous data such as issue tracking documents, bug reports, requirement documents, and test reports among others

could be leveraged to produce a massive amount of data for DL models. Devanbu et al. (2020) acknowledge the importance of considering these kind of heterogeneous documents represented in various formats such as source codes, natural language documents, and graphical artifacts as software developers use documents in their daily workflow activities. Documents generated in these formats provide diversity and capture different orthogonal properties and information about the software system, which can inherently aid in improving the performance of deep learning models trained on them. They thus advocate for future practitioners to find several creative methods of combining these information sources for a richer dataset.

WPDP still seems to be the main defect scenario considered by researchers. This is not surprising as software projects are usually developed in versions making it easy to collect larger volumes of data for a single software project. Consequently, the data collected are similar and homogeneous, which makes WPDP models perform significantly better than CPDP models. Generally, defect datasets are not that much suitable for training a DL model but rather sufficient only for traditional ML models. CPDP models are trained on different software datasets, which are heterogeneous to the test dataset. However, CPDP is still gaining momentum as our analysis revealed that almost a third of the selected primary studies conducted CPDP. CPDP studies can leverage an advantage of DL, which is that DL models require a lot of data for significantly improved performance. CPDP studies can use several datasets from different software projects for model training, thus making it more appealing than WPDP.

A key advantage of using DL models is the ability to automatically extract features from the data thus eliminating the manual effort of extracting features from data. The review revealed that almost all researchers provide as input either software metrics or convert the source codes in ASTs to the DL models. Two problems with such data inputs are that manually extracted metrics cannot be fully exploited by DL models, such as CNN, which require mostly data in the form of images. Additionally, converting source code into AST implies an additional step of using a tool, which might be somewhat complicated especially for non-Java source code. In addition, the application of DL for SDP is more complex than the traditional ML approaches. Our analysis revealed the complicated techniques and approaches researchers used in extracting features from source codes. For traditional ML approaches, simple tools, which are mostly freely available written in C or Java automatically extract well known metrics from source code. The use of DL for defect prediction requires researchers to develop new tools to convert the source codes to other representations such as AST, control flow and images because proper tools either do not exist or are not widely available. The source code conversion and automatic feature extraction phase remains one of the main challenges stifling the fast progress of the adoption and use of DL for defect prediction. Software data is mostly source code and commit messages, which can be considered as being not very suitable for most DL models. Converting source codes into images looks promising and only one study (Chen et al., 2020) has been able to investigate that possibility. Their study aims to avoid the use of feature extraction tools thus eliminating the use of intermediary representations e.g., ASTs, and instead obtaining code semantic information directly. Thus, they designed a novel, color-based augmentation method to generate 6 color images from each source code file which has been converted into a vector of 8-bit unsigned integers corresponding to the ASCII decimal values of the characters in the source code. Their results were more promising and future research should focus on proposing and designing techniques of converting source code/commit messages into images, which encapsulates the source code information from humans but can be read and processed by DL models. This would also encourage private/commercial organizations to freely provide their source code since researchers would only obtain images without exposing their source code to the public thus ensuring data privacy.

Our analysis revealed that few studies provide replication packages for their experiments. Liu et al. (2021a) highlighted the importance of reproducibility and replication of DL studies for SE research. They noted that several studies do not provide their artifacts (source code and datasets) with the same experimental setup, which may be due in part to the complex nature of the experiments, several manual parameters and time-consuming optimization process, which is the opposite of the conventional ML models. Similar to their findings, we observe that very few (11%) provide their source code and dataset to support the reproducibility of their studies. The majority share their dataset, but this is because they all use already existing publicly available datasets. This finding suggests the urgent need to motivate researchers to make their artifacts publicly available. Additionally, researchers who make their complete reproduction packages publicly available should share their packages on well-established research data archives and not their own websites (Lewowski and Madeyski, 2022) since personal websites may not have working links after a period.

There is still little evidence on the practicability and adoption of defect prediction models in industry. Most of the studies used publicly available datasets (Section 4.3) and these models were not evaluated with commercial datasets. This may be due in part to the difficulty of obtaining commercial software data. The difficulty can be linked to data privacy, security, and complexity of using defect models, which hinders software quality practitioners in adopting defect prediction models. However, the major challenge to practitioners adopting defect prediction is the complexity of use and incompatibility with their personal or organizational environments. A survey conducted by Wan et al. (2018) with 395 practitioners showed that although most practitioners were willing to adopt defect prediction tools (more than 90%), the major challenge to practitioners adopting defect prediction were the lack of continuous integration support, lack of IDE integration and lack of code review tool integration. This indicates the complexity of using defect prediction models and incompatibility of defect prediction tools with their personal or organizational environments.

As discussed above (Section 2.1), the process of defect prediction is not very trivial especially regarding data collection and model construction. The use of DL should be targeted to making it easier and more adaptable for practitioners to use. Researchers should consider channeling the immersive power of DL to make defect prediction and detection easier for software quality teams. The popular conventional method of data extraction where source codes are converted to ASTs can be improved by rather converting them into images, which provides several advantages such as encapsulating the data thus contributing to data privacy and the ability to easily provide them to several DL models. Additionally, a systematic DL workflow and taxonomy should be provided to help practitioners comprehend the main difference between defect prediction using DL approaches and defect prediction using traditional ML approaches. The workflow can elaborate on the DL abstractions, process, procedures required for an improved prediction performance.

Few studies considered the application of data resampling approaches. This is a well-known challenge of software defect datasets (Song et al., 2021; Bennin et al., 2016). This is unsurprising as data resampling approaches are usually applied to traditional defect datasets, which are tabular in nature and thus much less complex to pre-process with data resampling. As we observed in Section 4.4, 37% of the studies represented their

Table A.1

ACM	The below query was executed using "edit query" feature on advanced search interface. *((Title: software) AND ((Title: fault) OR (Title: defect) OR (Title: quality) OR (Title: bug)) AND ((Title: predict*) OR (Title: estimat*)) AND (Title: "deep learning")) OR ((Abstract: software) AND ((Abstract: fault) OR (Abstract: defect) OR (Abstract: quality) OR (Abstract: bug)) AND ((Abstract: predict*) OR (Abstract: estimat*)) AND (Abstract: "deep learning")) OR ((Keyword: software) AND ((Keyword: fault) OR (Keyword: defect) OR (Keyword: quality) OR (Keyword: bug)) AND ((Keyword: predict*) OR (Keyword: estimat*)) AND (Keyword: "deep learning"))*
IEEE Xplore	The below query was executed using "command search" feature on advanced search interface. *((("Document Title":software) AND ("Document Title":fault OR "Document Title":defect OR "Document Title":bug OR "Document Title":quality) AND ("Document Title":predict* OR "Document Title":estimat*) AND ("Document Title":"deep learning")) OR (("Abstract":software) AND ("Abstract":fault OR "Abstract":defect OR "Abstract":bug OR "Abstract":quality) AND ("Abstract":predict* OR "Abstract":estimat*) AND ("Abstract":"deep learning")) OR (("Author Keywords":software) AND ("Author Keywords":fault OR "Author Keywords":defect OR "Author Keywords":bug OR "Author Keywords":quality) AND ("Author Keywords":predict* OR "Author Keywords":estimat*) AND ("Author Keywords":"deep learning")))*
ScienceDirect	The two queries below were executed using advanced search interface. The reason of using two separate queries was that the search feature did not allow the use of wildcard (*). The results were combined, and the duplicates were removed. Query 1: *(software) AND (fault OR defect OR bug OR quality) AND (predict OR prediction) AND ("deep learning")* Query 2*: (software) AND (fault OR defect OR bug OR quality) AND (estimate OR estimation) AND ("deep learning")*
Springer	The eight queries below were executed using search interface. The content types "book", "protocol", "reference work" were removed from the results. The results were combined, and the duplicates were removed. Query 1: *"software fault" estimate* "deep learning"* Query 2: *"software fault" predict* "deep learning"* Query 3: *"software defect" estimate* "deep learning"* Query 4: *"software defect" predict* "deep learning"* Query 5: *"software bug" estimate* "deep learning"* Query 6: *"software bug" predict* "deep learning"* Query 7: *"software quality" estimate* "deep learning"* Query 8: *"software quality" predict* "deep learning"*
Wiley	The below query was executed three times by selecting "Title", "Abstract", and "Keywords" as the search field on advanced search interface. The results were combined, and the duplicates were removed. *(software) AND (fault OR defect OR bug OR quality) AND (estimat* OR predict*) AND ("deep learning")*

source codes in an AST format, which may not be feasible for applying data resampling approaches. Nevertheless, the solution is not so far-fetched. Practitioners can leverage already existing data augmentation techniques, which are usually applied to DL datasets to solve class imbalance and aid in data generalization. However, these data augmentation techniques can only be applied to image datasets.

In summary, we provide some key recommendations to address the issues extracted from our findings and discussed above. These recommendations have already been highlighted above and they include:

- Development of new, more comprehensive DL approaches that automatically captures the needed features in sufficient detail and quality from source codes, bug reports and others
- Adoption of data augmentation techniques to tackle the class imbalance issue if the data is efficiently converted into images
- Identification of the key source code defect attributes that need to be captured sufficiently and to support defect prediction
- Publication of replication packages
- Consideration of other sources of data such as requirement documents, test documents, graphical artifacts among others in addition to the source codes.

5.2. Limitations and potential threats to validity

The scope of this study is limited to the following parameters:

- Date: This study covers primary studies published until the end of 2021, i.e., 31 December 2021.
- Type of Literature: This study comprises studies published in peer-reviewed journals and conference/workshop/symposium proceedings. Grey literature, e.g., papers only published in arxiv.org, blogs, videos, etc., was excluded from the paper pool.
- Perspective: The primary studies were selected using the inclusion criterion of applying at least one DL algorithm to SDP problem and reporting related empirical results. The studies involving only traditional ML algorithms and statistical techniques were excluded.

Some validity considerations are applicable for SLR studies (Petersen et al., 2008, 2015). The threats to the validity of this study are mainly related to the identification of the candidate pool of papers, primary selection bias, data extraction, and data synthesis.

