Machine/Deep Learning for Software Engineering: A Systematic Literature Review

Simin Wang, Liguo Huang, Amiao Gao, Jidong Ge, Tengfei Zhang, Haitao Feng, Ishna Satyarth, Ming Li, He Zhang, and Vincent Ng

Abstract—Since 2009, the deep learning revolution, which was triggered by the introduction of ImageNet, has stimulated the synergy between Software Engineering (SE) and Machine Learning (ML)/Deep Learning (DL). Meanwhile, critical reviews have emerged that suggest that ML/DL should be used cautiously. To improve the applicability and generalizability of ML/DL-related SE studies, we conducted a 12-year Systematic Literature Review (SLR) on 1,428 ML/DL-related SE papers published between 2009 and 2020. Our trend analysis demonstrated the impacts that ML/DL brought to SE. We examined the complexity of applying ML/DL solutions to SE problems and how such complexity led to issues concerning the reproducibility and replicability of ML/DL studies in SE. Specifically, we investigated how ML and DL differ in data preprocessing, model training, and evaluation when applied to SE tasks, and what details need to be provided to ensure that a study can be reproduced or replicated. By categorizing the rationales behind the selection of ML/DL techniques into five themes, we analyzed how model performance, robustness, interpretability, complexity, and data simplicity affected the choices of ML/DL models.

Index Terms—Software engineering, machine learning, deep learning

1 Introduction

The software development and evolution paradigm has shifted from human experience-based to data-driven decision making. In the past, the state of software intelligence in Software Engineering (SE) tasks has been very rudimentary, with many of the decisions supported by gut feeling and at best through consultation with senior developers [1]. For instance, managers allocate development and testing resources based on their experience in previous projects and their intuition about the complexity of the new project relative to previous projects. This decision-making process leads to wasted resources and increased costs of building and maintaining large complex software systems. The primary reason is that SE data is more complex for its size than perhaps any other human construct, and many of the classical problems of developing software products derive from this essential complexity and its nonlinear increases with size [2].

Data plays an essential role in modern software development because what is hidden in the data and the relations among the data instances is information about the quality of software and services and the dynamics of software development and evolution. SE data, such as code bases, execution traces, historical code changes, mailing lists, forum discussion, and bug/issue reports, contain a wealth of information about a project's progress and evolution [3]. Although many traditional automated SE methods and tools were developed to assist human experience-based decision making and improve productivity in various SE tasks such as requirements traceability management, design specification, test data generation, defect tracking, cost estimation, etc., they focused on automating the generation, storage and management of the data localized and isolated in a specific SE task. However, these methods and tools could not reveal the deep semantics behind the data or the latent relationships among various kinds of data, which contained valuable information mentioned above to inform and impact the software project decision making process, especially under uncertainty. With the enhanced capabilities of Machine Learning (ML)/Deep Learning (DL) algorithms,[1] ML/DL models have been trained to undertake structured analysis of big data software repositories to discover patterns and novel information clusters and perform the systematic and continuous evaluation and integration of these data in neural networks. This allows for better

- *Simin Wang, Liguo Huang, Amiao Gao, and Ishna Satyarth are with the Department of Computer Science, Southern Methodist University, Dallas, TX 75275 USA. E-mail: {siminw, amiaog, isatyarth}@smu.edu, lghuang@lyle.smu.edu.*
- *Jidong Ge, Tengfei Zhang, Haitao Feng, Ming Li, and He Zhang are with the Nanjing University, Nanjing, Jiangsu 210093, China. E-mail: {gjd, hezhang}@nju.edu.cn, terryzhang1009@foxmail.com, fenghaitaofht@gmail.com, lim@lamda.nju.edu.cn.*
- *Vincent Ng is with the Human Language Technology Research Institute, University of Texas at Dallas, Richardson, TX 75083 USA. E-mail: vince@hlt.utdallas.edu.*

Manuscript received 1 January 2021; revised 1 April 2022; accepted 18 April 2022. Date of publication 10 May 2022; date of current version 15 March 2023.
This work was supported in part by NSF under Grant 2034508, in part by NSF of Jiangsu Province under Grant BK20201250, and in part by The Oversea Open Research Funding State Key Laboratory for Novel Software Technology, Nanjing University.
(Corresponding authors: Liguo Huang, Jidong Ge, Vincent Ng and He Zhang.)
Recommended for acceptance by H. Rajan.
Digital Object Identifier no. 10.1109/TSE.2022.3173346

[1] The term "machine learning" is an "umbrella" term that covers many kinds of models including deep learning models. For ease of exposition, however, we will use the term "ML" to refer to those studies that involve canonical/traditional machine learning and the term "DL" to refer to those studies that involve deep learning in this paper.

understanding of the deep semantics and inter-connections of the data using statistical and probabilistic routines [4] to generate comprehensive and systematic information and decision frameworks [5]. ML/DL techniques can automatically analyze and crosslink the rich data available in software repositories to uncover interesting and actionable information about software systems and projects, which is not achievable only through practitioners' intuition and experience. Moreover, with the rapid increase in the size and complexity of SE data, ML methods have found their way into the automation of SE tasks.

Underlying the popularity of ML/DL to represent and analyze data is the fact that a number of SE problems can naturally be formulated as data analysis (learning) tasks [6], including (1) classification tasks, where the goal is to classify a data instance into one of a predefined set of categories; (2) ranking tasks, where the goal is to induce a ranking over a set of data instances; (3) regression tasks, where the goal is to assign a real value to a data instance; and (4) generation tasks, where the goal is to produce a (typically short) natural language description as output. For example, binary defect prediction, which predicts whether new instances of code regions (e.g., files, changes, and methods) contain defects, can be naturally cast as a classification task. Code search [7], defect localization [8], bug assignment [9], pull requests/requirements/reports/test case prioritization [10], [11], [12], [13] and recommendation in software crowdsourcing [14] can be cast as ranking tasks. Continuous data are also utilized by SE researchers using regression models to estimate (1) the effort required to develop a software system [15], (2) the number of defects [16] and bug-fixing time [17], (3) performance of configurable software [18], (4) energy consumption [19], and (5) software reliability [18], which is a time series forecasting problem. Finally, code summarization [20], which provides a high level natural language description of the function performed by code, as well as the generation of well-formed code [21] and code artifacts (e.g., code comments) [22], have been reformulated as generation tasks.

The goals of this Systematic Literature Review (SLR) are three-fold. First, given the extensive use of ML/DL in SE, we believe the time is ripe to take a step back and examine the unique impacts of ML and DL on different kinds of SE tasks. In particular, we examine whether ML or DL techniques are more popularly used for a given category of SE tasks and analyze the circumstances in which one would prefer applying DL to ML (or vice versa) in SE. Second, we examine the complexity of applying ML/DL solutions to SE problems. Specifically, ML/DL applications to a particular task typically require specifying how the data are preprocessed and represented, and how the model is trained and evaluated. A proper understanding of these issues is crucial, as missing details in any areas above would result in a study that suffers from replicability and reproducibility problems. Consequently, we examine each of these issues, and in particular, we discuss the issues commonly shared by ML and DL and how they are different in terms of data representation and model design/training. Finally, given the plethora of ML/DL algorithms, it is essential to understand the rationales behind a researcher's choice of a particular ML/DL algorithm given a SE task. As noted above, there are task-specific circumstances in which we would prefer a particular learner over the others. There are also task-independent issues that researchers may take into account when determining which models to use (e.g., whether it is easy to interpret a model's decision). Therefore, we investigate the rationales commonly provided by SE researchers.

In sum, this study makes the following contributions:

- To the best of our knowledge, we are the first to carry out a comprehensive SLR on 1,428 papers published in the last twelve years. We demonstrated the unique impacts that ML and DL techniques each have on SE tasks and summarized some guidelines on the use of ML or DL for a given SE task.
- We examined the complexity of applying ML/DL solutions to SE problems and how such complexity has led to issues concerning the reproducibility and replicability of ML/DL studies in SE.
- By categorizing the rationales behind the selection of ML/DL techniques into five themes, we analyzed how model performance, robustness, interpretability, complexity, and data simplicity affected choices.

Paper Organization. The remainder of this paper is organized as follows. Section 2 introduces the background information concerning the ML and DL techniques adopted in SE. Section 3 presents our research methodology for identifying relevant studies and extracting (and synthesizing) related information for this SLR. Section 4 discusses the results of our research questions in detail. Section 5 presents the summary of findings, actionable implications and future work. The limitation of our SLR is discussed in Section 6. Section 7 presents the related work. Finally, Section 8 concludes this SLR.

2 Machine/Deep Learning: Preliminaries

There have been multiple efforts in academia to exploit the advantages of ML/DL to help solve various problems in SE tasks (illustrated in Section 1) in the past decade. This section describes these applied ML and DL technologies that will be mentioned in the rest of this paper. We provided an overview of the basic terminologies in machine learning (Section 2.1) and deep learning (Section 2.2).

2.1 Machine Learning

The origin of ML can be traced back to 1959 by Arthur Samuel [23]. A widely quoted, more formal definition of ML was proposed by Tom M. Mitchell: "a computer program is said to learn from experience E with respect to some class of tasks T and performance measure P if its performance at tasks in T, as measured by P, improves with experience E" [24]. The three main paradigms in ML include *supervised learning (SL)*, *unsupervised learning (UL)*, and *reinforcement learning (RL)*. In *SL*, the training data comprises examples along with their target values and its aim is to assign each input instance one of a finite number of discrete categories (classification) or a real value (regression) [25]. *UL* is often used to discover groups of similar examples within the data (clustering), where none of its training examples is labeled with the corresponding target value [25]. Finally, *RL* is concerned with the problem of finding suitable actions to take in a given situation in order to maximize a reward by interacting with the surrounding environment [25]. Deep learning is a branch of ML that has been rapidly expanding in the last decade and will be introduced further in the next section (Section 2.2).

According to the output of models, SL can be further divided into three categories: Classification-based, Regression-based, and Sequence-based.

- *Classification* predictive modeling is the task of approximating a mapping function (f) from input variables (X) to a discrete output variable (y) [26]. The output variables are often called labels or categories. The mapping function predicts the class or category for a given observation. Some common SL classifiers are: Decision Trees (DT), Support Vector Machines (SVM), Naive Bayes (NB), Logistic Regression (LoR), K Nearest Neighbor (KNN), Neural Networks (NN), Random Forest (RF).
- *Regression* predictive modeling is the task of approximating a mapping function (f) from input variables (X) to a continuous output variable (y) [26]. A continuous output variable is a real-value, such as an integer or floating point value. These are often quantities, such as amounts and sizes. Some common regression models are: Linear Regression (LiR), Support Vector Regression (SVR), Stepwise Regression (SWR), Classification And Regression Trees (CART), Ridge Regression, and Artificial Neural Network (ANN).
- *Sequence* generative modeling is the task of predicting what word/letter comes next. The current output is dependent on the previous input and the length of the input is not fixed. Most of the sequence models are based on DL, such as encoder-decoder framework, which is widely used in the SE community. Hidden Markov Models (HMMs) and Conditional Random Fields (CRFs) are two popular non-DL sequence models in SE.

There is some overlap between the algorithms for classification and regression. Some algorithms can be used for both classification and regression with minor modifications, such as DT and ANN. In certain cases, it is possible to convert a regression problem to a classification problem. For instance, linear regression could be converted into binary classification by setting appropriate thresholds: Given a threshold (k), if output (y) is larger than k, the document (d) would be labeled as 1; otherwise, d would be labeled as 0 [27].

In addition, some advanced ML approaches have been introduced to SE as well:

Ensemble learning. Rather than choosing one method, ensembles build multiple predictors, where estimates coming from different learners are combined through particular mechanisms, such as voting of individual learner estimates on the final prediction (the so-called majority voting) [28]. Bagging (Bootstrap Aggregating) and Boosting are among the most common approaches [29]. In Bagging, many solo methods are independently applied on different training samples, where each training sample is selected via bootstrap sampling with replacement. On the other hand, Boosting arranges solo methods sequentially: Each solo method pays more attention to the instances in which the previous method was unsuccessful.

Semi-supervised learning. In semi-supervised learning, the learning algorithm is fed a labeled subset of data used as the starting point in the construction of a model which classifies the remaining unlabeled data. *Self-training* is a standard semi-supervised learning method: It learns a classification model using all labeled training data and then takes the top unlabeled examples that the model is most certain of their labels. These unlabeled examples are then treated as labeled data and used along with existing labeled data to train a final prediction model [30].

Active learning. Active learning, which is another major approach for learning in the presence of a large number of unlabeled data points, aims to reduce annotation effort by selecting only the informative examples in the training set for manual annotation in an iterative fashion [31], so as to minimize the number of manually annotated examples needed to reach a certain level of performance. It assumes that the learner has some control over the data sampling process by allowing the learner to actively select and query the label of some informative unlabeled examples which, if the labels are known, may contribute the most for improving the prediction accuracy [32].

Transfer learning. Transfer learning is a form of machine learning that takes advantage of transferable knowledge from the source to learn an accurate, reliable, and less costly model for the target environment [33]. Rather than training a model from scratch, we can train a model that has been pretrained on a related task. Because of this ability to exploit knowledge from a related task, we may be able to reduce the amount of annotated training data needed (the so-called local data) for our task to reach a given level of performance. Researchers in transfer learning reported that using data from other projects can yield better predictors than just using local data. This is especially true when the local data is very scarce [34]. However, some studies [34], [35] warned that if predictors are always being updated based on the specifics of the new data, then those new predictors may suffer from overfitting. Such updates very commonly occur when newly constructed code modules are considered or when we learn using data from other newly available projects.

2.2 Deep Learning

A neural network is composed of many simple elements called "neurons" or "units." Neurons are connected together with weights on the connections so that they can process information collaboratively and store the information on these weights [36]. A collection of neurons, the so-called "layer," is operating together at a specific depth within a neural network. The first layer of the network is called the input layer which contains the raw training data, and the final layer is the output layer. The middle layer of neurons is called the hidden layer, because its values are not observed in the training set. Each layer can be viewed as creating an abstract representation of the output of the previous layer. Representation learning is a set of methods (one or more hidden layers) that allows a machine to be fed with raw data and to automatically discover the representations needed for detection or classification [37].

Deep learning methods are representation learning methods with multiple levels of representation (more than one hidden layer), obtained by composing simple but non-linear modules that each transform the representation at one level (starting with the raw input) into a representation at a higher, slightly more abstract level [37]. For example, the first hidden layer is responsible for creating a representation of the inputs. The second hidden layer is responsible for creating a representation of the output of the first hidden layer, and so on. Hence, the more layers there are in the network, the more abstract the resulting representation of the inputs will be. The resulting representation is typically passed to one or more so-called *dense* layers, which are typical feed-forward networks, before reaching the output layer. In addition, each node/unit in the network is associated with a non-linear activation function. Hence, a network with numerous layers of non-linear units will express highly non-linear, arbitrarily complex functions.

Four neural architectures that are commonly used in SE include: feed-forward neural networks (FNNs), deep belief networks (DBNs), recurrent neural networks (RNNs) and convolutional neural networks (CNNs). FNNs are a family of *acyclic* artificial neural networks with one or more hidden layers between the input and output layers that aim to represent high-level abstractions in data by using model architectures with multiple non-linear transformations [38]. In a FNN, data flows from one layer to the next without going backward, and the links between layers are one way in the forward direction. Since there are no backward links, a FNN does not have any memory: Once data passes through the network, it is forgotten, and it cannot be exploited (as historical context) to predict data items encountered at a later point in time. A DBN is a generative graphical model that uses a multi-level neural network to learn a hierarchical representation from training data that could reconstruct the semantic and content of input data with a high probability [39]. A CNN is especially well suited for image recognition and video analysis tasks because a CNN, which is inspired by the biological findings that the mammal's visual cortex has small regions of cells that are sensitive to specific features of the visual receptive field [40], can exploit the strong spatially local correlation present in images. A RNN, unlike a FNN, allows backward links and, therefore, can remember things. It is therefore well suited for processing sequential data such as text and audio because its feedback mechanism simulates a "memory" so that a RNN's output is determined by both its current and prior inputs [41]. While a RNN has memory, its memory may not be that good. Specifically, it may only remember things in the recent past and not those it saw a while ago due to a problem known as vanishing gradient. To address this problem with the standard RNN model, two variants are widely adopted in SE with mechanisms for capturing long-term dependencies: Long Short Term Memory (LSTM) networks [42], [43] and Gated Recurrent Units (GRUs) [44], [45].

3 Research Method

This SLR was initiated in the middle of 2018, following the approach proposed by Kitchenham and Charters [46] that uses database searches to identify relevant studies based on a rigorous research strategy. We also considered the snowballing approach but decided not to use it in the end (see Appendix A.1, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TSE.2022.3173346).

To avoid missing critical papers, we adopted an enhanced version of Kitchenham's approach, the **GQS** method proposed by Zhang *et al.* [47] and described in detail in Sections 3.2 and 3.3. The leading author of this study is a Ph.D. candidate whose research interest lies in employing ML/DL techniques to explore challenging SE tasks. The remaining co-authors and our supervisors have long-term experiences with either SE or ML/DL. This section describes the research methodology used for conducting this study.

3.1 Research Questions and Motivations

In 2009, the deep learning revolution — triggered by the introduction of ImageNet — has transformed AI research in both academia and industry [48]. At almost the same time, a leap in Nvidia's graphics processing units (GPUs) significantly reduced the computation time of DL algorithms. Both milestone accomplishments became the catalyst for the ML/DL boom in all applications, and SE has undoubtedly become one of the beneficiaries. Xie *et al.* [49] conducted an empirical study on data mining involving ML methods for SE in 2009, which described several challenges of mining SE data and proposed various algorithms to effectively mine sequences, graphs, and text from such data. The study projected to see the expansion of the scope of SE tasks that could benefit from data mining and ML methods and the range of SE data that can be mined and studied. Besides, the oldest active AI and ML repository on GitHub was created in 2009 [50]. The annual proportion of new repositories related to AI and ML gradually rose since then, until the "boom" in 2017. Considering all above factors, we set 2009 as the starting year for our search of publications when preparing a 12-year review that spans the period from 2009 to 2020.

Generalizing applications of ML/DL in SE remains a concern, which has been acknowledged by many studies [41], [51], [52], [53]. Specifically, research results of ML/DL studies in SE may not generalize and be applicable to other projects with different datasets, projects written in different languages, and projects from different domains and/or technology stacks [51]. The main purpose of this study is not to re-evaluate model performance but to investigate what is missing and how the missing information leads to the gap which impedes the application and generalization of ML/DL methods and results on SE tasks reported in the academic literature. Specifically, this empirical study aims to answer the following research questions:

RQ1. *What are the trends of impacts of ML and DL techniques on SE tasks from 2009 to 2020?*

RQ2. *How do ML and DL differ in data preprocessing, model training, and evaluation when applied to SE tasks, and what details need to be provided with respect to these three aspects to ensure that a study can be reproduced or replicated?*

RQ3. *How do SE studies select the ML/DL models?*

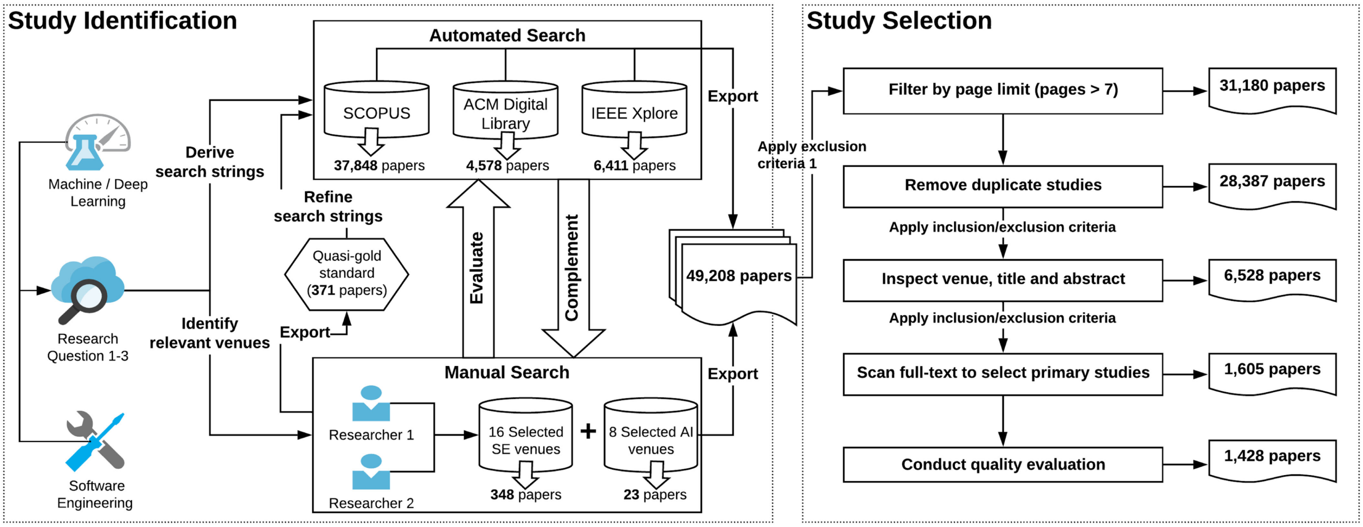


Fig. 1. Study identification and selection process.

RQ1 attempts to summarize the changes (accomplishments and deficiencies) we discovered as part of our trend analysis of ML- and DL-related SE studies. Through **RQ2** and **RQ3**, we hope to shed light on issues concerning how to improve the applicability and generalizability of ML/DL-related SE studies. Specifically, **RQ2** attempts to examine the complexity of applying ML/DL techniques to SE tasks. **RQ3** aims to find the patterns with respect to the choice of a ML/DL technique suitable for a particular SE task.

3.2 Search Strategy

As shown in Fig. 1, we applied the “Quasi-Gold Standard” (**QGS**) [47] method to construct a set of known studies for refining search strings by integrating manual and automated search strategies. We chose this search strategy because of the large number of relevant papers. It balances the search efficiency and the coverage of studies, which is much faster than purely manual search, and captures the most relevant studies following a relatively rigorous process. Specifically, we took six steps to identify relevant studies:

1) Select publication venues for manual search and select digital databases for automated search which can cover all selected venues.
2) Establish **QGS**: Screen all papers for manual search and filter by inclusion/exclusion criteria (defined in Table 2).
3) Subjectively define the search string based on domain knowledge.
4) Conduct automated search using the search string defined in Step 3.
5) Evaluate the quality of automated search against **QGS** by calculating *quasi-sensitivity*.
6) If *quasi-sensitivity* $\geq 80\%$, the results from the automated search can be merged with the **QGS** and move forward. Otherwise, the process has to go back to Step 3 for search string refinement, which forms an iterative improvement until the performance reaches the threshold.

As the manual search venues, we chose 16 top SE (ICSE, ASE, ESEC/FSE, ICSME, ICPC, RE, ESEM, ISSTA, MSR, SANER, TSE, TOSEM, EMSE, IST, JSS, JSEP) and eight AI (AAAI, IJCAI, ACL, ICML, AIJ, JMLR, EMNLP, CoNLL) conferences and journals that have published papers addressing the ML/DL applications in SE (shown in Table 1). Correspondingly, the follow-up databases for automated search are IEEE Xplore, ACM Digital Library and SCOPUS. Then, two authors independently screened the title-abstract-keywords fields of all the papers published in the selected venues from 2009 to 2020. Any disagreement on any of the identified papers was resolved via discussion after both of them examined the full text of the paper. In total, 371 papers (348 SE + 23 AI papers) were retrieved for building the **QGS**. The detailed process of refining the search string is described in Appendix A.2, available in the online supplemental material and the final search string is shown below:

“(‘machine learn’ OR ‘deep learning’ OR ‘neural network?’ OR ‘reinforcement learning’ OR ‘unsupervised learn*’ OR ‘supervised learn*’) AND (‘software engineering’ OR (software AND defect) OR ‘software requirement?’ OR ‘software design’ OR ‘software test*’ OR ‘software maintenance’ OR ‘source code’ OR ‘project management’ OR ‘software develop*’)”*[2]

Finally, we retrieved a total of 49,208 papers from three digital databases and via manual search. The automated search results can be directly downloaded as spreadsheets (.csv), containing paper titles, publication years, publication titles (venue), paper lengths (in pages), etc.

3.3 Study Selection

Once we retrieved the studies deemed potentially relevant based on our search strategy, an assessment of their actual relevance according to the inclusion and exclusion criteria in Table 2 was executed in order to select the primary studies that provide direct evidence about the research questions.

The selection procedure was performed in five phases as illustrated in Fig. 1. The first two phases (filtering and deduplication) were automatically processed by manipulating the

[2] An asterisk (*) in a search term is used to match zero or more characters, and a question mark (?) is used to match a single character.

TABLE 1
Publication Venues for Manual Search

Acronym	Venues
ICSE	International Conference on Software Engineering
ASE	International Conference on Automated Software Engineering
ESEC/FSE	European Software Engineering Conference and International Symposium on Foundations of Software Engineering
ICSME	International Conference on Software Maintenance and Evolution
ICPC	International Conference on Program Comprehension
ESEM	Symposium on Empirical Software Engineering and Measurement
RE	Requirements Engineering Conference
ISSTA	International Symposium on Testing and Analysis
MSR	Working Conference on Mining Software Repositories
SANER	International Conference on Software Analysis, Evolution and Reengineering
EMSE	Empirical Software Engineering
TSE	IEEE Transactions on Software Engineering
TOSEM	ACM Transactions on Software Engineering and Methodology
JSEP	Journal of Software: Evolution and Process
JSS	Journal of Systems and Software
IST	Information & Software Technology
AAAI	AAAI Conference on Artificial Intelligence
IJCAI	International Joint Conference on Artificial Intelligence
ACL	Meeting of the Association for Computational Linguistics
ICML	International Conference on Machine Learning
AIJ	Artificial Intelligence (Journal)
JMLR	Journal of Machine Learning Research
EMNLP	Empirical Methods in Natural Language Processing
CoNLL	Computational Natural Language Learning

TABLE 2
Study Inclusion and Exclusion Criteria

Inclusion criteria	
1)	The paper claims that a ML/DL technique is used
2)	The paper claims that the study involves an SE task or one or more topics covered in the field of SE [54]
3)	The paper with accessible full text
Exclusion criteria	
1)	The paper whose number of pages is less than 8
2)	The old version of the paper that has been extended from conference to journal
3)	The paper using SE methods to contribute to ML/DL systems
4)	The paper that is published as a SLR, review or survey
5)	Short papers, tool demos and editorials
6)	The paper that is published in a workshop or a doctoral symposium
7)	The paper that is a grey publication, e.g., a technical report or thesis

spreadsheet of search results, reducing the total number of papers to 28,387. Next, applying the inclusion/exclusion criteria to check the venue, paper title, and abstract, the total number of included papers declined substantially to 6,528 in the third phase. Unrelated topics to SE (not satisfying inclusion criteria 1) was the leading cause for the decline. We also removed 1,925 papers that were grey publications or were published in a workshop (exclusion criteria 5,6,7), 89 SLRs or mapping studies/surveys (exclusion criteria 4), 134 papers that focused on *SE for ML/DL* (exclusion criteria 3), and 24 papers that were old versions of extended papers (exclusion criteria 2). The *SE for ML/DL* papers, which apply SE methods to ML/DL systems[3], were not considered because our SLR focuses exclusively on *ML/DL for SE*, which concerns how SE tasks can be formulated as data analysis (learning) tasks and thus can be supported by ML and DL techniques (see Section 2) [6]. The differences between these two branches are: (1) in *ML/DL for SE* studies, the proposed methodologies were ML/DL-based technologies, but in *SE for ML/DL* studies, non-ML/DL based SE methods (e.g., software testing techniques, such as metamorphic testing [55]) were used; and (2) in *SE for ML/DL* studies, the used data was gathered from ML/DL systems, but *ML/DL for SE* studies explored various kinds of data generated in the software development and evolution lifecycle. To eliminate *SE for ML/DL* papers, we checked whether the proposed methodologies were non-ML/DL based in the method design of the paper and determined whether data was collected from ML/DL systems in the experimental design of the paper. Finally, before moving to the primary studies selection, a pilot was conducted in order to establish a homogeneous interpretation of the selection criteria between two researchers [56]:

1) Randomly select 30 studies from the current collection and assess them individually by full-text reading according to the inclusion/exclusion criteria.
2) Calculate the Cohen's Kappa value [57] after classifying all 30 studies as included or not.
3) Hold a discussion to resolve disagreement and strive to reach a consensus between two raters, if the Cohen's Kappa value did not reach *almost perfect agreement* (> 0.8) according to Landis and Koch [58].
4) Repeat Steps 1–3 by randomly selecting a new set of 30 studies until the Cohen's Kappa value is > 0.8.

Initially, two researchers had 25 agreements and five disagreements, which made the Cohen's Kappa value reach a *moderate agreement* (0.6). To resolve these disagreements, we had to determine (1) whether a paper that is a comparative study should be excluded or not and (2) whether a paper whose proposed methodology is statistical learning or data mining should be excluded or not. In the end, we agreed that a comparative study should be included as long as it

[3] In this paper, ML/DL systems refer to software frameworks, tools, or libraries that provide ML/DL functionalities, or software systems that have ML or DL models as their core with extra software encapsulation.

TABLE 3
Extracted Data Items and Related Research Questions

RQ	Data item
1	Published year
1,2,3	The category of SE task
1,2	The SE activity to which the SE task belongs
1,2,3	The adopted ML/DL techniques
2	Whether the dataset is from industry, an open source, or a student collection
1,2	The type of feature engineering and input data
2	The adopted data preprocessing techniques
2	The adopted hyper-parameter optimization techniques
2	The adopted weight training algorithms and optimizer
2	The selected evaluation metrics
1,2	The size of the data before and after preprocessing
2,3	The rationales behind ML/DL techniques selection
2	The number of times a study has been replicated or reproduced

conducted experiments and satisfied all inclusion criteria, and statistical learning or data mining should be excluded due to the exclusive focus on ML/DL applications in SE for this SLR. After another iteration, the Cohen's Kappa value increased to 0.9, which was our acceptable level to start the primary studies selection. Finally, 1,605 papers remained as primary studies by scanning the full-text in the final pool. The final number of included primary studies was reduced to 1,428 papers after conducting the quality assessment described in Section 3.5.

The above pilot process with Kappa measurement is also used for later data extraction, data synthesis, and study quality assessment with a slight difference in the number of selected sample studies or the contents to be assessed. Besides, our supervisors, the two domain experts in SE and ML/DL, provided their advice on "Hard to Determine" studies.

3.4 Data Extraction

Table 3 presents the data items extracted from the primary studies, where the column "RQ" shows the related research questions to be answered by the extracted data items on the right.

We distributed the workload among three researchers as follows: each researcher was given 2/3 of the studies in order to guarantee that all primary papers were assessed by at least two researchers [46]. All the information was recorded in spreadsheets via the data extraction form. By setting 20 sample studies for each iteration, the pilot process with Kappa measurement (stated in Section 3.3) was applied to each individual data item to check the data extraction consistency, during which the extracted data was cross-checked and disagreement was resolved by discussion or expert advice. After two iterations, the Cohen's Kappa value for extracting each data item exceeded 0.8, except three "Hard to Determine" items: the SE task, the rationale behind ML/DL techniques selection, and replicated/reproduced count. Then we decided to extract these three items separately, with the corresponding Kappa measurements documented:

- *SE task*: Initially, to determine the candidate term for a SE task addressed in each paper, we manually identified the keywords following the abstract or in the related work of the paper. Then, comparing with already identified SE tasks, we decided whether this candidate term should be merged with the existing SE tasks or kept as a new task. The Cohen's Kappa value increased to 0.9 within four iterations (0.4, 0.6, 0.7, 0.9, respectively).
- *Rationales behind ML/DL techniques selection*: We examined three places where the rationales are most likely explained (i.e., identifying the discussion of the suitability and advantages of the selected methods and why the selected model works better), including motivating examples, model design, and background. Three kinds of similar but noisy descriptions were excluded during the iterative process: the purpose of using ML/DL algorithms, the rationale of selecting non-ML/DL methods, and the introduction and definition of selected ML/DL algorithms. The Cohen's Kappa value increased to nearly 0.9 after five iterations (0.3, 0.5, 0.5, 0.7, 0.9, respectively).
- *Replicated/reproduced count*: First, we extracted titles of baseline studies replicated or reproduced by all 1,428 studies in our collection (this can typically be found in the section of experimental setup). Then, we checked each of the extracted paper titles and increased the count if the study was included in our collection. The Cohen's Kappa value increased to 0.9 within three iterations (0.5, 0.7, 0.9, respectively).