The selection of search terms and the limitations of search engines can lead to an incomplete set of candidate pool of papers. We carefully selected our search terms by examining related work and queried five widely used online databases used in SLRs on SE topics. We also combined database search with manual inspection of related reviews and forward snowballing using Google

Table A.2

Venue	Number of primary studies	Reference(s)
IEEE Access	10	Al Qasem et al. (2020), Cai et al. (2019), Chen et al. (2019), Deng et al. (2020b), Liang et al. (2019), Lin and Lu (2021), Qiu et al. (2019c), Sheng et al. (2020), Sun et al. (2020a) and Zhao et al. (2018)
IET Software	5	Deng et al. (2020a), Huang et al. (2021), Zhang et al. (2021b), Zhao et al. (2021b) and Zhu et al. (2020)
International Conference on Software Quality, Reliability and Security (QRS)	4	Li et al. (2017), Yang et al. (2015), Zhang et al. (2018) and Zhou and Lu (2020)
Expert Systems with Applications	3	Majd et al. (2020), Pandey et al. (2020) and Turabieh et al. (2019)
IEEE Transactions on Reliability	3	Wang et al. (2021) and Xu et al. (2021a,b)
IEEE Transactions on Software Engineering	3	Yedida and Menzies (2021), Wang et al. (2020) and Wen et al. (2018)
Applied Sciences	2	Pan et al. (2019) and Qiu et al. (2019b)
Asia-Pacific Software Engineering Conference (APSEC)	2	Fan et al. (2019b) and Zhang et al. (2020)
Information and Software Technology	2	Tong et al. (2018) and Zhou et al. (2019)
Information Sciences	2	Zhang et al. (2021a) and Zhu et al. (2021a)
International Conference on Dependable Systems and Their Applications (DSA)	2	Liu et al. (2020) and Yu et al. (2021b)
International Conference on Mining Software Repositories (MSR)	2	Dam et al. (2019) and Hoang et al. (2019)
International Conference on Software Engineering (ICSE)	2	Chen et al. (2020) and Wang et al. (2016)
International Conference on Software Engineering and Knowledge Engineering	2	Qiu et al. (2019a) and Wang and Lu (2020)
International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering (RAISE)	2	Humphreys and Dam (2019) and Young et al. (2018)
Journal of Systems and Software	2	Xu et al. (2019) and Zhu et al. (2021b)
Knowledge-Based Systems	2	Pandey and Tripathi (2020, 2021)
Neural Computing and Applications	2	Ardimento et al. (2021) and Nevendra and Singh (2021)
Neurocomputing	2	Qiao et al. (2020) and Zhao et al. (2019)
PLOS ONE	2	Munir et al. (2021) and Qiao and Wang (2019)
ACM on Programming Languages (OOPSLA)	1	Li et al. (2019c)
ACM SIGSOFT International Symposium on Software Testing and Analysis	1	Zeng et al. (2021)
Algorithms and Architectures for Parallel Processing	1	Sun et al. (2020b)
Annual ACM Symposium on Applied Computing	1	Zhao et al. (2021a)
Array	1	Ferenc et al. (2020)
Asia Pacific Symposium on Intelligent and Evolutionary Systems (IES)	1	Phan and Nguyen (2017)
Chinese Conference on Pattern Recognition and Computer Vision (PRCV)	1	Li et al. (2019b)
Chinese Journal Electronics	1	Wei et al. (2019)
Cluster Computing	1	Manjula and Florence (2019)
Cognitive Systems Research	1	Geng (2018)
Conference on Knowledge Based Engineering and Innovation (KBEI)	1	Eivazpour and Keyvanpour (2019)
Congress on Intelligent Systems	1	Thaher and Khamayseh (2021)
Euromicro Conference on Software Engineering and Advanced Applications (SEAA)	1	Fiore et al. (2021)
Future Internet	1	Albahli (2019)
IEEE Annual Computers, Software, and Applications Conference (COMPSAC)	1	Yu et al. (2021a)
IEEE International Conference for Innovation in Technology (INOCON)	1	Yadav (2020)
IEEE International Conference on Big Data and Cloud Computing (BdCloud)	1	Sun et al. (2018)
IEEE International Conference on Data Mining (ICDM)	1	Huo et al. (2018)
IEEE International Conference on Industrial Engineering and Engineering Management (IEEM)	1	Huang et al. (2019)
IEEE Uttar Pradesh Section International Conference on Electrical, Electronics and Computer Engineering (UPCON)	1	Bhandari and Gupta (2018)
IEICE TRANSACTIONS on Information and Systems	1	Gong et al. (2019)
Intelligent Data Analysis	1	Saifan and Al Smadi (2019)
International Conference on Advances in Science, Engineering and Robotics Technology (ICASERT)	1	Ayon (2019)
International Conference on Advances in the Emerging Computing Technologies (AECT)	1	Abozeed et al. (2020)
International Conference on Artificial Intelligence and Security (ICAIS)	1	Sun et al. (2021)
International Conference on Artificial Intelligence for Communications and Networks (AICON)	1	Zhu et al. (2019)

(*continued on next page*)

Table A.2 (*continued*).

Venue	Number of primary studies	Reference(s)
International Conference on Computational Performance Evaluation (ComPE)	1	Singh et al. (2020)
International Conference on Computer Communications and Networks (ICCCN)	1	Tian and Tian (2020)
International Conference on Computer Systems and Applications (AICCSA)	1	Samir et al. (2019)
International Conference on Electrical Engineering, Computer Sciences and Informatics (EECSI)	1	Bahaweres et al. (2020)
International Conference on Internet of Things and Connected Technologies (ICIoTCT)	1	Chatterjee et al. (2021)
International Conference on Knowledge and Systems Engineering (KSE)	1	Tran et al. (2019)
International Conference On Smart Cities, Automation & Intelligent Computing Systems (ICON-SONICS)	1	Bahaweres et al. (2021)
International Conference on Soft Computing and Signal Processing (ICSCSP)	1	Malohtra and Yadav (2021)
International Conference on Tools with Artificial Intelligence (ICTAI)	1	Phan et al. (2017)
International Cyberspace Congress, CyberDI and CyberLife	1	Yang et al. (2019)
International Joint Conference on Neural Networks (IJCNN)	1	Li et al. (2019a)
International Journal of Web Services Research (IJWSR)	1	Bhandari and Gupta (2020)
Journal of Computer Languages	1	Shi et al. (2020)
Journal of Software: Evolution and Process	1	Shi et al. (2021)
Mathematical Problems in Engineering	1	Song et al. (2021)
PeerJ Computer Science	1	Farid et al. (2021)
Progress in Advanced Computing and Intelligent Engineering (ICACIE)	1	Tameswar et al. (2021)
Scientific Programming	1	Fan et al. (2019a)
Software Quality Journal	1	Wu et al. (2021)
Wireless Personal Communications	1	Dong et al. (2018)

Scholar, while there is a risk of missing out some studies due to not performing backward snowballing (Badampudi et al., 2015). Nevertheless, we think that an adequate set of primary studies was collected for this study.

Application of inclusion and exclusion criteria is subject to researchers' bias and hence a potential threat to validity. The authors built a list of inclusion and exclusion criteria and applied a joint voting mechanism to mitigate the risk of ambiguous interpretations. Two authors independently applied inclusion and exclusion criteria to each candidate paper and agreed on 86% of the papers. All the conflicts between two authors' votes were recorded and resolved via the votes of the third and fourth authors.

Another essential aspect that directly affects the results of this study is the validity of the data extraction. The authors started with initial categories that were formed using the existing categories in the literature. In addition, the authors refined the categories iteratively and incrementally. They aimed at decreasing the risk of researcher bias via mapping the relevant data in primary studies to the specified categories. Whenever an author was undecided about the data to be extracted, he recorded that case, and these cases were resolved via discussions among the authors.

We used descriptive statistics to synthesize data for the RQs from one to nine. We think that threats to internal validity are relatively small for the responses to these RQs. We applied the open coding technique iteratively and incrementally to identify the challenges and solutions (RQ10). This coding process potentially entails some researcher bias.

6. Conclusion and future work

SDP comprises various techniques for automatically identifying defects and therefore help reduce the effort in fixing them. This is particularly beneficial nowadays given the increasing volume of software and scarce quality assurance resources. SDP using DL has particularly gained traction in the recent years. In this study, we performed a systematic literature review of existing SDP techniques using DL to paint a picture on the state-of-the-art. We applied a rigorous process to search for articles in several scientific databases, supported with snowballing. As a result of a multiple-assessor quality assessment step with well-defined criteria, we chose the articles to be considered for analysis. Our survey eventually included a total of 102 high-quality primary studies. Based on those we conducted quantitative and qualitative analysis on the pool of studies with respect to various aspects of SDP: SDP scenarios, ML categories, datasets, representation of source code, granularity level of prediction, dealing with the class imbalance problem, DL approaches, evaluation metrics and validation approach, reproducibility, and finally challenges along with several proposed solutions.

The results indicate an increasing trend of SDP research over the recent years, with a big variety of fundamental techniques, datasets and validation approaches being employed. An important observation is the lack of reproducibility packages for most of the surveyed articles, which can be problematic for transparency and further advancement of the field. We have also collected the reported challenges around the data engineering, model development aspects and SDP in general, along with several solutions proposed by researchers. Together with our critical discussion, we propose the following directions to pave the way for further research:

- Development of new, more comprehensive DL approaches automatically capturing richer representations and features from heterogeneous sources (source code, bug reports and others),
- Development of data augmentation techniques for tackling limited dataset sizes and class imbalance,
- Identification of key source code defect attributes for defect prediction as well as exploitation of automatic feature extraction of DL approaches,
- Establishing common criteria for evaluating the performance of DL-based SDP,
- More focus on CPDP scenarios next to WPDP,
- Better usability of SDP tools and integration into the daily practice of users,
- Reproducibility and open science.

Our results can be beneficial for both newcomers to SDP research to see the landscape of different approaches, and established researchers to focus their efforts in the coming years. As future work we aim to perform a more in-depth investigation into the state-of-the-art in SDP using DL, particularly doing an extensive

Table A.3

Category	Challenge	Proposed solution
Data engineering	Source and target dataset's different features and data distributions degrade the performance of classifier (Albahli, 2019; Gong et al., 2019; Li et al., 2019a; Qiu et al., 2019a,b; Sheng et al., 2020; Wang and Lu, 2020; Sun et al., 2020a; Huang et al., 2021; Sun et al., 2021; Wu et al., 2021)	Li et al. (2019a) used a cost-sensitive shared hidden layer autoencoder with shared parameter mechanism to make the distribution of source and target datasets more similar by minimizing reconstruction error loss. Qiu et al. (2019a) proposed a Transferable Hybrid Features Learning with CNN. Qiu et al. (2019a,b,c) employed a matching layer to bridge the source and target datasets to mine the transferable semantic-based features by simultaneously minimizing classification error and distribution divergence between projects. Albahli (2019) checked training data against outliers and processed these outliers accordingly to obtain a better model. Wang and Lu (2020) introduced a domain confusion loss based maximum mean discrepancy (MMD) in feature extraction to bridge the substantial distributional discrepancy between different projects. Wu et al. (2021) adopted a modified autoencoder algorithm for instance selection. Sheng et al. (2020) proposed Adversarial Discriminative Convolutional Neural Network (ADCNN) to extract transferrable semantic features from source code for CPDP tasks. Sun et al. (2021) proposed a deep adversarial learning based HDP approach and leveraged DNN to learn nonlinear transformation for each project to obtain common feature representation, which the heterogeneous data from different projects can be compared directly. Huang et al. (2021) proposed a model based on multi-adaptation and nuclear form to deal with different samples. Gong et al. (2019) utilized the Maximum Mean Discrepancy (MMD) to calculate the distance between the source and target data.
Data engineering	Developing a successful SDP model is a challenge when training data with sufficient amount and quality are not present (Zhao et al., 2018, 2019; Pandey and Tripathi, 2021).	Zhao et al. (2018, 2019) proposed using Siamese networks, which are capable of learning with a few samples. Pandey and Tripathi (2021) added more labeled data to their training set to improve the performance of DNN-based model.
Data engineering	Training data formation (Li et al., 2019b)	Li et al. (2019b) manually checked bug reports to validate whether reported bugs were true bugs. On the other hand, this introduces a bias since researchers are not the actual developers of the projects and hence may misunderstand the code and bug.
Data engineering	For change-level defect prediction, code snippets are used as training data and building AST for an incomplete code snippet is challenging (Wang et al., 2020).	Wang et al. (2020) proposed a heuristic approach to extracting important structural and context information from incomplete code snippets.
Data engineering	In a CPDP scenario, the granularity of the AST nodes and the data distribution difference among datasets may have negative impacts on the prediction performance (Deng et al., 2020b).	Deng et al. (2020b) proposed a CPDP framework based on multi-kernel transfer CNNs by considering AST node granularity.
Model development	Traditional features (such as lines of code, operand and operator counts, number of methods in a class, the position of a class in inheritance tree, and McCabe complexity) are not sensitive to programs' semantic information and hence harm defect prediction performance (Wang et al., 2016; Li et al., 2017; Huo et al., 2018; Cai et al., 2019; Chen et al., 2019; Dam et al., 2019; Fan et al., 2019a,b; Liang et al., 2019; Qiu et al., 2019c; Xu et al., 2019; Wang et al., 2020; Zhang et al., 2020; Bahaweres et al., 2021; Chatterjee et al., 2021; Xu et al., 2021a; Zhang et al., 2018, 2021a).	(Wang et al., 2016, 2020) leveraged the semantic features learned by DBN to improve WPDP and CPDP performance. Chen et al. (2019) a simplified AST for representation to capture semantic information of source code. They simplified AST by including project-independent nodes and ignoring project specific nodes (such as method and variable names). Xu et al. (2021a) used graph neural networks to capture semantic and context information using ASTs and learn latent defect information of defective subtrees. Chatterjee et al. (2021) utilized DNN to learn features automatically instead of designing handcrafted features. Huo et al. (2018) used code comments to generate semantic features besides other features to train a CNN. Dam et al. (2019) used a tree LSTM network that matches with AST representation to represent syntax and semantics of source code better. Liang et al. (2019) proposed a Semantic LSTM network to capture semantic information of source code.

(*continued on next page*)

Table A.3 (*continued*).

Category	Challenge	Proposed solution
		Zhang et al. (2020) proposed a model based on ensemble learning techniques and attention mechanisms for better source code representation. Li et al. (2017) used CNN to learn semantic and structural features of programs. Zhang et al. (2021a) used a hybrid model based on WGAN-GP (Wasserstein GAN with Gradient Penalty), multi-objective NSGA-III (Non-dominated Sorting Genetic Algorithm-III) algorithm and hybrid CNNSVM (Convolutional Neural Network – Support Vector Machine) to represent complex structure of programs. Xu et al. (2019) use a DNN with a new hybrid loss function that consists of a triplet loss to learn a more discriminative feature representation of defect data. Qiu et al. (2019c) proposed a new model, named neural forest (NF), which uses the DNN and decision forest to build a holistic system for the automatic exploration of powerful feature representations. Bahaweres et al. (2021) used AST nodes and word embeddings to build an LSTM network. Fan et al. (2019a,b) used word embeddings obtained via ASTs to form numeric vectors. Zhang et al. (2018) used cross-entropy, a common measure for natural language, as a new code metric and combined it with traditional metrics. Cai et al. (2019) used an AST-based representation along with Euclidean distance to represent semantic distance between nodes.
Model development	Feature redundancy, i.e., highly correlated features, may harm prediction performance (Tran et al., 2019; Turabieh et al., 2019; Wei et al., 2019; Zhao et al., 2021b; Zhu et al., 2021b)	Tran et al. (2019) leveraged DL and ensemble learning to learn effective representations of metrics. Wei et al. (2019) used DBN to learn features. Zhao et al. (2021b) used Principal Component Analysis (PCA) for feature representation learning. Turabieh et al. (2019) used a pool of meta-heuristic-based feature selection methods (i.e., Genetic Algorithm, Particle Swarm Optimization, and Ant Colony Optimization) to select features. Zhu et al. (2021b) leveraged Whale Optimization Algorithm (WOA) and another complementary Simulated Annealing (SA) to construct an enhanced metaheuristic search-based feature selection algorithm.
Model development	Manual feature selection may harm prediction performance (Chen et al., 2020)	Chen et al. (2020) represented source code as images, applied self-attention mechanism to extract image features, and fed images to a pre-trained DL model for SDP.
Model development	Different feature sets may provide the best prediction performance in different contexts (Wang et al., 2021).	Wang et al. (2021) used a gated merge layer in their LSTM network to obtain an optimum combination of the input features.
Model development	Random selection of initial input weights and hidden layer biases of Extreme Learning Machine (ELM) may lead to lower model performance (Zhang et al., 2021b).	Zhang et al. (2021b) utilized metaheuristic intelligence optimization algorithms to determine optimal input weights and hidden layer biases of ELM, including Gravitational Search Algorithm (GSA) and Particle Swarm Optimization (PSO).
Model development	Conventional classification algorithms, e.g., Naïve Bayes, Decision Tree, Logistic Regression, assume features represented by vectors of the same length. This scheme is not appropriate for the representation of change sequences with varying lengths (Wen et al., 2018).	Wen et al. (2018) used RNN, which can automatically derive latent features from sequence data.
Model development	Sequential networks do not represent syntax and semantics of AST and fail to capture dependencies in source code (Zhou and Lu, 2020; Yu et al., 2021a).	Zhou and Lu (2020) used a bidirectional LSTM to represent dependencies in source code and tree LSTM network to capture syntactic information from AST. Yu et al. (2021a) used a hierarchical neural network.
Model development	DL models are sensitive to hyperparameters leading to very different performance results. This is a serious challenge in reproducing previous experiments for which all hyperparameters were not reported (Pan et al., 2019).	No solution proposed
Model development	Model overfitting (Pandey and Tripathi, 2020)	Pandey and Tripathi (2020) used dropout regularization to avoid overfitting.
Model development	The performance of a prediction model may degrade (Albahli, 2019).	Albahli (2019) adjusted their prediction model according to the feedback (input from users on whether a prediction is correct).

(*continued on next page*)

Table A.3 (*continued*).

Category	Challenge	Proposed solution
General	Early defect prediction is a challenging task (Manjula and Florence, 2019).	Manjula and Florence (2019) combined genetic algorithm for feature optimization with DNN for classification and observed a performance improvement due to the application of optimization technique.
General	SDP at course-grained levels puts burden on developers for bug localization (Majd et al., 2020; Munir et al., 2021).	Majd et al. (2020) and Munir et al. (2021) proposed a DL-based approach for statement-level SDP.

meta-analysis on factors influencing the performance of SDP as reported in the surveyed articles in this study.

CRediT authorship contribution statement

Görkem Giray: Conceptualization, Methodology, Data curation, Supervision, Validation, Writing – original draft, Writing – review & editing. **Kwabena Ebo Bennin:** Data curation, Validation, Writing – original draft. **Ömer Köksal:** Data extraction, Validation, Writing – original draft. **Önder Babur:** Data extraction, Validation, Writing – original draft. **Bedir Tekinerdogan:** Writing – review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Data will be made available on request.

Appendix

A.1. Search strings for online databases

See Table A.1.

A.2. Distribution of the primary studies per venue

See Table A.2

A.3. The challenges and proposed solutions

See Table A.3

References

Akimova, E.N., Bersenev, A.Y., Deikov, A.A., Kobylkin, K.S., Konygin, A.V., Mezentsev, I.P., Misilov, V.E., 2021. A survey on software defect prediction using deep learning. Mathematics 9 (11), 1180.