3.5 Study Quality Assessment

Based on the five roles that quality assessment (QA) may play in SLRs by Kitchenham *et al.* [59], the two main roles played in our SLR are as follows: (1) *Selection* — to provide more extensive inclusion and exclusion criteria, and (2) *Interpretation* — to guide the interpretation of findings and determine the strength of inferences.

With a *selection* purpose, QA was conducted before the main data extraction. The included primary studies assessed as low-quality were inadequate to answer the research questions and possibly biased the results [60]. Therefore, three QA criteria (QA1-QA3) were created to evaluate the full-text again for all 1,605 primary studies. Papers that elicited a *NO* answer to any of the following questions were excluded:

- **QA1.** Is ML/DL adopted in the proposed methodology and not simply used in one of the baseline methods?
- **QA2.** Is the impact of proposed ML/DL techniques on SE clearly stated?
- **QA3.** Is the contribution of the research clearly stated?

Following the process of Kappa measurement stated in Section 3.3, we successfully improved the Cohen's Kappa value from 0.7 to 0.9 within two iterations. As a result, we excluded 38, 86, 13 papers after being assessed based on QA1, QA2, QA3, respectively. During the assessment process, 40 additional papers were excluded by applying the

TABLE 4
Checklist of Questions to Assess the Quality of ML/DL Studies in SE

ID	Quality assessment question
QA4	Is the raw dataset retrieved from open source?
QA5	Are the data extraction methods fully described?
QA6	Does the paper describe any data preprocessing process?
QA7	Does the paper describe any data cleaning process?
QA8	Are the independent variables clearly reported?
QA9	Does the paper report how the proposed ML/DL model is implemented?
QA10	Are the process of determining the correct hyper-parameters for ML/DL models fully described?
QA11	Does the paper describe evaluation metrics when comparing to other approaches?
QA12	Is the proposed ML/DL model compared with other approaches?
QA13	Does the paper provide error analysis after the performance evaluation?

inclusion/exclusion criteria, thus reducing the total number of included primary studies from 1,605 to 1,428.

With an *interpretation* purpose, quality data could be collected along with data extraction process using separate spreadsheets. As shown in Table 4, a quality checklist (QA4-QA13) was designed to generate quality data which was then used as evidence to support part of the answers to **RQ2**, which is to assess the quality (replicability and reproducibility) of ML/DL studies in SE. The answers to these questions in the checklist and subsequent analysis were stated in Section 4.3.5 and Appendix A.5, available in the online supplemental material.

3.6 Data Synthesis

We applied both quantitative and qualitative methods to collate and summarize the results of the included primary studies. For **RQ1**, *meta-summary* — a quantitative synthesis which aims to identify the frequency of each discovery as well as the discovery of high frequent findings [61] — was used to construct a frequency matrix of different ML/DL methods that the applied to different SE tasks (presented in Table 6). Then we discovered several patterns based on the higher frequency of ML/DL applications, such as demonstrating both the spectrum and research depth of SE tasks on which ML/DL methods were applied for *ML for SE*. For **RQ2**, we used *narrative synthesis* [61] — a qualitative synthesis which features its defining characteristic that is summarized in narrative — to identify whether results from studies are consistent with one another. We prepared the required elements for *narrative synthesis*:

- *Theoretical base*: (1) Required elements to design a solution for SE tasks (i.e., Data source, Retrieval methodology, Raw dataset, Extraction methodology, Study parameters, Processed dataset, Analysis methodology, and Results dataset) from [62], and (2) nine-stage ML workflow activities from [63].
- *Preliminary synthesis*: Upon the theoretical base, we identified relevant information in terms of ML/DL in data preparation, model training, and evaluation based on the extracted data.
- *Relationship exploration and evaluation*: We analyzed this information to discover the patterns about commonalities and differences between ML and DL in data preparation, model training, and evaluation. We also designed a quality checklist (in Table 4) to evaluate the replicability/reproducibility of ML/DL studies in SE.

For **RQ3**, *thematic synthesis* [64] — a qualitative synthesis for identifying, analyzing and reporting recurring patterns in the data — was employed to investigate how well the authors were able to understand and justify the SE task being addressed by their selected ML/DL methods. Based on the collected rationales behind ML/DL techniques selection, two researchers took the following steps to synthesize the evidence:

1) Randomly select 10 rationales and label them by *vivo coding* [64], which refers to a code scheme extracted directly from the data record. This code scheme stands out as a summary of what is being stated and also serves as the basis for the subsequent clustering process. For instance, the code "used in many previous empirical studies" was extracted from "We selected random forest since this algorithm has been used in many previous empirical studies and tends to have good predictive power."
2) Compare the documented themes (vivo codes) and calculate the Cohen's Kappa value.
3) If the Cohen's Kappa value is ≤ 0.8, a discussion would be held to resolve the disagreement on the selected vivo codes.
4) Repeat Steps 1 to 3 until the Cohen's Kappa value is > 0.8.

After two iterations, the Cohen's Kappa value was improved from 0.6 to 0.8. In the end, we manually grouped all the rationales by clustering vivo codes into categories, which will be elaborated in Section 4.4.

4 Results and Synthesis

4.1 Overview

We selected 1,428 papers related to the applications of ML/DL to SE, with 1,209 ML and 358 DL studies (139 studies employ both ML and DL techniques), which are publicly available at [65]. These studies are from 70 conferences and 27 journals, covering varieties of domains besides SE and AI, such as data mining (*Data and Knowledge Engineering*) and system engineering (*International Systems Conference*). Still, Fig. 2 shows that SE researchers contribute the most papers since the top ten conferences and journals are all related to SE domains, and the full list of included venues is publicly available at [65]. Here are the venues presented in Fig. 2 that are not included in Table 1: PROMISE (*International Conference on Predictive Models and Data Analytics in Software Engineering*), APSEC (*Asia-Pacific Software Engineering Conference*), ASE_J (*Automated Software Engineering Journal*), SQJ (*Software Quality Journal*), and TR (*IEEE Transactions on Reliablity*).

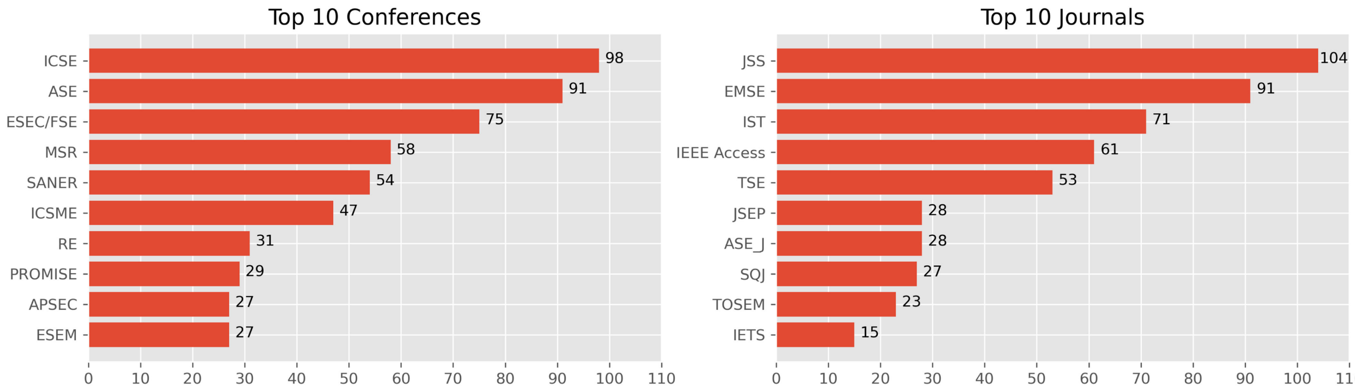


Fig. 2. Conferences and journals that published the largest number of papers in our study.

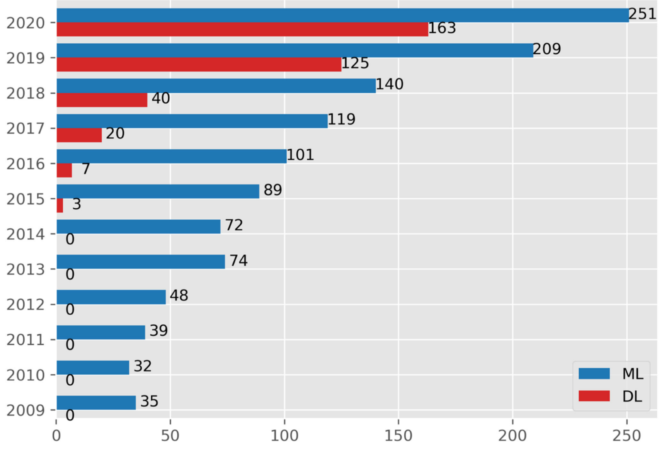


Fig. 3. Distribution of papers over years.

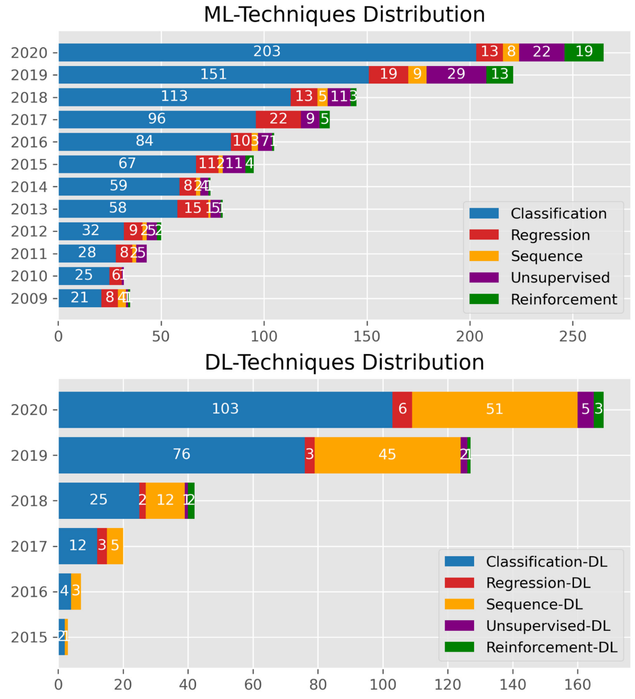


Fig. 4. Distribution of applied ML/DL techniques in SE over years.

Fig. 3 shows a significant increase in the number of papers published each year between 2009 and 2020. The blue bar shows the number of ML studies, and the red bar shows the number of DL studies. The initial application of DL to SE did not take place until 2015 when White *et al.* [66] introduced DL to software language modeling, which offered the SE community new ways to learn from source code files to support SE tasks. A possible explanation for this late date is that it took time for SE researchers to digest the DL techniques and cautiously validate their feasibility and effectiveness on SE tasks. Since 2015, DL has drawn increasing attention from researchers and practitioners in the SE community — especially in the past two years (2019-2020 in Fig. 3), tripling and quadrupling the 2018 number, respectively.

Fig. 4 shows several apparent trends with regard to the applications of five categories of ML and DL techniques in SE. The labeled number in white represents the number of papers that employed the techniques from each category per year.[4] First, for both ML and DL applications in SE, classification-based approaches are dominant and steadily increasing, indicating that classification tasks in SE are the priority concerns throughout the years. Second, unsupervised and reinforcement learning are more widely adopted in ML applications in the last two years. Third, the significant increase of deep sequence-based approaches reveals one of the greatest contributions by DL techniques. In the next section, we conduct a further analysis among SE tasks and ML/DL techniques.

4.2 Trend Analysis for *ML/DL for SE* (RQ1)

We analyzed the impact of *ML for SE* and *DL for SE*. Based on how ML/DL-based techniques generate impacts on diverse SE tasks, we categorized the 1,428 studies into a total of 77 distinct SE tasks over the aforementioned seven SE activities in Table 5. More detailed information is presented in Tables A.6 and A.7 (see Appendix A, available in the online supplemental material), which show the number of ML-based studies and the number of DL-based studies that were published in each year between 2009 and 2020 respectively for different SE tasks. Among these activities, *defect analysis* (349 ML+DL) and *software maintenance and evolution* (504 ML+DL) take up over half of the collection, mainly because significant amounts of bug reports and software evolution histories are publicly available in the open source repositories. According to the categories defined in Section 2.1, we classified both ML and DL studies into the same five categories for each SE task in Table 6. All the data is publicly available at [65].

[4] One paper might employ more than one category of techniques, so the labeled number in Fig. 4 might have the overlap among different categories.

TABLE 5
Distribution of 77 SE Tasks over Seven SE Activities

SE Activity	SE Task		Total
Requirements Engineering	R1.Requirements Tracing (9) R2.Requirements Detection and Classification (46) R3.Requirements Prioritization (4)	R4.User Story Detection (2) R5.Requirements Uncertainty/Inconsistency Detection (6) R6.Requirements Assessment (3)	70
Design and Modeling	D1.Architecture Tactics Detection (7) D2.Software Modeling (8) D3.Model Optimization (7) D4.Design Elements Management (14) D5.Design Pattern Detection (7)	D6.Architecture Evaluation (3) D7.Model Repair (1) D8.Model Extraction (3) D9.Design Discussion Mining (1)	51
Implementation	I1.Code Optimization (27) I2.Code Summarization (27) I3.API Learning (22) I4.Code Generation and Completion (26) I5.Code Search and Retrieval (19) I6.Program Synthesis (12)	I7.Code Smell/Anti-pattern Detection (32) I8.Program Classification (8) I9.Code Comments Management (14) I10.Error Specification Generation (1) I11.Type Inference (8) I12.Logging Statements Prediction (6)	202
Testing	T1.Test Case Generation (26) T2.Test Case Management (14) T3.Test Report Management (5)	T4.Test Automation and Prioritization (49) T5.Assert Statements Generation (2) T6.Runtime Verification (3)	99
Defect Analysis	A1.Defect Prediction (259) A2.Defect Detection and Localization (71) A3.Defect Categorization (15)	A4.Error Feedback Generation (2) A5.Root Cause Analysis (2)	349
Maintenance and Evolution	M1.Repository Mining (30) M2.Sentiment Analysis (29) M3.Code Clone and Similarity Detection (40) M4.Authorship Attribution (6) M5.Software Change Prediction (38) M6.Defect Fixing (25) M7.Software Applications Categorization (15) M8.Software Artifacts Classification (23) M9.Software Refactoring (5) M10.Software Quality Prediction (76) M11.Specification Mining (8) M12.Software Modularization (4) M13.Configuration Optimization (13) M14.Code Review (14)	M15.Tag Recommendation (7) M16.Traceability Recovery (14) M17.Report/Review Summarization (7) M18.Incident/Ticket Management (10) M19.Bug Report Management (44) M20.Bug Assignment (26) M21.Issue/Malware/Anomaly Detection (45) M22.Commits and Conflicts Management (7) M23.Pull Requests Management (6) M24.App Permission Recommendation (2) M25.Software localization (1) M26.Documentation Effort Prioritization (1) M27.App Usage Analytics (5) M28.Query Reformulation (3)	504
Project Management	P1.Software Effort/Cost Estimation (67) P2.Software Schedule Estimation (16) P3.Software Size Estimation (1) P4.Process Management (8) P5.Energy Estimation (6) P6.Risk Management (5)	P7.Performance Prediction (24) P8.Project Outcome Prediction (5) P9.Software Crowdsourcing Recommendation (6) P10.Community Smells Detection (1) P11.Developers' Behavior and Physiology Analysis (14)	153

The total number of relevant studies is shown in parentheses beside each SE task. "Total" shows the aggregated total number of papers for each SE activity.

4.2.1 ML for SE

ML applications cover almost all SE tasks (76 out of 77) except *Software Localization (M25)*. We investigated both the *breadth* and the *depth* of the changes that ML techniques have brought to SE. Of all the 77 SE tasks in Table 6, the total number of classification-based, regression-based and unsupervised ML studies is considerably larger than that of DL studies: 937 (C^M) versus 222 (C^D), 142 (R^M) versus 14 (R^D), 110 (U^M) versus 8 (U^D).

What impacts does the continuous and long-term use of ML approaches bring to SE tasks? According to Tables A.6 and A.7, we found that *Test Automation and Prioritization (T4)*, *Defect Prediction (A1)*, *Defect Detection and Localization (A2)*, *Software Quality Prediction (M10)*, *Bug Assignment (M20)*, and *Software Effort/Cost Estimation (P1)* are the six tasks that receive continuous contributions of ML applications from SE researchers every year between 2009 and 2020. Among these six tasks, *Defect Prediction (A1)* not only claims the top numbers of two columns (233 in C^M and 18 in U^M) in Table 6, but also contributes the largest number of papers (259) according to Table 5 (*depth*).