Amershi, S., Begel, A., Bird, C., DeLine, R., Gall, H., Kamar …, E., Zimmermann, T., 2019. Software engineering for machine learning: A case study. In: 2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP). IEEE, pp. 291–300.

Apicella, A., Donnarumma, F., Isgrò, F., Prevete, R., 2021. A survey on modern trainable activation functions. Neural Netw. 138, 14–32.

Arar, Ö.F., Ayan, K., 2015. Software defect prediction using cost-sensitive neural network. Appl. Soft Comput. 33, 263–277. http://dx.doi.org/10.1016/J.ASOC.2015.04.045.

Atif, F., Rodriguez, M., Araújo, L.J., Amartiwi, U., Akinsanya, B.J., Mazzara, M., 2021. A survey on data science techniques for predicting software defects. In: International Conference on Advanced Information Networking and Applications. Springer, Cham, pp. 298–309.

Badampudi, D., Wohlin, C., Petersen, K., 2015. Experiences from using snowballing and database searches in systematic literature studies. In: Proceedings of the 19th International Conference on Evaluation and Assessment in Software Engineering. pp. 1–10.

Basili, V.R., Caldiera, G., Rombach, H.D., 1994. The goal question metric approach. In: Chapter in Encyclopedia of Software Engineering. Wiley.

Batool, I., Khan, T.A., 2022. Software fault prediction using data mining, machine learning and deep learning techniques: A systematic literature review. Comput. Electr. Eng. 100, 107886.

Behnke, S., 2003. Hierarchical neural networks for image interpretation. p. 2766. http://dx.doi.org/10.1007/B11963.

Bennin, K.E., Keung, J., Monden, A., Phannachitta, P., Mensah, S., 2017b. The significant effects of data sampling approaches on software defect prioritization and classification. In: 2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement. ESEM, IEEE, pp. 364–373.

Bennin, K.E., Keung, J., Phannachitta, P., Monden, A., Mensah, S., 2017a. Mahakil: Diversity based oversampling approach to alleviate the class imbalance issue in software defect prediction. IEEE Trans. Softw. Eng. 44 (6), 534–550.

Bennin, K.E., Toda, K., Kamei, Y., Keung, J., Monden, A., Ubayashi, N., 2016. Empirical evaluation of cross-release effort-aware defect prediction models. In: 2016 IEEE International Conference on Software Quality, Reliability and Security. QRS, IEEE, pp. 214–221.

Brownlee, J., 2020. Imbalanced classification with python: Better metrics, balance skewed classes, cost-sensitive learning. Mach. Learn. Mastery.

Cabral, G.G., Minku, L.L., Shihab, E., Mujahid, S., 2019. Class imbalance evolution and verification latency in just-in-time software defect prediction. In: Proceedings - International Conference on Software Engineering, 2019-May pp. 666–676. http://dx.doi.org/10.1109/ICSE.2019.00076.

Calikli, G., Tosun, A., Bener, A., Celik, M., 2009. The effect of granularity level on software defect prediction. In: 2009 24th International Symposium on Computer and Information Sciences. IEEE, pp. 531–536.

Catal, C., Diri, B., 2009. A systematic review of software fault prediction studies. Expert Syst. Appl. 36 (4), 7346–7354.

Catal, C., Giray, G., Tekinerdogan, B., 2021. Applications of deep learning for mobile malware detection: A systematic literature review. Neural Comput. Appl. 1–26.

Catal, C., Giray, G., Tekinerdogan, B., Kumar, S., Shukla, S., 2022. Applications of deep learning for phishing detection: A systematic literature review. Knowl. Inf. Syst..

Chawla, N.V., Bowyer, K.W., Hall, L.O., Kegelmeyer, W.P., 2002. SMOTE: synthetic minority over-sampling technique. J. Artificial Intelligence Res. 16, 321–357.

Chen, H., Jing, X.-Y., Zhou, Y., Li, B., Xu, B., 2022. Aligned metric representation based balanced multiset ensemble learning for heterogeneous defect prediction. Inf. Softw. Technol. 147, 106892. http://dx.doi.org/10.1016/J.INFSOF.2022.106892.

Chen, X., Xie, H., Li, Z., Cheng, G., 2021. Topic analysis and development in knowledge graph research: A bibliometric review on three decades. Neurocomputing 461, 497–515.

Chidamber, S.R., Kemerer, C.F., 1994. A metrics suite for object oriented design. IEEE Trans. Softw. Eng. 20 (6), 476–493. http://dx.doi.org/10.1109/32.295895.

Cho, K., Van Merriënboer, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H., Bengio, Y., 2014. Learning phrase representations using RNN encoder–decoder for statistical machine translation. arXiv preprint arXiv:1406.1078.

Chollampatt, S., Ng, H.T., 2018. A multilayer convolutional encoder–decoder neural network for grammatical error correction. In: Proceedings of the AAAI Conference on Artificial Intelligence, Vol. 32. No. 1.

Chung, J., Gulcehre, C., Cho, K., Bengio, Y., 2014. Empirical evaluation of gated recurrent neural networks on sequence modeling. Retrieved from http://arxiv.org/abs/1412.3555.

D'Ambros, M., Lanza, M., Robbes, R., 2010. An extensive comparison of bug prediction approaches. In: 2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010). IEEE, pp. 31–41.

D'Ambros, M., Lanza, M., Robbes, R., 2012. Evaluating defect prediction approaches: a benchmark and an extensive comparison. Empir. Softw. Eng. 17 (4), 531–577.

Devanbu, P., Dwyer, M., Elbaum, S., Lowry, M., Moran, K., Poshyvanyk ., D., Zhang, X., 2020. Deep learning & software engineering: State of research and future directions. arXiv preprint arXiv:2009.08525.

Ferenc, R., Tóth, Z., Ladányi, G., Siket, I., Gyimóthy, T., 2018. A public unified bug dataset for java. In: Proceedings of the 14th International Conference on Predictive Models and Data Analytics in Software Engineering. pp. 12–21.

Ferreira, F., Silva, L.L., Valente, M.T., 2021. Software engineering meets deep learning: a mapping study. In: Proceedings of the 36th Annual ACM Symposium on Applied Computing. pp. 1542–1549.

Garousi, V., Giray, G., Tüzün, E., Catal, C., Felderer, M., 2019. Aligning software engineering education with industrial needs: a meta-analysis. J. Syst. Softw. 156, 65–83.

Giray, G., 2021. A software engineering perspective on engineering machine learning systems: State of the art and challenges. J. Syst. Softw. 180, 111031.

Giray, G., Tüzün, E., 2018. A systematic mapping study on the current status of total cost of ownership for information systems. Inform. Technol. J. 11 (2), 131–145.

Goel, L., Damodaran, D., Khatri, S.K., Sharma, M., 2017. A literature review on cross project defect prediction. In: 2017 4th IEEE Uttar Pradesh Section International Conference on Electrical, Computer and Electronics. UPCON, IEEE, pp. 680–685.

Golovko, V., Kroshchanka, A., Rubanau, U., Jankowski, S., 2014. A learning technique for deep belief neural networks. In: International Conference on Neural Networks and Artificial Intelligence. Springer, Cham, pp. 136–146.

González-Barahona, J.M., Robles, G., 2012. On the reproducibility of empirical software engineering studies based on data retrieved from development repositories. Empir. Softw. Eng. 17 (1), 75–89.

Goodfellow, I., Bengio, Y., Courville, A., 2016. Deep Learning. MIT Press.

Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair ., S., Bengio, Y., 2014. Generative adversarial nets. Adv. Neural Inf. Process. Syst. 27.

Gousios, G., 2021. Representing source code as text. Online at https://gousios.org/courses/ml4se/srcastext.html Last accessed on 27 2022.

Guan, Z., Wang, X., Xin, W., Wang, J., Zhang, L., 2020. A survey on deep learning-based source code defect analysis. In: 2020 5th International Conference on Computer and Communication Systems. ICCCS, IEEE, pp. 167–171.

Hall, T., Beecham, S., Bowes, D., Gray, D., Counsell, S., 2011. A systematic literature review on fault prediction performance in software engineering. IEEE Trans. Softw. Eng. 38 (6), 1276–1304.

Hassler, E., Carver, J.C., Kraft, N.A., Hale, D., 2014. Outcomes of a community workshop to identify and rank barriers to the systematic literature review process. In: Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering. pp. 1–10.

Hochreiter, S., Schmidhuber, J., 1997. Long short-term memory. Neural Comput. 9 (8), 1735–1780. http://dx.doi.org/10.1162/neco.1997.9.8.1735.

Hosseini, S., Turhan, B., Gunarathna, D., 2017. A systematic literature review and meta-analysis on cross project defect prediction. IEEE Trans. Softw. Eng. 45 (2), 111–147.

Huang, G. Bin, Zhu, Q.Y., Siew, C.K., 2006. Extreme learning machine: Theory and applications. Neurocomputing 70 (1–3), 489–501. http://dx.doi.org/10.1016/J.NEUCOM.2005.12.126.

Jiang, T., Tan, L., Kim, S., 2013. Personalized defect prediction. In: 2013 28th IEEE/ACM International Conference on Automated Software Engineering. ASE, Ieee, pp. 279–289.

Jorayeva, M., Akbulut, A., Catal, C., Mishra, A., 2022. Machine learning-based software defect prediction for mobile applications: A systematic literature review. Sensors 22 (7), 2551.

Joshi, Mahesh V., Kumar, Vipin, Agarwal, Ramesh C., 2001. Evaluating boosting algorithms to classify rare classes: Comparison and improvements. In: Proceedings 2001 IEEE International Conference on Data Mining. IEEE pp. 257–264.

Jureczko, M., Madeyski, L., 2010. Towards identifying software project clusters with regard to defect prediction. In: Proceedings of the 6th International Conference on Predictive Models in Software Engineering. pp. 1–10.

Kalaivani, N., Beena, R., 2018. Overview of software defect prediction using machine learning algorithms. Int. J. Pure Appl. Math. 118 (20), 3863–3873.

Kamei, Y., Shihab, E., 2016. Defect prediction: Accomplishments and future challenges. In: 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER), Vol. 5. IEEE, pp. 33–45.

Kamei, Y., Shihab, E., Adams, B., Hassan, A.E., Mockus, A., Sinha, A., Ubayashi, N., 2013. A large-scale empirical study of just-in-time quality assurance. IEEE Trans. Softw. Eng. 39 (6), 757–773. http://dx.doi.org/10.1109/TSE.2012.70.

Khoshgoftaar, T.M., Seliya, N., 2004. Comparative assessment of software quality classification techniques: An empirical case study. Empir. Softw. Eng. 9 (3), 229–257.

Kitchenham, B., Brereton, O.P., Budgen, D., Turner, M., Bailey, J., Linkman, S., 2009. Systematic literature reviews in software engineering–a systematic literature review. Inf. Softw. Technol. 51 (1), 7–15.

Kitchenham, B.A., Budgen, D., Brereton, P., 2015. Evidence-Based Software Engineering and Systematic Reviews, Vol. 4. CRC Press.

Kitchenham, B., Charters, S., 2007. Guidelines for Performing Systematic Literature Reviews in Software Engineering. EBSE Technical report, Ver. 2.3, EBSE-2007-01.

Koru, A.G., Liu, H., 2005. An investigation of the effect of module size on defect prediction using static measures. In: Proceedings of the 2005 Workshop on Predictor Models in Software Engineering. pp. 1–5.

Kumar, S., Mallik, A., Khetarpal, A., Panda, B.S., 2022. Influence maximization in social networks using graph embedding and graph neural network. Inform. Sci. 607, 1617–1636.

LeCun, Y., Bengio, Y., Hinton, G., 2015. Deep learning. Nature 5217553, 436–444.

Lessmann, S., Baesens, B., Mues, C., Pietsch, S., 2008. Benchmarking classification models for software defect prediction: A proposed framework and novel findings. IEEE Trans. Softw. Eng. 34 (4), 485–496.

Lewowski, T., Madeyski, L., 2022. How far are we from reproducible research on code smell detection? A systematic literature review. Inf. Softw. Technol. 144, 106783.

Li, Z., Jing, X.Y., Zhu, X., 2018. Progress on approaches to software defect prediction. IET Softw. 12 (3), 161–175.

Li, N., Shepperd, M., Guo, Y., 2020. A systematic review of unsupervised learning techniques for software defect prediction. Inf. Softw. Technol. 122, 106287.

Liu, X., Gao, C., Li, P., 2012. A comparative analysis of support vector machines and extreme learning machines. Neural Netw. 33, 58–66. http://dx.doi.org/10.1016/J.NEUNET.2012.04.002.

Liu, C., Gao, C., Xia, X., Lo, D., Grundy, J., Yang, X., 2021a. On the reproducibility and replicability of deep learning in software engineering. ACM Trans. Softw. Eng. Methodol. (TOSEM) 31 (1), 1–46.

Liu, Y., Loh, H.T., Tor, S.B., 2005. Comparison of extreme learning machine with support vector machine for text classification. In: Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics). In: LNAI, vol. 3533, pp. 390–399. http://dx.doi.org/10.1007/11504894_55.

Loper, E., Bird, S., 2002. NLTK: the natural language toolkit. In: Proceedings of the ACL-02 Workshop on Effective Tools and Methodologies for Teaching Natural Language Processing and Computational Linguistics-Volume 1 pp. 63–70.

Majumder, S., Mody, P., Menzies, T., 2022. Revisiting process versus product metrics: a large scale analysis. Empir. Softw. Eng. 27 (3), 1–42.

Malhotra, R., 2015. A systematic review of machine learning techniques for software fault prediction. Appl. Soft Comput. 27, 504–518.

Malhotra, R., Gupta, S., Singh, T., 2020. A systematic review on application of deep learning techniques for software quality predictive modeling. In: 2020 International Conference on Computational Performance Evaluation (ComPE). IEEE, pp. 332–337.

Matloob, F., Ghazal, T.M., Taleb, N., Aftab, S., Ahmad, M., Khan ., M.A., Soomro, T.R., 2021. Software defect prediction using ensemble learning: A systematic literature review. IEEE Access.

Mavrovouniotis, M.L., Chang, S., 1992. Hierarchical neural networks. Comput. Chem. Eng. 16 (4), 347–369. http://dx.doi.org/10.1016/0098-1354(92)80053-C.

McCabe, T.J., 1976. A complexity measure. IEEE Trans. Softw. Eng. SE-2 (4), 308–320. http://dx.doi.org/10.1109/TSE.1976.233837.

Mende, T., Koschke, R., 2010. Effort-aware defect prediction models. In: 2010 14th European Conference on Software Maintenance and Reengineering. IEEE, pp. 107–116.

Menzies, Tim, Greenwald, Jeremy, Frank, Art, 2006. Data mining static code attributes to learn defect predictors. IEEE Trans. Softw. Eng. 33 (1), 2–13.

Mikolov, T., Chen, K., Corrado, G., Dean, J., 2013. Efficient estimation of word representations in vector space. arXiv preprint arXiv:1301.3781.

Miles, M.B., Huberman, A.M., Saldaña, J., 2019. Qualitative Data Analysis: A Methods Sourcebook, fourth ed. SAGE Publications, Inc.

Montavon, G., Samek, W., Müller, K.R., 2018. Methods for interpreting and understanding deep neural networks. Digit. Signal Process. 73, 1–15.

Motta, R.C., de Oliveira, K.M., Travassos, G.H., 2018. On challenges in engineering IoT software systems. In: Proceedings of the XXXII Brazilian Symposium on Software Engineering. pp. 42–51.

Mou, L., Li, G., Zhang, L., Wang, T., Jin, Z., 2016. Convolutional neural networks over tree structures for programming language processing. In: Thirtieth AAAI Conference on Artificial Intelligence.

Nam, J., 2014. Survey on Software Defect Prediction. Tech. Rep, Department of Compter Science and Engineerning, The Hong Kong University of Science and Technology.

Nam, J., Fu, W., Kim, S., Menzies, T., Tan, L., 2018. Heterogeneous defect prediction. IEEE Trans. Softw. Eng. 44 (09), 874–896. http://dx.doi.org/10.1109/TSE.2017.2720603.

Ni, C., Liu, W.S., Chen, X., Gu, Q., Chen, D.X., Huang, Q.G., 2017. A cluster based feature selection method for cross-project software defect prediction. J. Comput. Sci. Tech. 32 (6), 1090–1107.

Omri, S., Sinz, C., 2020. Deep learning for software defect prediction: A survey. In: Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops. pp. 209–214.

Özakıncı, R., Tarhan, A., 2018. Early software defect prediction: A systematic map and review. J. Syst. Softw. 144, 216–239.

Pachouly, J., Ahirrao, S., Kotecha, K., Selvachandran, G., Abraham, A., 2022. A systematic literature review on software defect prediction using artificial intelligence: Datasets, data validation methods, approaches, and tools. Eng. Appl. Artif. Intell. 111, 104773.

Pandey, S.K., Mishra, R.B., Tripathi, A.K., 2021. Machine learning based methods for software fault prediction: A survey. Expert Syst. Appl. 172, 114595.

Petersen, K., Feldt, R., Mujtaba, S., Mattsson, M., 2008. Systematic mapping studies in software engineering. In: 12th International Conference on Evaluation and Assessment in Software Engineering (EASE), Vol. 12. pp. 1–10.

Petersen, K., Vakkalanka, S., Kuzniarz, L., 2015. Guidelines for conducting systematic mapping studies in software engineering: An update. Inf. Softw. Technol. 64, 1–18.

Prasad, V.S., Sasikala, K., 2019. Software defect prediction techniques: A review. J. Inf. Comput. Sci. 9 (9), 619–638.

Radjenović, D., Heričko, M., Torkar, R., Živkovič, A., 2013. Software fault prediction metrics: A systematic literature review. Inf. Softw. Technol. 55 (8), 1397–1418.

Rahman, F., Devanbu, P., 2013. How, and why, process metrics are better. In: 2013 35th International Conference on Software Engineering. ICSE, IEEE pp. 432–441.

Rao, D., McMahan, B., 2019. Natural Language Processing with PyTorch : Build Intelligent Language Applications using Deep Learning. O'Reilly M.

Rathore, S.S., Kumar, S., 2019. A study on software fault prediction techniques. Artif. Intell. Rev. 51 (2), 255–327.