What different impacts do ML techniques bring to SE tasks in the last six years? In general, we observed a rapid growth in the number of SE tasks that can be supported and automated by ML techniques. We set 2015 as the line of demarcation to

TABLE 6
Number of ML and DL Techniques over 77 SE Tasks

Task (Yr)	C^M	C^D	R^M	R^D	S^M	S^D	U^M	U^D	F^M	F^D	Task (Yr)	C^M	C^D	R^M	R^D	S^M	S^D	U^M	U^D	F^M	F^D
R1 (10)	5	1	0	0	0	0	3	0	1	0	R4 (17)	1	0	0	0	0	1	0	0	0	0
R2 (09)	36	4	0	0	2	2	7	1	0	0	R5 (10)	4	1	0	0	1	0	1	0	0	0
R3 (09)	2	0	1	0	0	0	1	0	0	0	R6 (15)	3	0	0	0	0	0	0	0	0	0
D1 (16)	6	1	0	0	1	0	0	0	0	0	D6 (10)	0	0	1	0	0	0	1	0	1	0
D2 (09)	4	0	1	0	0	1	0	0	3	0	D7 (20)	0	0	0	0	0	0	0	0	1	0
D3 (13)	3	0	1	0	0	0	2	0	1	0	D8 (10)	3	0	0	0	0	0	0	0	0	0
D4 (18)	3	8	0	0	0	1	0	**2**	1	0	D9 (20)	1	1	0	0	0	0	0	0	0	0
D5 (12)	6	1	0	0	0	0	2	0	0	0											
I1 (11)	8	3	0	0	6	11	2	0	1	0	I7 (11)	29	7	1	0	0	0	0	0	0	0
I2 (15)	0	0	0	0	2	**27**	0	0	0	**3**	I8 (14)	4	2	0	0	0	0	2	0	0	0
I3 (09)	13	1	0	0	4	2	3	0	0	0	I9 (13)	10	2	0	0	0	3	0	0	0	0
I4 (09)	4	0	0	0	**7**	15	1	0	2	0	I10 (19)	1	0	0	0	0	0	0	0	0	0
I5 (11)	3	8	0	0	0	7	0	0	2	0	I11 (16)	3	2	0	0	0	3	1	0	0	0
I6 (14)	0	1	0	0	5	4	0	0	5	0	I12 (15)	4	2	0	0	0	0	0	0	0	0
T1 (11)	3	1	0	0	4	8	0	1	**8**	1	T4 (09)	30	4	5	0	0	1	7	0	7	2
T2 (09)	9	0	1	0	0	0	0	0	6	0	T5 (14)	1	1	0	0	0	0	0	0	0	0
T3 (16)	5	0	0	0	0	0	0	0	0	0	T6 (17)	2	0	0	0	0	1	0	0	0	0
A1 (09)	**233**	**35**	17	0	0	2	**18**	1	0	0	A4 (16)	0	1	0	0	0	0	1	0	0	0
A2 (09)	57	20	3	1	0	1	5	0	1	0	A5 (13)	2	0	0	0	0	0	0	0	0	0
A3 (09)	13	1	0	0	0	0	1	0	0	0											
M1 (13)	22	10	0	0	0	3	3	0	0	0	M15 (13)	5	4	0	0	0	0	0	0	0	0
M2 (14)	25	6	0	0	1	1	0	0	0	0	M16 (11)	13	3	0	0	0	0	1	0	0	0
M3 (11)	20	19	1	1	0	2	0	0	1	0	M17 (10)	5	0	0	0	0	1	3	**2**	0	0
M4 (13)	6	2	0	0	0	0	0	0	0	0	M18 (17)	8	5	0	0	0	0	0	0	0	0
M5 (11)	36	0	0	0	1	2	1	0	0	0	M19 (11)	37	12	0	0	0	1	2	0	1	0
M6 (12)	12	3	1	0	0	11	1	0	0	0	M20 (09)	24	3	0	0	0	0	2	0	0	0
M7 (09)	9	3	0	0	0	0	4	0	0	0	M21 (12)	30	12	1	0	0	0	11	1	1	0
M8 (09)	18	4	0	0	1	0	1	0	1	0	M22 (10)	7	0	0	0	0	0	0	0	0	0
M9 (17)	3	1	1	0	0	0	2	0	0	0	M23 (14)	6	0	0	0	0	0	0	0	0	0
M10 (09)	56	14	13	0	0	1	3	0	0	0	M24 (19)	2	0	0	0	0	0	0	0	0	0
M11 (13)	4	2	0	0	0	0	2	0	1	0	M25 (19)	0	0	0	0	0	1	0	0	0	0
M12 (12)	1	0	0	0	0	0	3	0	0	0	M26 (18)	1	0	0	0	0	0	0	0	0	0
M13 (14)	6	2	3	0	0	1	0	0	2	0	M27 (15)	3	0	1	0	0	0	1	0	0	0
M14 (13)	10	5	0	0	0	2	1	0	0	0	M28 (13)	2	0	0	0	0	1	0	0	0	0
P1 (09)	4	0	**63**	**4**	0	0	4	0	0	0	P7 (09)	9	1	13	3	0	0	1	0	1	0
P2 (09)	11	0	6	1	1	0	1	0	0	0	P8 (13)	4	0	1	1	0	0	0	0	0	0
P3 (17)	0	0	1	1	0	0	0	0	0	0	P9 (16)	5	0	0	0	0	1	0	0	0	0
P4 (10)	5	0	0	0	0	0	3	0	0	0	P10 (20)	1	0	0	0	0	0	0	0	0	0
P5 (14)	0	0	5	1	0	0	1	0	1	0	P11 (15)	13	2	1	1	1	0	2	0	0	0
P6 (10)	4	1	0	0	1	0	0	0	0	0											

The Task-ID *is directly mapped from Table 5. The columns* C^M,R^M,S^M,U^M,F^M *are ML-based and the columns* C^D,R^D,S^D,U^D,F^D *are DL-based, which represent the following categories: classification-based (C), regression-based (R), sequence-based (S), unsupervised (U), and reinforcement (F). The earliest year of each task that appeared in our collections is shown in parentheses beside each* Task-ID. *The red numbers indicate the largest number(s) of papers from the corresponding columns.*

compare studies published in two time periods, 2009-14 and 2015-20, as 2015 is the year in which DL applications started to become popular in the SE community. According to Table 6, 22 SE tasks experimented with different categories of ML methods after 2014. Based on a further analysis on 77 studies from these 22 tasks, we discovered two changes that ML techniques bring to SE (*breadth*).

First, *a larger variety of SE artifacts has been effectively analyzed using different ML techniques to improve the productivity of the development processes*, including user stories (R4), architecture tactics (D1), design discussions (D9), test reports (T3), incident reports or support tickets (M18), user interaction data (M27), crowdsourcing resources (P9) and developer interaction data (P11). For example, Bao *et al.* [67] used a Condition Random Field (CRF) sequence-based ML approach to infer a set of basic development activities in real world settings by analyzing developers' low-level actions and Girardi *et al.* [68] used six popular supervised classifiers (NB, KNN, DT, SVM, NN, RF) to predict developers' emotions based on biometric features during the programming tasks. Then these interaction data could be utilized for facilitating coordination between software developers by using SVM to recommend coordination needs to developers [69], determining whether a developer will leave the software team by using five supervised classifiers (NB, SVM, DT, KNN, and RF) to predict the turnover of software developers [70], and recommending suitable architectural expertise to support the design decision-making process by applying SVM to build an expert recommendation system [71].

Second, *word embedding techniques have begun to be integrated into different ML-based applications in SE after 2014, which can* *typically improve the performance in text and code based SE tasks*. Even though word embeddings are more frequently used in DL-based applications, ML-based applications in SE can also benefit from embeddings in that they can bridge the lexical gap by projecting natural language descriptions and code snippets as meaningful vectors in a shared representation space [72]. Among the 22 SE tasks, we found three classification-based applications in *Test Report Management (T3)* [73], *Incident/Ticket Management (M18)* [74], and *App Permission Recommendation (M24)* [75]. Two popular pre-trained embeddings are used for these applications: Word2Vec (trained on Google News [76]) and GloVe (trained on Wikipedia [77]). For instance, Liu *et al.* [75] developed a permission recommendation system based on KNN that recommends permissions for given apps according to their used APIs and API descriptions. To prepare API-API similarities for all training apps, they first mapped each word from API descriptions in Android documentation into the pre-trained GloVe embeddings and then computed the semantic similarities between APIs. In the end, they showed that their GloVe+KNN model outperforms NB and KNN-only models for the app permission recommendation. In addition, we identified more SE tasks that used word embeddings when we continued to investigate the remaining ML studies beyond the above 22 tasks, including sequence-based applications in *API Learning (I3)* and *Program Synthesis (I6)*, and unsupervised applications in *Repository Mining (M1)*. For instance, Ye *et al.* [78] solved a sequential labeling task through two steps: (1) using pre-trained Word2Vec embeddings with unsupervised algorithms (Brown Clustering and K-means) to learn word representations of unlabeled API-related information from Stack Overflow and (2) training a CRF on these word representations with a small set of human labeled sentences to classify each word as an API mention or a normal word. Apart from pre-trained Word2Vec and GloVe embeddings, Han *et al.* [79] developed a knowledge graph embedding which embeds software weakness and their relations in the knowledge graph into a semantic vector space, and Ye *et al.* trained [72] a SE-specific embedding on API documents, tutorials and reference documents.

What impacts does non-DL-based RL bring to SE tasks? According to Table 6, the major contribution of RL applications is in the testing domain. Specifically, RL is adopted to (1) help random input generators toward producing a diverse set of valid inputs (*T1*) by maximizing the number of unique valid inputs generated [80], (2) prioritize test cases (*T2*) by maximizing some predefined criteria such as additional code coverage or fault detection rate [81], and (3) automate the testing process (*T4*) by maximizing the number of execution paths in the shortest possible time [82]. In addition, as mentioned in Section 4.1, the number of RL applications was rapidly increasing in the last two years. Besides the steady increase in the testing domain, we observed a broader range of RL applications to SE tasks, such as automatically repairing software models (*D7*) by minimizing the model distance with respect to the original model [83] and synthesizing a program input grammar from a given set of seed inputs (*I6*) by maximizing the total number of constructed input accepted by the target program [84].

What SE tasks are addressed by ML techniques but not DL techniques? According to Tables 5 and 6, the following seven SE tasks were explored by at least 5 studies with only ML techniques: *Model optimization (D3)*, *Test Case Management (T2)*, *Test Report Management (T3)*, *Commits and Conflicts Management (M22)*, *Pull Requests Management (M23)*, *Process Management (P4)*, and *Software Crowdsourcing Recommendation (P9)*. It can be seen that the majority of these SE tasks are management-based, including classification (or categorization) [85], prioritization (or recommendation) [81], and quality assessment [86] of these SE artifacts. For instance, Hönel *et al.* [85] introduced source code density[5], which is incorporated into traditional hand-crafted features (e.g., keywords and comments) to classify commits based on size by using RF, achieving up to a 89% accuracy for the cross-project and a 93% accuracy for within-project commit classification. However, there may be other semantic features of textual descriptions that could be generated by deep neural networks to improve the performance of these SE tasks further [87].

4.2.2 DL for SE

Since 2015, SE researchers have demonstrated enormous interest in applying different categories of DL techniques to 59 of the 77 SE tasks (77%) across all seven SE activities, as shown in Tables A.6 and A.7. The majority of DL applications are classification-based and sequence-based, as shown in Fig. 4. Based on the 358 studies applying DL to SE problems, we observed the use of four basic types of deep learning networks (DNNs), namely feed-forward neural networks (FNNs), deep belief networks (DBNs), recurrent neural networks (RNNs) and convolutional neural networks (CNNs) (see Section 2.2 for an overview of these models). To further explore the impact of DL when integrated with SE, we made a comparative analysis of the SE tasks that employed both ML and DL techniques, and also investigated the unique contributions of DL studies in the most recent two years (2019-2020).

For classification-based SE tasks where ML techniques are dominant, what different impacts does DL bring to SE? According to the results in Table 6, we identified the 11 classification-based SE tasks that have the largest number of ML applications but have also used DL techniques. Table 7 presents the top three commonly used classifiers and all DL models for these 11 tasks. We found that (1) *the classification results could be improved by the replacement or combination of the hand-crafted features (required by ML) with representation learning (by DL); (2) the results are continuously improved by enhanced DL models; and (3) the generalizability to different presentation styles (unseen projects) could be improved by BERT (Bidirectional Encoder Representations from Transformers)*.

First, two examples of SE tasks are illustrated below to show the changes of feature representations for classification problems. Traditional supervised classifiers are the dominant solutions to software defect prediction and these classifiers were mainly trained on hand-crafted features, including code metrics (e.g., McCabe features [182]) and process metrics (e.g., change histories), which sometimes

[5] Source code density refers to the ratio of net size to gross size. Net size is the size of the unique code in the system and gross size is the size of everything, including clones, comments, and white-space.

TABLE 7
ML and DL Models Used in Selected Classification-Based SE Tasks

SE Task	Top-3 ML Classifiers	DL models
R2.Requirements Detection and Classification	(1) NB [88], (2) SVM [89], (3) RF [90]	(1) BiLSTM [89], (2) CNN [91], (3) CNN + BiLSTM [90], (4) BERT [88]
I7.Code Smell/Anti-pattern Detection	(1) RF [92], (2) NB [92], (3) DT [92]	(1) CNN [92], (2) Variational Auto-Encoder (VAE) [93], (3) CNN + RNN [94]
T4.Test Automation and Prioritization	(1) SVM [95], (2) DT [96], (3) RF [97]	(1) RNN/LSTM [95], (2) CNN [97], (3) DNN [97], (4) Deep Reinforcement Learning (DRL) [98], (5) RNN Encoder-Decoder [99]
A1.Defect Prediction	(1) NB [100], (2) RF [101], (3) LoR [102]	(1) RNN/LSTM/Tree-based LSTM (+supervised classifier) [43], (2) CNN/Graph-based CNN (+supervised classifier) [103], (3) DBN (+supervised classifier) [104], (4) DNN (+supervised classifier) [105], (5) Stacked Denoising Autoencoders (SDAEs) [106], (6) Deep Forest [107], (7) Graph Neural Network (GNN) [108], (8) Deep Adaptation Networks (DAN) [103]
A2.Defect Detection and Localization	(1) SVM [109], (2) DT [110], (3) NB [111],	(1) (Bi-)RNN/GRU/LSTM [112], (2) CNN/Tree-based CNN (+supervised classifier) [113], (3) DBN (+supervised classifier) [114], (4) DNN [115], (5) (Attention+) CNN + LSTM/GRU [116], (6) Knowledge Graph Embedding + Bi-Attention [117], (7) RNN Encoder-Decoder [118], (8) Tree-based CNN (TBCNN) [119], (9) Critic Neural Network [120], (10) Enhanced CNN [121]
M2.Sentiment Analysis	(1) SVM [122], (2) NB [122], (3) LoR [122]	(1) RNN [123], (2) Recursive Neural Tensor Network [124], (3) Attentional RNN Encoder-Decoder [125], (4) Text Attention+Audio Attention+CNN [126]
M3.Code Clone and Similarity Detection	(1) DT [127], (2) SVM [127], (3) NB [127]	(1) (Siamese) RNN GRU/LSTM/RNN (+supervised classifier) [128], (2) (Siamese) CNN/Tree-based CNN (+supervised classifier) [129], (3) (Siamese) DNN (+supervised classifier) [130], (4) RNN + Recursive Autoencoder + Graph Embedding [131], (5) Graph Neural Network [132]
M10.Software Quality Prediction	(1) RF [133], (2) SVM [133], (3) NB [133]	(1) (Bi-)LSTM/GRU (+supervised classifier) [134], (2) CNN (+supervised classifier) [135], (3) DNN [136], (4) CNN + RNN (+supervised classifier) [137], (5) RNN Encoder-Decoder [138], (6) Maximal Divergence Sequential Autoencoder [139], (7) Random Vector Functional Link network (RVFL) [140]
M19.Bug Report Management	(1) NB [141], (2) SVM [141], (3) RF [141]	(1) (Bi-)LSTM/GRU (+supervised classifier) [142], (2) (Siamese) CNN (+supervised classifier) [143], (3) DNN [144], (4) GNN [145], (5) CNN + BiLSTM [146], (6) Textual Encoder (BiLSTM) + Embedding + DNN (SABD) [147]
M20.Bug Assignment	(1) NB [148], (2) SVM [148], (3) KNN [148]	(1) CNN [149], (2) (Dual) DNN [150]
M21.Issue/Malware/Anomaly Detection	(1) SVM [151], (2) RF [151], (3) DT [151]	(1) (Siamese/Phased) LSTM + Attention (+supervised classifier) [152], (2) CNN (+supervised classifier) [153], (3) CNN + LSTM [154], (4) Deeplearning4j [155]