Rodríguez, D., Ruiz, R., Riquelme, J.C., Aguilar-Ruiz, J.S., 2012. Searching for rules to detect defective modules. Inf. Sci. Inf. Comput. Sci. Intell. Syst. Appl. Int. J. 191, 14–30. http://dx.doi.org/10.1016/J.INS.2011.01.039.
Shepperd, M., Song, Q., Sun, Z., Mair, C., 2013. Data quality: Some comments on the nasa software defect datasets. IEEE Trans. Softw. Eng. 39 (9), 1208–1215.
Son, L.H., Pritam, N., Khari, M., Kumar, R., Phuong, P.T.M., Thong, P.H., 2019. Empirical study of software defect prediction: a systematic mapping. Symmetry 11 (2), 212.
Spinellis, D., 2005. Tool writing: a forgotten art? (software tools). IEEE Softw. 22 (4), 9–11.
Tarhan, A., Giray, G., 2017. On the use of ontologies in software process assessment: a systematic literature review. In: Proceedings of the 21st International Conference on Evaluation and Assessment in Software Engineering. pp. 2–11.
Tóth, Z., Gyimesi, P., Ferenc, R., 2016. A public bug database of github projects and its application in bug prediction. In: International Conference on Computational Science and Its Applications. Springer, Cham, pp. 625–638.
Turhan, B., Menzies, T., Bener, A.B., Stefano, J.Di., 2009. On the relative value of cross-company and within-company data for defect prediction. Empir. Softw. Eng. 14 (5), 540–578.
Vannucci, M., Colla, V., 2018. Self–organizing–maps based undersampling for the classification of unbalanced datasets. In: 2018 International Joint Conference on Neural Networks. IJCNN, IEEE, pp. 1–6.
Wahono, R.S., 2015. A systematic literature review of software defect prediction. J. Softw. Eng. 1 (1), 1–16.
Wan, Z., Xia, X., Hassan, A.E., Lo, D., Yin, J., Yang, X., 2018. Perceptions, expectations, and challenges in defect prediction. IEEE Trans. Softw. Eng. 46 (11), 1241–1266.
Wang, J., He, H., Cao, Y., Xu, J., Zhao, D., 2012. A hierarchical neural network architecture for classification. In: Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics). In: LNCS (PART 1), vol. 7367, pp. 37–46. http://dx.doi.org/10.1007/978-3-642-31346-2_5.
Watson, C., Cooper, N., Palacio, D.N., Moran, K., Poshyvanyk, D., 2020. A systematic literature review on the use of deep learning in software engineering research. arXiv preprint arXiv:2009.06520.
Wohlin, C., 2014. Guidelines for snowballing in systematic literature studies and a replication in software engineering. In: Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering. pp. 1–10.
Wu, R., Zhang, H., Kim, S., Cheung, S.C., 2011. Relink: recovering links between bugs and changes. In: Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering pp. 15–25.
Yang, Y., Xia, X., Lo, D., Grundy, J., 2020. A survey on deep learning for software engineering. arXiv preprint arXiv:2011.14597.
Yu, X., Wu, M., Jian, Y., Bennin, K.E., Fu, M., Ma, C., 2018. Cross-company defect prediction via semi-supervised clustering-based data filtering and mstra-based transfer learning. Soft Comput. 22 (10), 3461–3472.
Zhang, J., Zhang, J., Wu, J., Wu, J., Chen, C., Chen ., C., Lyu, M.R., 2020b. CDS: A cross-version software defect prediction model with data selection. IEEE Access 8, 110059–110072. http://dx.doi.org/10.1109/ACCESS.2020.3001440.
Zheng, W., Shen, T., Chen, X., Deng, P., 2022. Interpretability application of the just-in-time software defect prediction model. J. Syst. Softw. 188, 111245.

List of Excluded Studies

Ahmed, M.M., Kiran, B.S., Sai, P.H., Bisi, M., 2021. Software fault-prone module classification using learning automata based deep neural network model. In: 2021 12th International Conference on Computing Communication and Networking Technologies. ICCCNT, IEEE, pp. 1–6.
Al Qasem, O., Akour, M., 2019. Software fault prediction using deep learning algorithms. Int. J. Open Source Softw. Process. (IJOSSP) 10 (4), 1–19.
Alazawi, S.A., Salam, M.N.A., 2021. Evaluation of LMT and DNN algorithms in software defect prediction for open-source software. In: Research in Intelligent and Computing in Engineering. Springer, Singapore, pp. 189–203.
Ardimento, P., Aversano, L., Bernardi, M.L., Cimitile, M., 2020. Temporal convolutional networks for just-in-time software defect prediction. In: ICSOFT. pp. 384–393.
Chaubey, P.K., Arora, T.K., 2020. Software bug prediction and classification by global pooling of different activation of convolution layers. Mater. Today: Proc..
Kumar, Y., Singh, V., 2021. A practitioner approach of deep learning based software defect predictor. Ann. Rom. Soc. Cell Biol. 25 (6), 14615–14635.
Lin, X., Yang, J., Li, Z., 2020. Software defect prediction with spiking neural networks. In: International Conference on Neural Information Processing. Springer, Cham, pp. 660–667.
Liu, W., Wang, B., Wang, W., 2021b. Deep learning software defect prediction methods for cloud environments research. Sci. Program. 2021.
Manjula, C., Florence, L., 2018. Software defect prediction using deep belief network with L1-regularization based optimization. Int. J. Adv. Res. Comput. Sci. 9 (1).
Thapa, S., Alsadoon, A., Prasad, P.W.C., Al-Dala'in, T., Rashid, T.A., 2020. Software defect prediction using atomic rule mining and random forest. In: 2020 5th International Conference on Innovative Technologies in Intelligent Systems and Industrial Applications. CITISIA, IEEE, pp. 1–8.
Wongpheng, K., Visutsak, P., 2020. Software defect prediction using convolutional neural network. In: 2020 35th International Technical Conference on Circuits/Systems, Computers and Communications (ITC-CSCC). IEEE, pp. 240–243.
Zhang, Q., Wu, B., 2020. Software defect prediction via transformer. In: 2020 IEEE 4th Information Technology, Networking, Electronic and Automation Control Conference (ITNEC), Vol. 1. IEEE, pp. 874–879.
Zheng, W., Mo, S., Jin, X., Qu, Y., Xie, Z., Shuai, J., 2019. Software defect prediction model based on improved deep forest and autoencoder by forest. In: SEKE. pp. 419–540.
Zheng, W., Tan, L., Liu, C., 2021. Software defect prediction method based on transformer model. In: 2021 IEEE International Conference on Artificial Intelligence and Computer Applications. ICAICA, IEEE, pp. 670–674.

Primary Studies (Sources Reviewed in the SLR)

Abozeed, S.M., ElNainay, M.Y., Fouad, S.A., Abougabal, M.S., 2020. Software bug prediction employing feature selection and deep learning. In: 2019 International Conference on Advances in the Emerging Computing Technologies. AECT, IEEE, pp. 1–6.
Al Qasem, O., Akour, M., Alenezi, M., 2020. The influence of deep learning algorithms factors in software fault prediction. IEEE Access 8, 63945–63960.
Albahli, S., 2019. A deep ensemble learning method for effort-aware just-in-time defect prediction. Future Internet 11 (12), 246.
Ardimento, P., Aversano, L., Bernardi, M.L., Cimitile, M., Iammarino, M., 2021. Just-in-time software defect prediction using deep temporal convolutional networks. Neural Comput. Appl. 1–21.
Ayon, S.I., 2019. Neural network based software defect prediction using genetic algorithm and particle swarm optimization. In: 2019 1st International Conference on Advances in Science, Engineering and Robotics Technology. ICASERT, IEEE, pp. 1–4.
Bahaweres, R.B., Agustian, F., Hermadi, I., Suroso, A.I., Arkeman, Y., 2020. Software defect prediction using neural network based SMOTE. In: 2020 7th International Conference on Electrical Engineering, Computer Sciences and Informatics. EECSI, IEEE, pp. 71–76.
Bahaweres, R.B., Jumral, D., Hermadi, I., Suroso, A.I., Arkeman, Y., 2021. Hybrid software defect prediction based on LSTM (long short term memory) and word embedding. In: 2021 2nd International Conference on Smart Cities, Automation & Intelligent Computing Systems (ICON-SONICS). IEEE, pp. 70–75.
Bhandari, G.P., Gupta, R., 2018. Measuring the fault predictability of software using deep learning techniques with software metrics. In: 2018 5th IEEE Uttar Pradesh Section International Conference on Electrical, Electronics and Computer Engineering. UPCON, IEEE, pp. 1–6.
Bhandari, G.P., Gupta, R., 2020. Fault prediction in SOA-based systems using deep learning techniques. Int. J. Web Serv. Res. 17 (3), 1–19.
Cai, Z., Lu, L., Qiu, S., 2019. An abstract syntax tree encoding method for cross-project defect prediction. IEEE Access 7, 170844-170853.
Chatterjee, A., Kumar, K.V., Mohapatra, D.P., 2021. Fault prediction using deep neural network. In: International Conference on Internet of Things and Connected Technologies. Springer, Cham, pp. 447–457.
Chen, D., Chen, X., Li, H., Xie, J., Mu, Y., 2019. DeepCPDP: Deep learning based cross-project defect prediction. IEEE Access 7, 184832-184848.
Chen, J., Hu, K., Yu, Y., Chen, Z., Xuan, Q., Liu, Y., Filkov, V., 2020. Software visualization and deep transfer learning for effective software defect prediction. In: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering. pp. 578–589.
Dam, H.K., Pham, T., Ng, S.W., Tran, T., Grundy, J., Ghose, A., . Kim, C.J., 2019. Lessons learned from using a deep tree-based model for software defect prediction in practice. In: 2019 IEEE/ACM 16th International Conference on Mining Software Repositories. MSR, IEEE, pp. 46–57.
Deng, J., Lu, L., Qiu, S., 2020a. Software defect prediction via LSTM. IET Softw. 14 (4), 443–450.
Deng, J., Lu, L., Qiu, S., Ou, Y., 2020b. A suitable AST node granularity and multi-kernel transfer convolutional neural network for cross-project defect prediction. IEEE Access 8, 66647–66661.
Dong, F., Wang, J., Li, Q., Xu, G., Zhang, S., 2018. Defect prediction in android binary executables using deep neural network. Wirel. Pers. Commun. 102 (3), 2261–2285.
Eivazpour, Z., Keyvanpour, M.R., 2019. Improving performance in software defect prediction using variational autoencoder. In: 2019 5th Conference on Knowledge Based Engineering and Innovation. KBEI, IEEE, pp. 644–649.
Fan, G., Diao, X., Yu, H., Yang, K., Chen, L., 2019a. Software defect prediction via attention-based recurrent neural network. Sci. Program. 2019.