TABLE 8
ML and DL Models Used in Selected Generation-Based SE Tasks

SE Task	Top-3 ML Models	DL Generators
I2.Code Summarization	(1) Nearest Neighbor Generator [156]	(1) CODE-NN [157], (2) Attentional RNN Encoder-Decoder [158], (3) Code-RNN [22], (4) Graph Neural Network [159], (5) BERT + Encoder-Decoder + Transformer [160], (6) Attentional BiLSTM + CNN + TreeLSTM [161], (7) DRL [20], (8) Convolutional Attentional Model [162]
I4.Code Generation and Completion	(1) PCFGs and neuro-probabilistic language models [163], (2) HMM [164]	(1) Attentional RNN Encoder-Decoder [21], (2) RNN [165], (3) LSTM + Attention + Embedding + BERT [166], (4) Latent Predictor Networks [167], (5) Tree-based CNN [168], (6) RNN + GNN + LSTM + GRU (DIRE) [169], (7) Transformer [166]
T1.Test Case Generation	(1) RL [170]	(1) RNN [171], (2) Attentional RNN Encoder-Decoder [172], (3) Transformer [173], (4) Wasserstein generative adversarial networks (WGANs) [174], (5) DRL [175]
M6.Defect Fixing	None	(1) RNN [176], (2) Attentional RNN Encoder-Decoder [177], (3) BiRNN + Attention + GRU + GNN [178], (4) Tree-based RNN Encoder-Decoder + CNN [179], (5) Word2Vec + RNN + Recursive Autoencoders + K-Means [180], (6) BERT [181]

failed to capture the semantics of programs [183]. To address this problem, since 2015, DL-based approaches have been widely adopted to generate more expressive, complicated, and nonlinear features from the initial feature sets [184] or directly from source code [39], [185]. For instance, Wang *et al.* [104] leveraged DBN to automatically learn semantic features using token vectors extracted from the programs' Abstract Syntax Trees (AST) and fed them into ML classifiers (ADTree, NB, LoR) for file-level defect prediction. Their experimental results indicated the DBN-based semantic features can significantly improve the performance of within-project defect prediction against the ML-based 20 hand-crafted features by 2.1% to 41.9% in F1. In addition, Li *et al.* [186] built a defect prediction model consisting of CNN-based features with hand-crafted features, which performs better than models that use purely CNN-based features and purely hand-crafted features. *Code Clone and Similarity Detection (M3)* is another common SE task whose initial capability was limited to detecting only Type I-III clones based on the textual similarity computed from hand-crafted features. It was then augmented to spot all four types of clones using both textual and functional similarities through the source code representation learned via DL [128]. For instance, Li *et al.* [187] implemented the first solely token-based clone detection approach using a FNN, which effectively captured the similar token usage patterns of clones in the training data and detected nearly 20% more Strong Type 3 clones than ML approaches.

Second, besides ML classifiers and the four basic types of DL architectures, many studies from these 11 tasks currently involved some continuously enhanced DL models for classification tasks, as shown in Table 7. For defect prediction, most existing approaches started by exploiting the tree representations of programs — ASTs. They simply represented the abstract syntactic structure of source code but did not show the execution process of programs, so software metrics and AST features might not reveal different types of defects in programs. Phan *et al.* [188] formulated a directed graph-based convolutional neural network (DGCNN) over control flow graphs (CFGs) that indicate the step-by-step execution process of programs to automatically learn defect features. DGCNNs can treat large-scale graphs and process the complex information of vertices like CFGs, significantly outperforming baselines by 1.2% to 12.39% in terms of accuracy. For defect detection and localization, two enhanced DL approaches were proposed to improve the mean average precision (MAP) by around 5%. Specifically, since ML approaches ignored the semantic information in bug reports and code tokens in source files, while DL approaches ignored the structural information of both bug reports and source files [189], Xiao *et al.* [189] proposed CNN_Forest, a CNN and Random Forest-based approach where an ensemble of random forests is applied to detect the structural information from the source code and the alternate cascade forest works as the layer-structure in the CNN to learn the correlated relationships between bug reports and source files. Moreover, current studies using DL achieved poor performance and most improvements still came from Information Retrieval (IR) techniques (which focused more on textual similarity than semantics). In other words, the final results may still be heavily influenced by the performance of IR [190], meaning that the deep neural network in their model was more like a subsidiary. Xiao *et al.* proposed an enhanced model for bug localization, DeepLocator, which consists of a revised TF-IDuF (term frequency-user focused inverse document frequency) method, word2vec and an enhanced CNN by adding bug-fixing recency and frequency in the fully connected layer as two penalty terms to the cost function. DeepLocator correlated the bug reports to the corresponding buggy files rather than relying on the textual similarity used in IR-based approaches.

Third, unlike traditional word embedding techniques (summarized in Table 9) that produce fixed representations regardless of the context, BERT is pre-trained on large text corpora and produces word representations that are dynamically informed by the text [197], which has been proven useful for transfer learning in text processing tasks. For requirements detection and classification, state-of-the-art ML-based approaches usually use lexical and syntactic features, but their main problem is poor generalization, meaning that their performance drops when applied to unseen projects [88]. Taking advantage of BERT's fine tuning mechanism on specific tasks by providing only a small amount of data, Hey *et al.* [88] investigated its performance on unseen projects and concluded that BERT performs better than NB and CNN for both functional and non-functional requirements classification.

What benefits does DL bring to the SE tasks that ML techniques were not capable of tackling? In Table 6, we found three tasks where the number of studies using DL techniques is considerably larger than that using ML techniques, namely *Design Elements Management (D4)*, *Code Summarization (I2)*, and *Code Search and Retrieval (I5)*. We also found that the number of sequence-based DL applications is much larger than that of ML applications, especially in *Code Optimization (I1)*, *Code Summarization (I2)*, *Code Generation and Completion (I4)*, *Code Search and Retrieval (I5)*, *Test Case Generation (T1)*, and *Defect Fixing (M6)*. Given the above two findings, we conducted a further analysis and discovered two primary contributions for DL models.

First, for studies in *Code Search and Retrieval* and *Design Elements Management*, *through the application of CNNs, SE researchers have made significant progress in identifying and extracting elements embedded in multimedia (e.g., image, programming screenshots, video) artifacts, which has expanded SE data sources and given developers access to a richer set of documented data that was previously not leveraged*. For the task of extracting correct code appearing in video tutorials, existing approaches that applied Optical Character Recognition (OCR) techniques to software programming screencasts often result in a lot of noise (e.g., menus) being extracted with the source code [198]. Therefore, it is necessary to first accurately identify the section of the screen where the code is located and then apply OCR only to that section. With the powerful and accurate object recognition abilities through the use of filters, CNNs are currently the best choice to classify the presence or absence of code [199], remove non-code and noisy-code frames from programming screencasts [200], and predict the exact location of source code within each frame (code editing window) [198]. This accelerates code identification and code example resolution in video tutorials. For detecting Graphical User Interface (GUI) elements in GUI images, existing non-ML/DL methods are intrusive and require the support of accessibility APIs or runtime infrastructures that expose information about GUI

TABLE 9
Summary of Embedding Techniques and Their Advantages in DL Studies

Embedding	Advantages	Related Tasks
Word2Vec	Better express the similarity and analogy relationship between words. Two training modes: skip-gram and continuous bag of words (CBOW). The skip-gram model is concerned with using one word to predict the surrounding words, while CBOW model is concerned with using the surrounding words to predict the central word [191].	R1, R2, R4, R5, I1, I2, I4, I5, I6, I7, I9, I11, I12, T1, A1, A2, A4, M1, M2, M3, M6, M8, M10, M14, M15, M16, M18, M19, M20, M21, P1, P2, P9
GloVe	Inspired by the word co-occurrence probability that may encode global information for words, this model can make up for the weakness of Word2Vec just using local word co-occurrence information [77].	I7, I9, A2, M20, M7, M8, M10, P1
FastText	As a derivative of Word2Vec, the advantage of this method is that in English words, the morphological similarity of prefixes or suffixes can be used to establish relationships between words [192].	I7
ELMo	To input sentences into a pre-trained language model in real time to get dynamic word vectors, which can effectively deal with polysemy [193].	I7, M8, M20
Code2Vec	An attention-based neural code embedding model developed to predict the semantic properties of code fragments [194], for instance, to predict method names.	I1, A1, A2, M6, M10
Doc2Vec	An unsupervised framework mostly used to learn continuous distributed vector representations of sentences, paragraphs and documents, regardless of their lengths [195].	A2, M6, M7, M10, M15, P1
CC2Vec	A specialized hierarchical attention neural network model which learns vector representations of code changes (i.e., patches) guided by the associated commit messages, which presents promising performance on commit message generation, bug fixing patch identification, and just-in-time defect prediction [196].	I1, M6

The Task-IDs *are in Table 5.*

elements within a GUI [201]. Borrowing mature pixel-based methods from the computer vision domain, some popular CNN-based object detection models have been adopted in SE, which can directly analyze the image of a GUI to support many design elements management tasks, such as GUI automation and testing [202], [203], GUI skeletons generation [40] and linting of GUI visual effects [204] in both Android and IOS. CNN-based object detection often involves two subtasks: (1) *Region detection* locates the bounding box that contains an object and (2) *region classification* determines the class of the object in the bounding box. Two popular object detection models used in SE studies are: (1) Faster R-CNN, which first computes an objectness score to determine whether it contains an object or not by a region proposal network (RPN) and then uses a CNN-based image classifier to determine the object class [205]; and (2) YOLO (You Only Look Once), which labels an image by seeding the whole image through a CNN once and predicting the positions and dimensions of the objects in an image [203].

Second, *with the help of sequence-to-sequence (*SEQ2SEQ*) deep generation models, code and text based generation tasks in SE have been tackled more effectively than before.* Table 8 shows the ML and DL models that have been used in four popular generation-based SE tasks, including *Code Summarization (I2)*, *Code Generation and Completion (I4)*, *Test Case Generation (T1)*, and *Defect Fixing (M6)*. As can be seen, the number of ML models that have been applied to these generation tasks is fairly limited, owing in part to the fact that there are a limited number of canonical ML models that can be used for generation. In fact, we found no attempt to apply canonical ML models to *M6*. In contrast, popular DL generators for generation-based SE tasks include SEQ2SEQ models, of which the encoder-decoder architecture is arguably the most popular. As an example, most of the existing code summarization methods learn the semantic representation of source codes based on statistical language models. However, a statistical language model (e.g., the n-gram model) has a major limitation: It predicts a word based on a fixed number of predecessor words [20]. Following the trend of employing different variations of the DL-based attentional encoder-decoder framework, recent studies have built a language model for natural language text and aligned the words in text with individual code tokens directly using an attention component. These DL studies can predict a word using preceding words that are farther away from it. In addition, two enhanced DL approaches were introduced that improved performance by around 20% and 10%, respectively, in terms of ROUGE.[6] First, Wan *et al.* [20] integrated RL into the attentional encoder-decoder framework to remove the biased assumption that decoders are trained to predict the next word by maximizing the likelihood of the next ground-truth word given the previous ground-truth word. Specifically, using deep reinforcement learning, one can generate text from scratch without relying on ground truth in the testing phase. Second, due to the fact that the attentional encoder-decoder framework does not exploit the code block representation vectors, Liang *et al.* [22] proposed a new RNN model called Code-RNN, which gets a vector representation of each code block and this vector contains rich semantics of a code block.

Is there any correlation among an SE task, the DL architectures that have been applied to the task and the data types that have been used to support the task? To answer this question, we enumerate for each SE task the DL architectures and the data types that have been used in Table A.12 (see Appendix

[6] The ROUGE score counts the number of overlapping units between a generated sentence and a target sentence [22].

A, available in the online supplemental material). Based on 358 DL studies over 59 SE tasks, we identified the following input data types:

- *Metrics values*: Calculated from a set of software metrics or measures (i.e., traditional metrics, object-oriented metrics, and process metrics) that provide some insights about the software system, such as the number of lines of source code and the number of possible execution paths in a method [101].
- *Code*: Processed by extracting different syntactic information from source code as features for a ML/DL model, such as variables (declaration), methods (method signature, method body), and method calls (the API invocation sequence) [39].
- *Text*: Processed based on the bag-of-words (BoW) model, where n-grams or some exquisitely designed patterns are typically extracted as features [206].
- *Image*: Comprised of pixels (e.g., RGB color or grayscale pixels) that are transformed from the raw images, video frames and screencasts [198].
- *Others*: E.g., stack/log/execution traces [207], PDF objects [171].

According to Table A.12, we found two general trends of applying different DL architectures in SE. First, CNN or RNN based architectures are applied to almost all 59 SE tasks (except *R5*, *A4*, *M28*, *P8*, and *P9*). For one reason, *Code* and *Text* are the dominant data types in SE studies, which cover 204 (57%) and 105 (29%) of 358 studies, respectively. Given a text document (or a piece of code) represented as a sequence of words (or code elements), the use of RNNs allows us to easily extract n-gram level features from the sequence. Besides, RNNs such as LSTMs will enable us to model long-distance dependencies. In contrast, the advantage of CNNs is that CNN layers can be stacked to extract hierarchical features and better model source code at different granularity levels (e.g., statements and functions) [208]. Many researchers attempt to combine RNNs and CNNs in their model when given text data as input, as the resulting model allows them to combine the best of both worlds. CNNs can capture local textual dependencies, particularly the dependencies among the words or code elements in an n-gram, whereas RNNs can capture long-distance dependencies. Second, whenever *images* are involved, CNNs are used. This explains the dominant usage of CNNs in *Design Elements Management (D4)* (see Table A.12): All the datasets for this task are image-based. As noted before, CNNs have become the de facto model for image processing, as they are adept at modeling spatial correlations and hence the spatial locality that are often crucial to object identification in images.

In addition, many types of CNNs and RNNs (LSTMs, GRUs) are specifically modified to fit SE tasks. In particular, CNNs and RNNs are commonly embedded with two types of architectures as shown in Table A.12: (1) Siamese and (2) Tree-based (i.e., TBCNN, Tree-LSTM, CNN-TreeLSTM). Siamese architectures contain two or more identical sub-neural networks, which are best suited for SE tasks where two objects must be compared in order to assess their similarity, such as *Code Clone and Similarity Detection (M3)* [45], [129], [130], [209], *Defect Detection and Localization (A2)* [210], and *Issue/Malware/Anomaly Detection (M21)* [211]. For instance, given two methods in code clone detection, the Siamese network (Siamese GRU) first maps them to the same feature space [45]. If they are not a clone pair, the network will adjust its parameters to make them less similar as training progresses. On the contrary, if they are a clone pair, the model parameters will be adjusted so that they will become more similar to each other, thus making it possible to detect semantic clones even if they are syntactically dissimilar. The key benefit brought by Siamese architectures is a reduction in the number of parameters: the weight parameters are shared within two identical sub-neural networks, so it requires fewer parameters than a plain architecture with the same number of layers [130]. Tree-based architectures are often adopted to tackle tree-structured data by sliding over an entire tree to capture subtree features and are especially suitable for SE tasks which require parsing code fragments into ASTs [168], [179], [212], [213], [214]. For instance, Tree-LSTM (Recursive Neural Network) exploits a tree-structured sequence to extract the features of a code snippet from its AST since the output of the root node will contain the feature information of all AST nodes, thus achieving node-level feature extraction [212]. In addition, as discussed earlier, two variants of CNN — Faster R-CNN, which embeds a region-proposal network, and YOLO, which divides images into a grid system — are widely utilized to predict the presence and location of GUI elements on image-based datasets.