Fan, G., Diao, X., Yu, H., Yang, K., Chen, L., 2019b. Deep semantic feature learning with embedded static metrics for software defect prediction. In: 2019 26th Asia-Pacific Software Engineering Conference. APSEC, IEEE, pp. 244–251.
Farid, A.B., Fathy, E.M., Eldin, A.S., Abd-Elmegid, L.A., 2021. Software defect prediction using hybrid model (CBIL) of convolutional neural network (CNN) and bidirectional long short-term memory (Bi-LSTM). PeerJ Comput. Sci. 7, e739.
Ferenc, R., Bán, D., Grósz, T., Gyimóthy, T., 2020. Deep learning in static, metric-based bug prediction. Array 6, 100021.
Fiore, A., Russo, A., Gravino, C., Risi, M., 2021. Combining CNN with DS 3 for detecting bug-prone modules in cross-version projects. In: 2021 47th Euromicro Conference on Software Engineering and Advanced Applications. SEAA, IEEE, pp. 91–98.
Geng, W., 2018. Cognitive Deep Neural Networks prediction method for software fault tendency module based on Bound Particle Swarm Optimization. Cogn. Syst. Res. 52, 12–20.
Gong, L., Jiang, S., Yu, Q., Jiang, L., 2019. Unsupervised deep domain adaptation for heterogeneous defect prediction. IEICE Trans. Inf. Syst. 102 (3), 537–549.
Hoang, T., Dam, H.K., Kamei, Y., Lo, D., Ubayashi, N., 2019. DeepJIT: an end-to-end deep learning framework for just-in-time defect prediction. In: 2019 IEEE/ACM 16th International Conference on Mining Software Repositories. MSR, IEEE, pp. 34–45.
Huang, C.Y., Huang, C., Yang, M.C., Su, W.C., 2019. A study of applying deep learning-based weighted combinations to improve defect prediction accuracy and effectiveness. In: 2019 IEEE International Conference on Industrial Engineering and Engineering Management. IEEM, IEEE, pp. 1471–1475.
Huang, Q., Ma, L., Jiang, S., Wu, G., Song, H., Jiang, L., Zheng, C., 2021. A cross-project defect prediction method based on multi-adaptation and nuclear norm. IET Softw..
Humphreys, J., Dam, H.K., 2019. An explainable deep model for defect prediction. In: 2019 IEEE/ACM 7th International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering. RAISE, IEEE, pp. 49–55.
Huo, X., Yang, Y., Li, M., Zhan, D.C., 2018. Learning semantic features for software defect prediction by code comments embedding. In: 2018 IEEE International Conference on Data Mining. ICDM, IEEE, pp. 1049–1054.
Li, H., Li, X., Chen, X., Xie, X., Mu, Y., Feng, Z., 2019b. Cross-project defect prediction via ASTToken2Vec and BLSTM-based neural network. In: 2019 International Joint Conference on Neural Networks. IJCNN, IEEE, pp. 1–8.
Li, J., He, P., Zhu, J., Lyu, M.R., 2017. Software defect prediction via convolutional neural network. In: 2017 IEEE International Conference on Software Quality, Reliability and Security. QRS, IEEE, pp. 318–328.
Li, J., Jing, X.Y., Wu, F., Sun, Y., Yang, Y., 2019a. A cost-sensitive shared hidden layer autoencoder for cross-project defect prediction. In: Chinese Conference on Pattern Recognition and Computer Vision. PRCV, Springer, Cham, pp. 491–502.
Li, Y., Wang, S., Nguyen, T.N., Nguyen, S.Van., 2019c. Improving bug detection via context-based code representation learning and attention-based neural networks. Proc. ACM Program. Lang. 3 (OOPSLA), 1–30.
Liang, H., Yu, Y., Jiang, L., Xie, Z., 2019. Seml: A semantic LSTM model for software defect prediction. IEEE Access 7, 83812–83824.
Lin, J., Lu, L., 2021. Semantic feature learning via dual sequences for defect prediction. IEEE Access 9, 13112–13124.
Liu, Q., Xiang, J., Xu, B., Zhao, D., Hu, W., Wang, J., 2020. Aging-related bugs prediction via convolutional neural network. In: 2020 7th International Conference on Dependable Systems and their Applications. DSA, IEEE, pp. 90–98.
Majd, A., Vahidi-Asl, M., Khalilian, A., Poorsarvi-Tehrani, P., Haghighi, H., 2020. SLDeep: Statement-level software defect prediction using deep-learning model on static code features. Expert Syst. Appl. 147, 113156.
Malohtra, R., Yadav, H.S., 2021. An improved CNN-based architecture for within-project software defect prediction. In: Soft Computing and Signal Processing. Springer, Singapore, pp. 335–349.
Manjula, C., Florence, L., 2019. Deep neural network based hybrid approach for software defect prediction using software metrics. Cluster Comput. 22 (4), 9847–9863.
Munir, H.S., Ren, S., Mustafa, M., Siddique, C.N., Qayyum, S., 2021. Attention based GRU-LSTM for software defect prediction. Plos One 16 (3), e0247444.
Nevendra, M., Singh, P., 2021. Defect count prediction via metric-based convolutional neural network. Neural Comput. Appl. 33 (22), 15319–15344.
Pan, C., Lu, M., Xu, B., Gao, H., 2019. An improved CNN model for within-project software defect prediction. Appl. Sci. 9 (10), 2138.
Pandey, S.K., Tripathi, A.K., 2020. Bcv-predictor: A bug count vector predictor of a successive version of the software system. Knowl.-Based Syst. 197, 105924.
Pandey, S.K., Tripathi, A.K., 2021. DNNAttention: A deep neural network and attention based architecture for cross project defect number prediction. Knowl.-Based Syst. 233, 107541.
Pandey, S.K., Mishra, R.B., Tripathi, A.K., 2020. BPDET: An effective software bug prediction model using deep representation and ensemble learning techniques. Expert Syst. Appl. 144, 113085.
Phan, A.V., Nguyen, M.Le., 2017. Convolutional neural networks on assembly code for predicting software defects. In: 2017 21st Asia Pacific Symposium on Intelligent and Evolutionary Systems. IES, IEEE, pp. 37–42.
Phan, A.V., Le Nguyen, M., Bui, L.T., 2017. Convolutional neural networks over control flow graphs for software defect prediction. In: 2017 IEEE 29th International Conference on Tools with Artificial Intelligence. ICTAI, IEEE, pp. 45–52.
Qiao, L., Wang, Y., 2019. Effort-aware and just-in-time defect prediction with neural network. PLoS One 14 (2), e0211359.
Qiao, L., Li, X., Umer, Q., Guo, P., 2020. Deep learning based software defect prediction. Neurocomputing 385, 100–110.
Qiu, S., Lu, L., Cai, Z., Jiang, S., 2019a. Cross-project defect prediction via transferable deep learning-generated and handcrafted features. In: SEKE. pp. 431–552.
Qiu, S., Xu, H., Deng, J., Jiang, S., Lu, L., 2019b. Transfer convolutional neural network for cross-project defect prediction. Appl. Sci. 9 (13), 2660.
Qiu, Y., Liu, Y., Liu, A., Zhu, J., Xu, J., 2019c. Automatic feature exploration and an application in defect prediction. IEEE Access 7, 112097-112112.
Saifan, A.A., Al Smadi, N., 2019. Source code-based defect prediction using deep learning and transfer learning. Intell. Data Anal. 23 (6), 1243–1269.
Samir, M., El-Ramly, M., Kamel, A., 2019. Investigating the use of deep neural networks for software defect prediction. In: 2019 IEEE/ACS 16th International Conference on Computer Systems and Applications. AICCSA, IEEE, pp. 1–6.
Sheng, L., Lu, L., Lin, J., 2020. An adversarial discriminative convolutional neural network for cross-project defect prediction. IEEE Access 8, 55241–55253.
Shi, K., Lu, Y., Chang, J., Wei, Z., 2020. PathPair2Vec: An AST path pair-based code representation method for defect prediction. J. Comput. Lang. 59, 100979.
Shi, K., Lu, Y., Liu, G., Wei, Z., Chang, J., 2021. MPT-embedding: An unsupervised representation learning of code for software defect prediction. J. Softw.: Evol. Process 33 (4), e2330.
Singh, R., Singh, J., Gill, M.S., Malhotra, R., 2020. Transfer learning code vectorizer based machine learning models for software defect prediction. In: 2020 International Conference on Computational Performance Evaluation (ComPE). IEEE, pp. 497–502.
Song, K., Lv, S., Hu, D., He, P., 2021. Software defect prediction based on elman neural network and cuckoo search algorithm. Math. Probl. Eng. 2021.
Sun, Y., Jing, X.Y., Wu, F., Li, J., Xing, D., Chen, H., Sun, Y., 2020a. Adversarial learning for cross-project semi-supervised defect prediction. IEEE Access 8, 32674–32687.
Sun, Y., Sun, Y., Wu, F., Jing, X.Y., 2021. Deep adversarial learning based heterogeneous defect prediction. In: International Conference on Artificial Intelligence and Security. Springer, Cham, pp. 326–337.
Sun, Y., Xu, L., Guo, L., Li, Y., Wang, Y., 2020b. A comparison study of VAE and GAN for software fault prediction. In: International Conference on Algorithms and Architectures for Parallel Processing. Springer, Cham, pp. 82–96.
Sun, Y., Xu, L., Li, Y., Guo, L., Ma, Z., Wang, Y., 2018. Utilizing deep architecture networks of VAE in software fault prediction. In: 2018 IEEE Intl Conf on Parallel & Distributed Processing with Applications, Ubiquitous Computing & Communications, Big Data & Cloud Computing, Social Computing & Networking, Sustainable Computing & Communications (ISPA/IUCC/BDCloud/SocialCom/SustainCom). IEEE, pp. 870–877.
Tameswar, K., Suddul, G., Dookhitram, K., 2021. Enhancing deep learning capabilities with genetic algorithm for detecting software defects. In: Progress in Advanced Computing and Intelligent Engineering. Springer, Singapore, pp. 211–220.
Thaher, T., Khamayseh, F., 2021. A classification model for software bug prediction based on ensemble deep learning approach boosted with SMOTE technique. In: Congress on Intelligent Systems. Springer, Singapore, pp. 99–113.
Tian, J., Tian, Y., 2020. A model based on program slice and deep learning for software defect prediction. In: 2020 29th International Conference on Computer Communications and Networks. ICCCN, IEEE, pp. 1–6.
Tong, H., Liu, B., Wang, S., 2018. Software defect prediction using stacked denoising autoencoders and two-stage ensemble learning. Inf. Softw. Technol. 96, 94–111.
Tran, H.D., Hanh, L.T.M., Binh, N.T., 2019. Combining feature selection, feature learning and ensemble learning for software fault prediction. In: 2019 11th International Conference on Knowledge and Systems Engineering. KSE, IEEE, pp. 1–8.
Turabieh, H., Mafarja, M., Li, X., 2019. Iterated feature selection algorithms with layered recurrent neural network for software fault prediction. Expert Syst. Appl. 122, 27–42.
Wang, H., Zhuang, W., Zhang, X., 2021. Software defect prediction based on gated hierarchical LSTMs. IEEE Trans. Reliab. 70 (2), 711–727.
Wang, S., Liu, T., Tan, L., 2016. Automatically learning semantic features for defect prediction. In: 2016 IEEE/ACM 38th International Conference on Software Engineering. ICSE, IEEE, pp. 297–308.
Wang, S., Liu, T., Nam, J., Tan, L., 2020. Deep semantic feature learning for software defect prediction. IEEE Trans. Softw. Eng. 46 (12), 1267–1293.
Wang, Z., Lu, L., 2020. A semantic convolutional auto-encoder model for software defect prediction. In: SEKE. pp. 323–328.

Wei, H., Shan, C., Hu, C., Zhang, Y., Yu, X., 2019. Software defect prediction via deep belief network. Chin. J. Electron. 28 (5), 925–932.
Wen, M., Wu, R., Cheung, S.C., 2018. How well do change sequences predict defects? sequence learning from software changes. IEEE Trans. Softw. Eng. 46 (11), 1155–1175.
Wu, J., Wu, Y., Niu, N., Zhou, M., 2021. MHCPDP: multi-source heterogeneous cross-project defect prediction via multi-source transfer learning and autoencoder. Softw. Qual. J. 29 (2), 405–430.
Xu, J., Wang, F., Ai, J., 2021a. Defect prediction with semantics and context features of codes based on graph representation learning. IEEE Trans. Reliab. 70 (2), 613–625.
Xu, Z., Li, S., Xu, J., Liu, J., Luo, X., Zhang, Y., . Tang, Y., 2019. LDFR: Learning deep feature representation for software defect prediction. J. Syst. Softw. 158, 110402.
Xu, Z., Zhao, K., Zhang, T., Fu, C., Yan, M., Xie, Z., . Catolino, G., 2021b. Effort-aware just-in-time bug prediction for mobile apps via cross-triplet deep feature embedding. IEEE Trans. Reliab..
Yadav, H.S., 2020. Increasing accuracy of software defect prediction using 1-dimensional CNN with SVM. In: 2020 IEEE International Conference for Innovation in Technology. INOCON, IEEE, pp. 1–6.
Yang, S., Yang, S., Fang, Z., Yu, X., Rui, L., Ma, Y., 2019. Fault prediction for software system in industrial internet: A deep learning algorithm via effective dimension reduction. In: Cyberspace Data and Intelligence, and Cyber-Living, Syndrome, and Health. Springer, Singapore, pp. 572–580.
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, pp. 17–26.
Yedida, R., Menzies, T., 2021. On the value of oversampling for deep learning in software defect prediction. IEEE Trans. Softw. Eng..
Young, S., Abdou, T., Bener, A., 2018. A replication study: just-in-time defect prediction with ensemble learning. In: Proceedings of the 6th International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering. pp. 42–47.
Yu, H., Sun, X., Zhou, Z., Fan, G., 2021a. A novel software defect prediction method based on hierarchical neural network. In: 2021 IEEE 45th Annual Computers, Software, and Applications Conference. COMPSAC, IEEE, pp. 366–375.
Yu, T.Y., Huang, C.Y., Fang, N.C., 2021b. Use of deep learning model with attention mechanism for software fault prediction. In: 2021 8th International Conference on Dependable Systems and their Applications. DSA, IEEE, pp. 161–171.
Zeng, Z., Zhang, Y., Zhang, H., Zhang, L., 2021. Deep just-in-time defect prediction: how far are we? In: Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis. pp. 427–438.
Zhang, N., Ying, S., Ding, W., Zhu, K., Zhu, D., 2021a. WGNCS: A robust hybrid cross-version defect model via multi-objective optimization and deep enhanced feature representation. Inform. Sci. 570, 545–576.
Zhang, N., Ying, S., Zhu, K., Zhu, D., 2021b. Software defect prediction based on stacked sparse denoising autoencoders and enhanced extreme learning machine. IET Softw. 16 (1), 29–47.
Zhang, T., Du, Q., Xu, J., Li, J., Li, X., 2020. Software defect prediction and localization with attention-based models and ensemble learning. In: 2020 27th Asia-Pacific Software Engineering Conference. APSEC, IEEE, pp. 81–90.
Zhang, X., Ben, K., Zeng, J., 2018. Cross-entropy: A new metric for software defect prediction. In: 2018 IEEE International Conference on Software Quality, Reliability and Security. QRS, IEEE, pp. 111–122.
Zhao, K., Xu, Z., Yan, M., Tang, Y., Fan, M., Catolino, G., 2021a. Just-in-time defect prediction for android apps via imbalanced deep learning model. In: Proceedings of the 36th Annual ACM Symposium on Applied Computing. pp. 1447–1454.
Zhao, K., Xu, Z., Yan, M., Xue, L., Li, W., Catolino, G., 2021b. A compositional model for effort-aware Just-In-Time defect prediction on android apps. IET Softw..
Zhao, L., Shang, Z., Zhao, L., Qin, A., Tang, Y.Y., 2018. Siamese dense neural network for software defect prediction with small data. IEEE Access 7, 7663–7677.
Zhao, L., Shang, Z., Zhao, L., Zhang, T., Tang, Y.Y., 2019. Software defect prediction via cost-sensitive Siamese parallel fully-connected neural networks. Neurocomputing 352, 64–74.
Zhou, T., Sun, X., Xia, X., Li, B., Chen, X., 2019. Improving defect prediction with deep forest. Inf. Softw. Technol. 114, 204–216.
Zhou, X., Lu, L., 2020. Defect prediction via LSTM based on sequence and tree structure. In: 2020 IEEE 20th International Conference on Software Quality, Reliability and Security. QRS, IEEE, pp. 366–373.
Zhu, K., Ying, S., Ding, W., Zhang, N., Zhu, D., 2021a. IVKMP: A robust data-driven heterogeneous defect model based on deep representation optimization learning. Inform. Sci. 583, 332–363.
Zhu, K., Ying, S., Zhang, N., Zhu, D., 2021b. Software defect prediction based on enhanced metaheuristic feature selection optimization and a hybrid deep neural network. J. Syst. Softw. 180, 111026.
Zhu, K., Zhang, N., Ying, S., Zhu, D., 2020. Within-project and cross-project just-in-time defect prediction based on denoising autoencoder and convolutional neural network. IET Softw. 14 (3), 185–195.
Zhu, Y., Yin, D., Gan, Y., Rui, L., Xia, G., 2019. Software defect prediction model based on stacked denoising auto-encoder. In: International Conference on Artificial Intelligence for Communications and Networks. Springer, Cham, pp. 18–27.

Görkem Giray is a software engineer, a researcher, and a part-time lecturer. He has been working in the industry for more than 20 years and pursuing an executive level position in recent years in a multinational company. In addition, he has been conducting research in software engineering and semantic web (knowledge graphs) and delivering software engineering courses at universities. Giray received a B.Sc. (1999) and a Ph.D. (2011) in computer engineering from Ege University. He holds an MBA degree (2001) from Koç University. More details can be found on his LinkedIn profile: https://www.linkedin.com/in/gorkemgiray/

Kwabena Ebo Bennin is an assistant-professor in the Information Technology Group at Wageningen University, The Netherlands. He has a Ph.D. in Computer Science from City University of Hong Kong (2018) and a BA (Hons) in Computer Science and Statistics from University of Ghana (2011). He conducts research in empirical Software Engineering with a focus on software quality and software analytics.

Ömer Köksal is a senior lead research engineer in ASELSAN Research Center, Artificial Intelligence and Information Technologies Department. He has more than 20 years of experience in information technology and software systems engineering. He had a Ph.D. degree from Wageningen University and Research in 2018. In this Ph.D. thesis, we researched the internet of things systems and software engineering. His current research concerns natural language processing, deep learning, the internet of things, and software engineering. He is the author of tens of academic research papers, technical reports, and book chapters in these fields. In addition, he is currently researching the development of artificial intelligence-based techniques and algorithms in the software engineering domain. He also participated in several national and international research projects with various large software companies. In addition, he has worked as a senior software team lead/architect in several mission-critical avionic, naval, and unmanned software systems. Further, he has vast experience in software development and architecting product line engineering, model-driven software engineering, aspect-oriented software engineering, distributed software development, and command and control software development.

Önder Babur is an assistant-professor (tenure-track) in the Information Technology Group at Wageningen University & Research, The Netherlands. He holds a Ph.D. from Eindhoven University of Technology, M.Sc. from RWTH Aachen, Germany and B.Sc. from METU, Turkey. He was employed as a post-doctoral researcher in the Software Engineering & Technology group at Eindhoven University of Technology from 2019 to 2021, with which he is still affiliated as a guest researcher. He has further experience as a software engineer in Germany and as a researcher in Spain. His main research interests lie in the fields of model-driven engineering, systems modeling, software analytics, AI for software engineering and empirical software engineering. Over the years, he has participated in a number of research projects on automotive software engineering, digitalization and industrial automation, precision agriculture and multiscale modeling. He has initiated and co-chaired the International Workshop on Analytics and Mining of Model Repositories, and the International Conference on Systems Modelling and Management. Around the main research lines, he has been collaborating with many international research groups and high tech companies.

Bedir Tekinerdogan, is full professor and chair of the Information Technology group at Wageningen University, The Netherlands He has more than 25 years of experience in software/systems engineering and is the author of more than 400 peer-reviewed scientific papers. He has been active in dozens of research and consultancy projects with various large software companies, whereby he has worked as a principal researcher and leading software/system architect in various domains. His research interests include software and systems engineering, system of systems engineering, data science, and artificial intelligence. More details can be found on his LinkedIn Profile: https://www.linkedin.com/in/bedir