What architectures have not yet been implemented for specific SE tasks? First, Tree-based CNNs/RNNs (discussed above) and Graph Neural Networks (GNN) are rarely applied to the tasks for which code-based datasets are scarce (e.g., requirements detection). Gated Graph Neural Networks (GGNNs), the most commonly-used type of GNNs in SE [132], [169], [178], use an iterative graph propagation method to learn the neural representation of nodes in a graph. Different from images and natural languages, graph data is much more complex. An image can be viewed as a set of pixels and a text can be viewed as a sequence of words. In contrast, in a graph, there are at least two types of information: nodes and the relationship between them (edges). Since ASTs and and graph information are the required inputs for Tree-based architectures and GNNs, *Code* is currently the most suitable data type to be transformed into the required formats. Second, while Generative Adversarial Networks (GANs) (i.e., two neural networks contest with each other in a zero-sum game framework) [215] are generally applied on image-based datasets [204] due to the abundance of data and their continuous nature, we observed very few GAN applications in the SE tasks that primarily assume as inputs *Code* and *Text* because applying GANs to discrete data (e.g., text) poses technically challenging issues that are not present in the continuous case (e.g., propagating gradients through discrete values) [216].

4.2.3 Novel ML/DL Applications in SE

As mentioned above, much recent research involved applying ML and DL techniques to novel SE artifacts. In this subsection, we will introduce some of these novel ML/DL applications.

Screencast analysis in SE. Screencasting is a technique for recording the computer or mobile screen output at a specific time interval [217]. Each screenshot is a screen image and is referred to as a frame in the screencast. Work on automatically analyzing screencasts (or screenshots) in SE can broadly be divided into two categories: (1) content detection and extraction [198], [201], and (2) video search and navigation [202], [217]. Content detection and extraction (e.g., code extraction and GUI elements detection), an active research topic in SE, is performed primarily through the application of CNNs, as discussed in Section 4.2.2. For video search and navigation, CNN models are usually developed to translate video recordings of app usages into replayable scenarios [202] and automate the recognition of developer actions in programming screencasts [217]. For instance, programming screencasts provide a direct record of both a developer's workflow actions and the application content involved in programming tasks (e.g., typing code, scrolling content, switching windows). Workflow actions in a programming screencast, if available, can significantly improve video search and navigation efficiency and enhance a user's learning experience [217]. In addition, they are a common content carrier for disseminating SE knowledge, such as seeing a developer's coding in action (e.g., how changes are made to source code step by step; how errors occur and how they are fixed), which can be more valuable than text-based tutorials [218]. Therefore, we expect more SE researchers to leverage ML/DL techniques to enhance the interactive learning experience of programming video tutorials.

Use of biometrics in SE. Biometric sensors are used to measure the link between emotions and physiological feedback (i.e., cognition, processes, and states) [219], and some of the most commonly used measures in SE can be divided into four categories: eye-related (e.g., *eye-tracking*), brain-related (e.g., EEG (*Electroencephalography*)), skin-related (e.g., EDA (*Electrodermal*)) and heart-related (e.g., BVP (*Blood Volume Pulse*)). The SE research community has begun to apply ML/DL techniques to study the relations between developers'/users' physiological feedback (as measured using these sensors) and several SE tasks, including *Developers' Behavior and Physiology Analysis (P11)* [68], *Code Review (M14)* [220], [221], *Program Classification (I8)* [219], and *Sentiment Analysis (M2)* [222]. For instance, based on the psychophysiological data recorded from a combination of *eye-tracking*, EDA, and EEG, Fritz *et al.* [219] applied NB to predict whether a code comprehension task is perceived as easy or difficult.

Keystrokes evaluation in code completion. Keystroke is the number of times a developer or user needs to type to complete a task such as completing a whole line of code. It is commonly used to evaluate ML/DL-based code completion systems in SE [164], [165], [223]. For instance, Han *et al.* [164] presented a HMM for abbreviation completion that is integrated with a new user interface for multiple-keyword completion. To evaluate time savings and keystroke savings, the time usage and the number of keystrokes needed in the Abbreviation Completion system were compared with those needed in a conventional code completion system in Eclipse, a popular Java development tool. If the code completion system can significantly reduce the keystrokes, coding can be more efficient.

Emojis in sentiment analysis. An emoji is a digital image that is added to a message in electronic communication in order to express a particular idea or feeling. Not only are emojis pervasive in social media, but they are also widely adopted in the communication of developers to express sentiment [224]. Given that the small amounts of annotated text data available for many SE tasks can cover only very limited expressions, Chen *et al.* [225] employed emotional emojis as noisy labels of sentiments and proposed *SEntiMoji*, a customized DL-based sentiment classifier that uses both Tweets and GitHub posts containing emojis to learn sentiment-aware representations for SE-related texts. These emoji-labeled posts not only supply the technical jargon, but also incorporate more general sentiment patterns shared across domains.

4.2.4 Novel ML/DL Models for SE Applications

It is worth noting that some ML/DL models were specifically developed for the SE domain. In this subsection, we give an overview of some of these models.

Pre-trained models of source code. One of the exciting developments in DL involves pre-training. Specifically, pre-training has revolutionized the way computational models are trained in the natural language processing (NLP) community [226]. For a long time, supervised learning has been the most successful natural language learning paradigm. The pioneers of the pre-training idea challenged this view by showing that a vast amount of general knowledge about language, including both linguistic and commonsense knowledge, can be acquired by (pre-)training a model in a *task-agnostic* manner using *self-supervised* learning tasks. Self-supervised learning tasks are NLP tasks for which the label associated with a training instance can be derived automatically from the text itself. Consider, for instance, one of the most well-known self-supervised learning tasks, Masked Language Modeling (MLM) [227]. Given a sequence of word tokens in which a certain percentage of tokens is masked randomly, the goal of MLM is to predict the masked tokens. As can be easily imagined, a model for MLM can therefore be trained on instances where each one is composed of a partially masked sequence of word tokens and the associated "class" value is the masked tokens themselves. Because no human annotation is needed, a model can be pre-trained on a very large amount of labeled data which is automatically generated, thereby acquiring a potentially vast amount of knowledge about language. A pre-trained model can then be optimized for a specific task by fine-tuning its parameters using task-specific labeled data in the standard supervised fashion.

A number of pre-trained models have been successfully developed and applied in NLP, including BERT [228]. GPT-2 [229], LNet [230], RoBERTa [231], ELECTRA [226], T5 [232], and BART [233]. These pre-trained models differ terms of (1) what is being pre-trained (e.g., the encoder, the decoder, or both), (2) the pre-training objectives (e.g., MLM), and (3) the dataset(s) used for pre-training (e.g., Wikipedia).

Inspired by the successes of pre-trained models in NLP, a number of pre-trained models of source code have been proposed and successfully applied to a range of SE tasks that involve code understanding and generation. Well-known pre-trained models of source code include SCELMo [234], CodeDisen [235], CuBERT [236], C-BERT [237], JavaBERT [238], CugLM [239], CodeBERT [240], OSCAR [241], GraphCodeBERT [242], SynCoBERT [243], GPT-C [244], DOBF [245], DeepDebug [246], T5-learning [247], PLBART [248], CoTexT [249], ProphetNet-Code [250], CodeT5 [251], TreeBERT [252], and SPT-Code [253]. Like the pre-trained models developed in NLP, pre-trained models of source code can also be distinguished by (1) what is being pre-trained; (2) the pre-training objectives, and (3) the datatsets used for pre-training. Unlike the pre-trained models developed in NLP, which assume primarily text (i.e., word sequences) and features as inputs, many pre-trained models of source code have been specifically designed to take as inputs not only source code, which is viewed as a token sequence, but also the natural language embedded in the code (e.g., documentation, variable names) as well as the code structure (e.g., ASTs, Data Flow Graphs (DFGs)). Given these additional input modalities, novel pre-training tasks have been specifically designed to acquire information from these input modalities. For instance, pre-training tasks such as Edge Prediction [242], which masks the edges connecting randomly selected nodes in DFGs and aims to predict the masked edges, and Node Order Prediction [252], which randomly changes the order of some nodes in the ASTs and aims to identify if a change occurs, allow a model to learn representations of the code structure. Moreover, there are pre-training tasks that allow a model to learn across input modalities and thus capture the relationships between different modalities. For instance, Bimodal Data Generation [251] aims to generate a natural language summary (if code is given) or code (if NL is given), and Tree MLM [252], which masks some terminal node/identifiers in AST/code on the encoder/decoder side, aims to generate a complete code sequence.

Other ML/DL models. In addition to pre-trained models of source code, there are several well-known ML/DL models specifically developed for the SE domain:

- *CO-PILOT (COllaborative Planning and reInforcement Learning On sub-Task curriculum)* [254] is a novel goal-conditioned RL technique where RL and planning can collaboratively learn from each other to overcome sparse reward and inefficient exploration in navigation and continuous control tasks.
- *DACE (Deep Automatic Code reviEw)* [255] is a novel DL model for automatic code review, which learns the revision features based on pairwise autoencoding and a context-enriched representation of source code.
- *TAG (Type Auxiliary Guiding)* [256] is a novel encoder-decoder framework for code comment generation, which consists of an adaptive *Type-associated encoder*, a *Type-restricted decoder*, and a hierarchical RL approach that jointly optimizes the operation selection and word selection stages.
- *HOPPITY* [257] is a novel DL model to detect and fix a broad range of bugs in Javascript programs via learning a sequence of graph transformations.
- *CNN Decoder* [168] is a grammar-based structural CNN for code generation, including tree-based convolution and pre-order convolution, whose information is further aggregated by dedicated attentive pooling layers.
- *MDSAE (Maximal Divergence Sequential Auto-Encoder)* [139] is a novel DL model for binary code vulnerability detection, which can work out representations of binary code in such a way that representations of vulnerable and non-vulnerable binaries are encouraged to be maximally different for vulnerability detection purposes, while still preserving crucial information inherent in the original binaries.

4.2.5 ML or DL?

Given the above discussion, it should be clear that for some SE tasks, ML has been predominantly used, while for other tasks, DL is the preferred approach. In general, ML and DL differ in terms of how to understand and represent data. The relevant question is: *How should we decide whether we should employ ML or DL for a given SE task?* Below we provide some guidelines that SE researchers can follow in their decision-making process. A detailed discussion of how SE studies select specific ML/DL models can be found in Section 4.4.

Feature Engineering. In canonical ML, given an input (be it an image, a text document, or a non-linear structure such as a graph), features will have to be manually designed and extracted from the input, and the resulting feature vectors will then be used to train a model. The success of canonical ML, therefore, depends heavily on the success of manual feature engineering. While some of these features are task-independent and can be computed automatically (e.g., n-gram features), others may be task-dependent and need to be designed by domain experts. In contrast, DL obviates the need for manual feature engineering. The input for a DL model can simply be the raw data, an image or a text document. During the model training process, a DL model will learn representations of the input that would be useful for the task. For instance, manual feature engineering is a challenging task for vulnerability severity prediction problem (*a Software Quality Prediction* task) because of the diversity of software vulnerabilities and the conciseness of vulnerability descriptions [135]. Software vulnerabilities are diverse in terms of the range of products from which vulnerabilities are discovered, the amount of vulnerability data for different products, and the mechanisms of how vulnerabilities work. Vulnerability descriptions are concise in that they are brief in form but comprehensive in scope, which results in a very high-dimensional and sparse feature space. To address this problem, Han *et al.* [135] design a CNN architecture to learn to extract and compose the most informative n-grams of vulnerability descriptions when mapping the meaning of individual words in a sentence to a continuous vector of the sentence. It removes the need for manual feature engineering and greatly reduces the need for adapting to other vulnerability rating systems. Hence, if complex, domain-dependent features are needed but a domain expert is either unavailable or too costly to hire, one may want to apply DL instead.

If one has the resources to perform manual feature engineering, it does not imply that one should use ML rather than DL despite the latter's ability to learn feature representations. The reason is that a DL model could be improved with hand-engineered features. We refer the reader to Section 4.3.1 and

Appendix A.3, available in the online supplemental material, for a further discussion of feature engineering.

Concept Complexity. If the target concept that the learner is supposed to learn is not particularly complex, ML may be the preferred choice; otherwise, it may be better to employ DL. For instance, mutation testing is widely recognized to be expensive due to the expensive mutant execution procedure [258]. However, a mutant has two alternative execution results — killed or alive — and thus Zhang *et al.* [258] simplified the prediction of mutant execution results as a binary classification problem solved by RF. As an extreme example, if the target concept can be represented by a linear function, we can simply train a SVM with a linear kernel. Using a model as complex as a DL model will lead to overfitting (discussed further in Section 4.3.2) even with regularization. In contrast, given the complexity of DL models, they can easily represent any function and should be used when the target concept is potentially complex. This guideline has been recognized in many ML/DL studies in SE, and will be further discussed in the theme "Simple Task/Data" in Section 4.4.

Amount of Labeled Training Data. The decision of whether to apply ML or DL is in part determined by the amount of annotated data available for model training. Typically, if labeled training data is abundant, then DL is the preferred choice; otherwise, ML may be the better choice. For example, to predict if spectrum-based fault localization (SBFL) is effective, Golagha *et al.* [259] did not consider any DL architectures but picked four ML classifiers (LoR, DT, RF, and SVM) because their dataset only consists of 341 instances. The reason is that the number of parameters of a DL model is typically much larger than that of a ML model. For instance, in an SVM, there is one weight parameter associated with each feature. Even when n-grams are used as features, it would be uncommon to see more than several millions of features. In contrast, it would be uncommon to see a DL model, specifically those that achieve state-of-the-art results on SE tasks, with less than several millions of parameters. The substantially larger number of parameters typically associated with a DL model implies that robust performance cannot be achieved unless the model is trained on a large amount of labeled data. Having said that, how much labeled data is needed for a given task depends on the complexity of the task.

The Need for Deep Semantic Understanding. If a deep semantic understanding of the input is needed, then DL is the preferred choice, but if a shallow understanding is sufficient for achieving good performance, then ML can be considered. For instance, most non-DL techniques lack the sophistication needed to reason about semantic associations between artifacts in requirements traceability and therefore fail to establish trace links when there is little meaningful overlap in the use of terms [41]. Guo *et al.* [41] utilized word embedding and RNN models to generate trace links because word embeddings represent knowledge of the domain corpus and RNN uses these word vectors to learn the sentence semantics of requirements artifacts. It is well-known that commonly used hand-crafted features, such as n-gram features, are lexical in nature and may only encode shallow semantic information. While semantic features can be designed to address this problem, computing such features may be difficult: It may require access to a knowledge base, and in addition, heuristics may need to be designed to extract information from the knowledge based, thus yielding noisy extractions.

One may argue that we can use word embeddings as inputs for ML models since word embedding techniques have begun to be integrated into different ML-based applications in SE as mentioned in Section 4.2.1, including pre-trained Word2Vec and GloVe [75], [78], and task-specific embedding trained on SE data [72]. These embeddings are typically trained on large text corpora and are reasonably good at encoding semantic information. The reason is that two semantically similar words are trained to have similar embeddings. However, these embeddings are typically trained in a context-independent manner, meaning that each word will only have one embedding, even when the word is polysemous (e.g., has multiple senses). In contrast, DL enables *contextualized* representations to be learned as part of the model training process. These representations are not only context-dependent but also task-specific. It is typically because of these automatically learned representations that allow a DL model to achieve good performance on a SE task.

The Need to Capture Spatial/Temporal Correlations. It is not uncommon to see that the input for learning a SE task involves an image (e.g., [201], [203], [205]), a text document (e.g., [206], [260]), or a non-linear structure such as a graph (e.g., [81], [261]). Even if it is easy to design features to encode such input, it may not be easy to design features to capture the spatial and/or temporal correlations that exist in the input. If it is important to encode spatial correlations in an image and long-distance dependencies in a text document, for instance, then DL may be preferred to ML, as CNNs and LSTMs/GRUs can naturally capture such correlations in images and text dcouments, respectively. To handle data instances that are structurally more complex than images and sequences, such as trees and graphs, which are commonly found in SE (e.g., control flow graphs, API call graphs, AST), one may employ new neural models such as tree LSTMs [262] and graph convolutional neural networks [263].

Classification versus Generation. While canonical ML models are good at classification tasks, they are by no means good at tasks that involve generating text. In principle, generation via canonical ML can be performed using sequence-based generative models such as HMMs, CRFs, and Probabilistic Context-Free Grammars (PCFGs). In practice, these generative models are weak at generating long sequences and words that are not seen in the training data. In SE, however, there are many tasks that involve text generation of long sequences involving words that are unseen with respect to the training data, such as bug report/reviews summarization [264] and code summarization [20], [22], whose goal is to generate a short natural language summary of a given input (e.g., a method). Traditional approaches to summarization/generation in SE do not rely on generative models. Instead, these approaches proceed in multiple steps, where one needs to first extract the relevant information from the input, and then the extracted elements are fed into some hand-crafted templates for generating the output. Such hand-crafted templates are needed because canonical ML approaches fail to provide a way to directly generate text output from a given input.

DL, on the other hand, provides an end-to-end framework (the so-called encoder-decoder framework) where a given input is being mapped to the desired output directly in a model. This framework enables so-called sequence-to-sequence (SEQ2SEQ) learning [265]. As the name suggests, this neural architecture takes a sequence as input and produces a sequence as output. Hence, it is a natural framework for generation tasks such as machine translation, where the input is a word sequence in the source language and the output is a word sequence in the target language, as well as summarization, where the input is a sequence of words or code elements and the output is a textual summary (i.e., a word sequence). The encoder-decoder framework can be improved using a mechanism known as *attention* [266]. Intuitively, attention aims to amplify the relevant information from the input and de-emphasize the not-so-relevant information from the input. Attention has been shown to be effective in improving the encoder-decoder framework and has been extensively used by SE researchers.

Interpretability. Despite the large amount of recent work on interpretability, it remains difficult to interpret the output of DL models. Hence, if interpretability is a key issue, then ML models that are easily interpretable, such as decision trees and SVMs with a linear kernel, can be used. We refer the reader to Section 4.4 for a detailed discussion of ML/DL interpretability.

4.2.6 Challenges with Applying ML/DL to SE

Given the above observations, next we will discuss the challenges that need to be addressed to better leverage ML and DL to improve the productivity of SE tasks.

Addressing the Data Annotation Bottleneck. This challenge is common to both the ML and DL applications to SE tasks. Training ML/DL models typically requires a large amount of annotated training data. This is in general a key issue in the application of ML/DL to SE tasks: Although, for certain SE tasks, labeling is inexpensive or even free because they are either directly recorded in a software artifact (e.g., predicting whether a bug will be closed, how long it will take to close it, and who will close it) or easy to compute/mine from software artifacts (e.g., fault prediction), for the majority of SE tasks (e.g., code summarization), this is not possible. Unfortunately, obtaining manually annotated data is time-consuming and labor-intensive.

To address the data annotation bottleneck, we recommend the development of new unsupervised and semi-supervised learning algorithms, which by definition have less reliance on labeled data than their supervised counterparts. Though the application of unsupervised and semi-supervised learning to SE has been somewhat successful, the resulting models typically do not offer the same level of performance as those trained in a fully supervised manner. The challenge, then, would be to design unsupervised and semi-supervised learners that can achieve similar levels of performance as their supervised counterparts. One option may be to inject domain-specific knowledge (about the target SE task) in the learning process, either as hard constraints that a clustering algorithm must satisfy, or as soft constraints by encoding such knowledge as features in the learning process. The additional challenge, then, would be to identify and accurately extract such potentially useful domain-specific features.

Another way we can address the data annotation bottleneck is to directly obtain annotated data. This can be achieved in a cost-effective manner such as active learning or crowdsourcing, the latter of which involve hiring human workers at a cheap rate for performing annotation tasks. While the use of active learning and crowdsourcing to obtain annotated data is not a new idea, crowdsourcing has so far been successfully applied to obtain annotated data for simple SE tasks [14]. It is well-known that training crowdsourced workers to produce high-quality annotated data for complex SE tasks remains a challenge for SE researchers. However, it is typically the complex tasks that require a lot of annotated data to train accurate models.

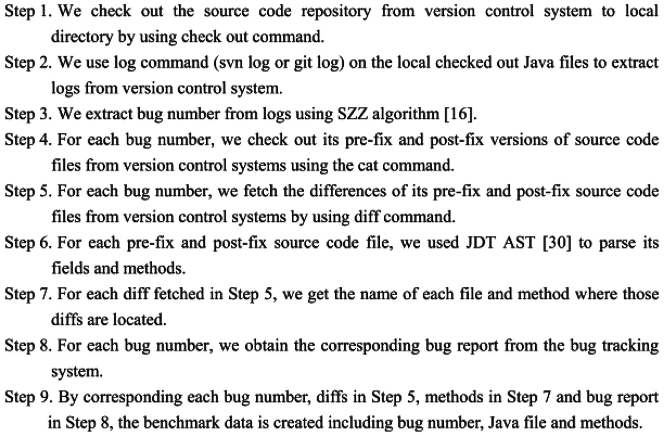
Improving Pre-Trained Models of Source Code. Despite the recent promising results achieved by pre-trained models of source code, the design of these models is still heavily influenced by the ideas developed in NLP and may therefore not yield optimal performances for SE tasks. For instance, these pre-trained models use the tokenization and embedding methods developed in NLP, such as SentencePiece and position embeddings. However, code is not exactly the same as natural language: It contains different types of lexical tokens such as variables, control symbols, and keywords. In addition, despite the fact that many pre-training tasks have been specifically designed to handle code characteristics (see Section 4.2.4), many of these SE-specific pre-training tasks still do not completely step outside the NLP mindset. IMLM, for example, is just a version of MLM that masks identifiers, and in fact, pre-training on IMLM can sometimes even yield worse results than pre-training on MLM [245]. We believe that the design of code-specific pre-training methods is currently limited in part by the NLP tokenization and embedding methods that are currently in use, and that a fundamental overhaul in the design of code-specific pre-training methods may involve designing code-specific tokenization and embedding methods.

4.3 Applying ML/DL to SE (RQ2)

How do ML and DL differ in data preprocessing, model training, and evaluation when applied to SE tasks, and what details need to be provided with respect to these three aspects to ensure that a study can be reproduced or replicated? Data preparation, model training, and evaluation are the core steps in applying ML/DL techniques to SE tasks. Missing details in any one of the three aforementioned areas would result in a study suffering from replicability or reproducibility. To answer the question above, based on all 1,428 papers, we will first summarize the patterns of how SE studies describe the details in the three steps in Sections 4.3.1, 4.3.2, and 4.3.3. Then we will conduct a comparative analysis to compare the purpose, the datasets, the applied preprocessing techniques, the tuning strategies, and the evaluation metrics for ML and DL applied to three popular SE tasks in Section 4.3.4. Finally, we will assess the replicability and reproducibility of the collected studies by checking whether they have provided detailed descriptions of these three steps according to Table 4.

4.3.1 Data Preparation

Data preparation involves three steps: (1) Identify the appropriate software repositories from which the raw



Step 1. We check out the source code repository from version control system to local directory by using check out command.

Step 2. We use log command (svn log or git log) on the local checked out Java files to extract logs from version control system.

Step 3. We extract bug number from logs using SZZ algorithm [16].

Step 4. For each bug number, we check out its pre-fix and post-fix versions of source code files from version control systems using the cat command.

Step 5. For each bug number, we fetch the differences of its pre-fix and post-fix source code files from version control systems by using diff command.

Step 6. For each pre-fix and post-fix source code file, we used JDT AST [30] to parse its fields and methods.

Step 7. For each diff fetched in Step 5, we get the name of each file and method where those diffs are located.

Step 8. For each bug number, we obtain the corresponding bug report from the bug tracking system.

Step 9. By corresponding each bug number, diffs in Step 5, methods in Step 7 and bug report in Step 8, the benchmark data is created including bug number, Java file and methods.

Fig. 5. The procedures for establishing benchmark data.

dataset(s) can be directly obtained, (2) extract the relevant data from the raw dataset(s), and (3) preprocess the extracted data to be ready for training. There are similarities as well as differences between ML and DL in terms of data preparation. Below we describe first the similarities and then the differences.

Data Source. We observed three types of data sources: open source (benchmark) datasets, industrial datasets, and student collections.[7] Open source or benchmark datasets are publicly available to everyone, while most industrial datasets and student collections were proprietary without public accessibility. Most ML and DL studies (nearly 90%) used open source repositories (e.g., GitHub [267]). In addition, 20 of the 70 studies (29%) in requirements engineering employed datasets from industry, which is the highest rate among the seven SE activities. Examples include extracting transaction functions from a financial software in a commercial bank [268], and generating trace links based on a Positive Train Control (PTC) domain, which is a communication-based train control system [41]. The source of a raw dataset (QA4) could have an impact on replicability [269] since the closed data source would hinder the replication process. Due to the confidentiality of a proprietary dataset, researchers cannot replicate these types of studies but can only reproduce the experiment on an open source dataset [270].

Data Extraction. Data extraction refers to the process of extracting and storing the relevant data from a raw dataset, usually implemented (totally or in part) with software tools and self-designed scripts [62]. The more detailed steps of data extraction provided by a paper, the fewer discrepancies between the regenerated data and the original data. For instance, Zhang *et al.* [271] provided the procedures, as shown in Fig. 5, to establish the benchmark data for method-level bug localization.

Although it is not mandatory, data annotation places an important role in supervised and unsupervised learning. Data annotation is the categorization and labeling of data for ML/DL applications, where training data must be properly organized and annotated for a specific use case. In this study, we observed four scenarios of data annotation: Researchers in a given study may choose to (1) annotate manually, (2) automatically generate labeled data, (3) use an open source dataset where labels are available, and/or (4) use annotated datasets from studies provided by the original authors. We found that the rates for all 1,428 papers that used Scenario 1 through Scenario 4 are about 18%, 18%, 66%, 17%, respectively. Though the rate of manual annotation is low, manually annotating over a thousand samples is sometimes a mandatory process when it comes to a SE task with new data or a completely new SE task, such as manual labeling of developer actions in programming screencasts [217]. According to the results, Scenario 3 has been applied to the majority of studies since labeling is inexpensive or even free for many SE tasks (both regression and classification tasks). This is because SE has repositories where labels can be mined directly, such as predicting whether a bug will be closed, how long it will take to close it, and who will close it (which appear directly in the repositories) [17], [272]. Scenario 2 is another way to reduce the cost of data labeling by creating an algorithm for generating labeled data. For instance, Heo *et al.* [109] automatically generated training data from an existing annotated codebase for the ML technique on anomaly detection. Scenario 4 often appears in comparative or replicated studies, which usually revisit other's work with the same datasets [273].

Data Preprocessing. Data preprocessing is indispensable for generating the final training data for ML/DL models because a dataset could contain various problems — such as inconsistencies, errors, out-of-range values, impossible data combinations, and missing values — making it unsuitable to start a ML/DL workflow [274]. Alternatively, datasets (especially industrial proprietary data and open source data) are usually different from each other, thus calling for extra caution when selecting appropriate types of data preprocessing methods that match a dataset. We observed four types of data preprocessing techniques widely used in ML/DL studies, as described below.

- Data cleaning (DC): DC is the process of either removing the noisy data (noise filtering, which includes stopword removal, stemming, and/or downcasing) or filling in the missing values (missing data imputation). Raw datasets are often noisy and contain outliers and missing values that can skew results [275]. Therefore, confidence in the prediction results by other researchers (who intend to replicate and reproduce) can be compromised where there is a lack of description about the data cleaning process [275], [276], [277]. Additionally, when the data cleaning process is stated, the availability of the data sizes before and after data cleaning [275] could also have an impact on replication because missing both or either of the data sizes may lead to the inconsistent sizes of the regenerated and original training data.
- Imbalanced Data Preprocessing (IDP): Training a classifier on an imbalanced dataset makes it biased toward the majority class label. This is due to the fact that the classifier tends to increase the overall accuracy, which results in ignoring minority class samples in the training set [278]. Hence, when a paper mentions that its used dataset is imbalanced, lacking

[7] A student collection is a dataset from an exclusive source, such as a student submission, a survey, or a field study.

TABLE 10
Data Preprocessing Techniques, Feature Engineering, and Input Data Types over 7 SE Activities

SE Activity	ML				DL			
	DC	IDP	FS	FC	DC	IDP	FS	FC
RE (66,9)	55%	15%	30%	18%	78%	44%	22%	22%
DM (41,15)	44%	10%	29%	17%	87%	13%	20%	13%
CO (119,100)	61%	16%	25%	19%	75%	5%	9%	9%
TE (82,20)	41%	11%	23%	22%	55%	15%	20%	25%
DA (325,61)	48%	32%	36%	26%	49%	43%	17%	36%
ME (427,137)	64%	16%	35%	26%	69%	9%	21%	18%
PM (149,16)	47%	7%	29%	35%	63%	6%	31%	38%

The total number of relevant ML and DL studies is shown in the parentheses besides each SE activity. For instance, "RE (66,9)" means that there are 66 ML studies and 9 DL studies in the RE domain. "DC", "IDP", "FS," and "FC" show the total number of studies (in percentage) that used these four data preprocessing techniques in each SE activity, respectively: Data Cleaning (DC), Imbalanced Data Preprocessing (IDP), Feature Selection (FS), and Feature Scaling (FC).

the description of how to overcome the imbalanced dataset problem may downgrade the credibility of the prediction results from the original studies and is also less useful for other researchers to replicate or reproduce.

- Feature Selection (FS): FS is the process of identifying and removing as much irrelevant and redundant information from a dataset as possible. A variant of feature selection is feature weighting. Rather than identifying and removing irrelevant features, feature weighting retains all the available features but assigns lower weights to those features that are determined to be less relevant to the task under consideration. A more advanced version of feature selection is dimensionality reduction, where high-dimensional data instances are projected into a low-dimensional space using techniques such as Principal Component Analysis [279]. For ease of exposition, we will henceforth refer to this collection of related techniques simply as "feature selection." If feature selection is employed in the studies, the level of details about the feature selection process may have an impact on replication. An incomplete description of the process may make it difficult to replicate the feature selection methods.
- Feature Scaling (FC): Also known as data normalization, FC is a method used to normalize the range of independent variables or features of data [280]. Since feature values extracted from different datasets often have varied scales, they are often normalized before further processing, which can improve prediction performance [281].

Based on all 1,428 papers, we further investigated the adopted data preprocessing techniques mentioned in these papers over seven SE activities: Requirements Engineering (RE), Design and Modeling (DM), Implementation (CO), Testing (TE), Defect Analysis (DA), Maintenance and Evolution (ME), and Project Management (PM). Results are shown in Table 10, and additional statistics over all 77 SE tasks are publicly available at [65].

What patterns did we observe by examining the data preprocessing techniques for ML and DL studies? According to Table 10, for data preprocessing techniques, we observed that ML and DL studies present a similar pattern in general. Data cleaning was mentioned in a large majority of papers for both ML and DL compared to the other three techniques for each SE activity. Specifically, we observed two common DC techniques: data filtering and missing data imputation. For data filtering, the most common way is to filter the raw dataset based on manually designed heuristics [267] or some common practices, such as stop words removal. On the other hand, we found that many studies in PM, typically for *Software Effort/Cost Estimation* tasks [282], [283], [284], have been conducted on handling missing data due to the fact that historical datasets used by these studies, such as the ISBSG (International Software Benchmarking Standard Group), contain a large amount of missing data caused by measurement noise or data corruption [285]. Missing data can be handled using either the embedded method (e.g., missing data toleration) or the independent method (e.g., Class Mean Imputation) [283].

For imbalanced dataset preprocessing, it is evident, as shown in Table 10, that more ML and DL studies from the DA domain provided the description of IDP techniques because in defect datasets there are fewer defective data instances than non-defective data instances [286]. Many techniques are proposed by SE researchers to address the imbalanced data challenge, e.g., Synthetic Minority Oversampling Technique (SMOTE) [100], [104], oversampling [286], undersampling [287], and cost-sensitive learning [258]. On the other hand, it is somewhat surprising to see that IDP is more frequently mentioned in DL studies than ML studies in RE. In particular, DL studies [88], [197] used undersampling techniques more often than oversampling techniques because the latter would increase the training set size and hence the training time.

Feature selection techniques are mentioned more often in ML than DL studies. A possible reason is that features are automatically learned by DL models, thus allowing them to ignore feature selection. Besides, we observed that feature selection is more often used by studies with *metrics value* based datasets. Because software metrics often have strong correlations among themselves (e.g., the widely used NASA datasets in the DA domain) and not all metrics are relevant to the proposed ML/DL models [288], many studies [130], [289], [290] used FS techniques to remove metrics that are correlated and irrelevant in order to improve model performance. For SE studies, filter-based and wrapper-based feature selection techniques are two commonly used automated feature selection techniques. Filter-based feature selection techniques search for the best subset of metrics according to an evaluation criterion regardless of model construction [288], including Information Gain (IG) [291], Correlation-based [292], and Chi-Squared-based [290]. Wrapper-based feature selection techniques use classification techniques to assess each subset of metrics and find the best subset of metrics according to an evaluation criterion [288], including Recursive Feature Elimination (RFE) [293] and Stepwise Regression [294].

Feature scaling is used more often in PM as shown in Table 10. This may imply that normalization has a positive impact on regression models as they are dominant approaches in *Software Effort/Cost Estimation* and

Performance Prediction. While the most common method is to normalize the values into the range from 0 to 1 in the data vectors [104], some studies adopted the *z-score* to normalize software metrics, which made the normalized software metric have a mean value of zero and a variance of one [280].

While ML and DL share the aforementioned commonalities in data preparation, there is a crucial difference between the two as far as data representation is concerned. As mentioned in Section 4.2.5, canonical ML approaches have a significant time-sink in manual feature engineering techniques to improve the data representation, whereas DL approaches obviate the need for manual feature engineering and allow task-specific data representations to be learned as part of the model training process. Next, we will investigate the trend of applying feature engineering in SE and discuss the relations among different data types (see Appendix A.3, available in the online supplemental material).

4.3.2 Model Training

The issues involved in training a ML model are different from those involved in training a DL model. Below we describe how the two differ from each other with respect to model construction and hyper-parameter optimization (i.e., choosing a set of optimal hyper-parameters for an algorithm). Generally speaking, a study should list all the attributes of a proposed approach described below, as failure to do so will result in a lack of reproducibility and replicability of the approach.

Model Construction. To construct a ML model, one needs to specify the learner to be used (e.g., DT, NB, or SVM). Specifying the learner is equivalent to specifying the algorithm to be used in the model construction process. For instance, a decision tree learner uses a particular splitting criterion (e.g., information gain, gain ratio) to learn a small tree (Occam's Razor — discussed in Section 4.4) that achieves a high accuracy on the training set. A SVM, on the other hand, uses the sequential minimal optimization algorithm to find the hyperplane with the largest margin. Once the learner is specified, one needs to specify a set of (typically) learner-specific hyper-parameters. For example, in random forest learning, one would specify how many trees are used.

Constructing a DL model is slightly more complicated. While there are standard, off-the-shelf neural network architectures that can be used, such as CNNs and RNNs (e.g., LSTMs, GRUs), if one desires to achieve good performance on a specific SE task, it is typically important to design task-specific architectures. For instance, one can (1) create multiple layers of LSTMs by stacking them, (2) employ bidirectional LSTMs to encode information from both sides of an input sequence, and/or (3) combine CNNs with RNNs. Since DL models typically assume embeddings as input, one has to decide what kind of embeddings to use and whether the embeddings can be updated in the model construction process. More recently, pre-trained models have been extensively used to construct DL models for SE tasks (see Section 4.2.4). Specifying the pre-trained model (if one is to be used) is part of the DL model construction process. As for hyper-parameters, there is typically a set of neural network-specific hyper-parameters that needs to be tuned, such as weight training algorithm (e.g., stochastic gradient descent [41], gradient descent [295], gradient ascent [296], conjugate gradient [297], quasi-newton method — Limited-memory BFGS [128], Levenberg-Marquardt [298]), the dropout rate, the number of epochs (training iterations), the learning rate, the number of hidden layers, the activation function, the dimensionality of a particular representation, the loss function, and the optimizer used for weight estimation (e.g., Adam [43], RMSprop [129], AdaGrad [22], Adadelta [299], AdaMax [295], the Momentum [300]).

We discovered two common ways that SE studies implemented their proposed ML and DL models: (1) using an off-the-shelf toolkit package, or (2) creating self-designed versions. In general, both ways are adopted in ML studies, while DL studies tend to build models from scratch or modify an existing model since off-the-shelf packages are more readily available for canonical ML algorithms. An off-the-shelf toolkit encapsulates all the implementation details to allow experiments to be configured on it and run on a user's machine [269]. For example, many ML studies [301], [302] used Waikato Environment for Knowledge Analysis (WEKA) [303], which is an open source collection of machine learning algorithms for data mining tasks, to create adopted classifiers with the default configuration. Compared to ML, DL usually adds two additional components to its model construction: word embedding and enhanced structures. As mentioned in Section 4.2.2, varieties of word embedding techniques are integrated into DL models and, typically applied to very large corpora. Embeddings can be used to reduce the space complexity of computation and measure the similarity of the words [304]. According to Table 7, standard DL architectures can be enhanced by other DL architectures (e.g, CNN+RNN), ML learners (e.g, DL+RL) and non-ML/DL techniques (e.g., Neural Machine Translation + Information Retrieval). To promote replicability and reproducibility, ML studies using an open source toolkit do not need to describe model elements in detail and can only specify their configuration of hyper-parameters. On the other hand, for the ML and DL studies that do not take advantage of an existing toolkit, missing any description of the aforementioned model elements in the original studies may make it difficult to replicate and reproduce the necessary details of proposed ML/DL methods.

Hyper-Parameter Optimization. As for hyper-parameter optimization, even though it is typically regarded as a "black art," its impact is well understood [305] and tuning needs to be repeated whenever the data or the goals are changed [306]. We discovered two different ways employed by ML and DL studies in SE for parameter tuning: (1) Use state-of-the-art hyper-parameter optimization techniques, and (2) create a self-designed algorithm or strategy to tune specific hyper-parameters. Both ways are adopted in ML studies, while DL studies tend to use a self-designed script. Specifically, we observed four automated parameter optimization techniques: (1) Grid Search (e.g., [307], [308]), which is the simplest optimization technique based on an exhaustive search through a set of parameters within a manually specified parameter space; (2) Random Search (e.g., [309]), which exhaustively searches through a set of parameters within a randomly generated parameter space; (3) Genetic Algorithm (e.g., [310]), which is an evolutionary-based optimization technique based on natural selection and genetics concepts, where the chromosomes of the two individuals (parents) work together to form a new individual by keeping the best properties from each of the parents; and (4) Differential Evolution (e.g., [311]), which is another evolutionary-based optimization technique based on the differential equation concept and uses mutation as a search mechanism [312]. Since DL typically has many hyper-parameters, the hyper-parameter space, which is the Cartesian product of the domains of all hyper-parameters, is huge. Ha *et al.* [313] proposed an efficient hyper-parameter optimization strategy for a deep FFN with three hyper-parameters that control the complexity of the network (i.e., number of layers, number of neurons/layer, regularization hyper-parameter) and two hyper-parameters that control the model training process (i.e., learning rate, number of epochs). It reduces the hyper-parameter search effort by (1) fixing some dependent hyper-parameters and (2) deriving a search strategy to effectively reduce the hyper-parameter space. We refer the reader to Appendix A.4 (available in the online supplemental material) for an investigation of whether there are hyper-parameter values and optimization techniques that are typically used in existing ML/DL models for SE tasks.

Overfitting. There is one critical problem for ML/DL models to combat during the model training process — overfitting, which occurs when a model that fits the training data too well but performs poorly on new, unseen data. Many methods are proposed to reduce the effect of overfitting and we listed some widely-used techniques in Table A.8 (see Appendix A), available in the online supplemental material. Based on our collection, we found that *cross-validation* is the most widely-used method in ML studies [123], [314] but it is not prevalent in DL studies. This is probably because much less data is needed by ML than DL and *k-fold cross-validation* is particularly useful when data is scarce (according to Table A.8). However, recent studies [104] in defect prediction revealed that *k-fold cross-validation* often introduces a nontrivial bias for evaluation, which makes the evaluation inaccurate, especially for change-level defect prediction. One typical reason is that randomly partitioning a dataset into k folds may cause a model to use future knowledge which should not be known at the time of prediction to predict changes in the past. On the other hand, due to the model depth and the capacity required to capture more complex representational spaces, DL models are often more susceptible to overfitting [315], particularly in networks with millions or billions of learnable parameters. For DL studies, we found that regularization strategies (e.g., *L1 regularization* [313], *L2 regularization* [313], *dropout* [90], [313], [316], [317], *early stopping* [90], *batch normalization* [153]) are often adopted to add some constraints to the objective function, allowing for good generalization to unseen data even when training on a small training set or with an inadequate number of iterations. Among them, we discovered that *dropout* is the most popular one since it not only prevents overfitting but also provides a way of approximately combining exponentially many different neural network architectures (e.g., Deep Siamese Network [90]) efficiently [318]. In addition, we found that CNNs [199] and the Siamese architecture [130] usually use a shared weight paradigm, and *Xavier initializer* [319] is a popular technique to initialize the non-embeddings weights [256], [320].

4.3.3 Evaluation

To comprehensively evaluate the proposed ML and DL models, performance evaluation and subsequent error analysis are two common activities.

Performance Evaluation. Prediction results are produced by applying the proposed ML/DL methodology to the preprocessed training and testing data, which would be the basis for the research results and outcomes [62]. To make the prediction results more credible, many studies chose to compare their proposed ML/DL models to other models or baseline approaches. There are many ways to determine the "quality" of an approach when compared to others. Providing the used evaluation metrics of this comparison is important for the replicability of the results.

Table 11 presents nine commonly used evaluation metrics against prediction performance and their definitions. Accuracy, precision, recall, F1, AUC and MAP are often used in classification tasks. However, precision and accuracy may not be robust to datasets where the target class is rare [328], which is common in defect prediction. On the other hand, F1 and Recall are relatively robust to data imbalance problems. AUC has been advocated to be a robust scalar summary of the performance of a binary scoring classifier [329]. However, the computational cost of AUC is high, especially for discriminating a volume of generated solutions of multi-class problems [330]. MAE has been recommended for the evaluation of software effort estimators because it is unbiased towards over or underestimations [331]. Choosing the right evaluation metric for a given task is essential, as failure to do so may provide an inaccurate characterization of how good a system is. How to choose the right evaluation metric for a given task depends on at least two factors. First, it depends on whether the task is a classification, regression, ranking, or generation task. As shown in the table, BLEU is appropriate for generation tasks but not classification tasks. Second, it depends on the class distribution. While accuracy makes the most sense to use when the class distribution in the test data is relatively balanced, the other metrics may be more suitable for datasets with skewed class distributions. As an extreme example, consider a 2-class classification task where one class comprises 99% of the instances. Merely classifying every instance in this test set as belonging to the majority class will enable the model to achieve a 99% accuracy. However, this is not reflective of the model's actual performance because, with such a skewed distribution, it would typically be necessary for the model to perform well on the minority class, which comprises only 1% of the test data. For this reason, metrics such as recall, precision, and F1, which can be computed for each class in the dataset regardless of whether the class is a majority class or a minority class, may provide a better characterization of model performance. For instance, to evaluate the performance of a neural network for API retrieval tasks, Nguyen *et al.* [332] used precision to show the number of correctly predicted relevant fragment-API pairs over all the retrieved pairs. They also used recall to get the number of the correctly predicted relevant

TABLE 11
Summary of Common Performance Evaluation Metrics for ML and DL Studies

Metrics	Definition	Suitable Tasks
Accuracy	The ratio of numbers of correct classification in a dataset.	Classification tasks.
Top-k accuracy	The ratio of the number of hits over the total number of cases.	Recommendation or prioritization tasks.
Precision	The ratio between the number of correctly predicted relevant data over all the retrieved data.	Classification tasks.
Recall	The ratio between the number of the correctly predicted relevant data over all the data.	Classification tasks.
F1	The harmonic mean of precision and recall, which gives a combined measure of accuracy	Classification tasks.
AUC (Area Under the Curve)	The relationship between true positive rate (TPR) and false positive rate (FPR).	Classification tasks.
MAP (Mean Average Precision)	The mean of average precision across all data.	Classification tasks.
BLEU	The similarity between two sentences in evaluation of machine translation systems.	Code or text generation tasks.
MAE (Mean Absolute Error)	A measure of errors between paired observations expressing the same pattern	Regression tasks.

fragment-API pairs over all the pairs. Top-k accuracy is used to reward a ML/DL model that makes at least one correct recommendation in the top k% ranked classes, and will allow us to see the trade-off performance when as k increases [333]. BLEU measures the quality of generated comments by calculating the similarity between the generated comments and references [334].

In addition, we observed some non-traditional and task-specific metrics being used, such as robustness and effectiveness. For instance, in test automation [335], robustness was defined as the percentage of passed test cases that were properly classified, measuring how well a ML model correctly identifies the negative cases; and effectiveness was defined as the percentage of failed test cases that were properly classified, measuring how well a model correctly identifies a condition. To evaluate the Discrete Adversarial Manipulation of Programs (DAMP) attack using three DL architectures [336], in targeted attacks robustness was defined as the percentage of examples in which the correctly predicted label was not changed to the adversary's desired label, and in non-targeted attacks it was defined as the percentage of examples in which the correctly predicted label was not changed to any label other than the correct label. Effectiveness was just the opposite: the lower the model robustness is, the higher the effectiveness of the attack is. We also found that explainability has been indirectly measured in technical debt detection [337], where the Jaccard coefficient was employed to measure the similarity among the set of key phrases extracted by a CNN. The higher the similarity is, the more intuitive and explainable the CNN-extracted key phrases are.

Error Analysis. Error analysis for ML/DL models examines the instances that the model misclassified so that one can understand the underlying causes of the errors, which is essential for reproducing the studies on a different problem. When the reproduced results are far from the results reported in the original study, previous error analysis could provide clues to locate the problem by steps such as: (1) Prioritize which problems deserve attention, (2) suggest the missing critical information in the methodology design, (3) provide a direction for handling the errors, and (4) check the validity of the assumptions. As an excellent example of error analysis, Liu *et al.* [338] performed a comprehensive error analysis on the misclassified pairs of changed source code and untangled change intents of AutoCILink-ML, a supporting tool for automatically identifying/recovering links between the untangled change intents in segmented commit messages and the changed source code files. They prioritized the problems of the misclassified pairs. One primary source of errors they found was misclassifying a "linked" code-intent pair (as "not linked"). Through their error analysis, this misclassification can be attributed to the inconsistent definitions/ambiguity of specific terms used in commit messages and their related documents and the same ones used in source code. To address this kind of error, they recommended word sense disambiguation.

4.3.4 Comparison of the Latest ML and DL Studies

In this subsection, we examine the latest ML and DL studies on the three SE tasks listed in Table 12 that are associated with the largest number of publications, namely *Defect Prediction (A1)*, *Defect Detection and Localization (A2)*, and *Software Quality Prediction (M10)*. Specifically, for each of these three SE tasks, we selected two recent ML studies and two recent DL studies. Table 12 presents the purpose of these ML/DL studies and shows for each study (1) the proposed ML/DL model(s), (2) the chosen dataset(s), (3) the data preprocessing techniques, (4) whether hyper-parameter optimization was conducted, and (5) the evaluation metrics. For each SE task, the two ML studies are listed above the two DL studies.

For each task in Table 12, we found that it is difficult to directly compare the model performance or even draw general conclusions among different studies. The reasons are two-fold. First, the purposes of different studies for the same task can be different. Second, there is a lack of a standard evaluation methodology/framework for a given SE

TABLE 12
Comparison of the Latest ML and DL Studies on Three Popular SE Tasks: Defect Prediction (A1), Defect Detection and Localization (A2), and Software Quality Prediction (M10)

Task	Purpose	Model	Dataset	Preprocessing	Tuning	Metric
A1	Modify and enhance three Cross-Project Just-In-Time Software Defect Prediction (JIT-SDP) approaches [321]	Bagging	9 open source projects (e.g., Tomcat, JGroups) & 3 proprietary projects	IDP	NO	Recall, G-mean
	Compare supervised and unsupervised methods for Effort-Aware Cross-Project Defect Prediction [273]	Supervised	AEEEM, NASA, PROMISE, RELINK	NO	NO	F1, AUC
	Propose a novel approach for defect prediction based on visualizing program files as images [103]	CNN	ant, camel, jEdit, log4j, lucene, xalan, xerces, ivy, synapse, poi	FS	NO	F1
	Propose a novel just-in-time defect prediction approach [322]	CNN	Bugzilla, Platform, Mozilla, JDT, Columba, PostgreSQL	DC, IDP, FC	NO	Accuracy, Precision, Recall, F1
A2	Propose an approach to automatically localize faults in software by modeling and predicting counterfactual outcomes [323]	RF	Defects4J	NO	NO	Exam Score
	Conduct a study on whether a fault can be detected by specific code coverage in automated test generation [324]	BN, SVM, RF	Apache Commons Codec, CLI, CSV, JXPath, Lang, Math, JFreeChart	IDP	NO	Precision, Recall, F1, AUC
	Propose an approach to predict locations of try blocks and automatically generate the complete catch blocks [118]	Seq2Seq	TBLD, CBGD	DC, FC	NO	Precision, Recall, F1
	Improve deep-learning-based fault localization with resampling [325]	CNN	chart, math, mockito, time, python, gzip, libtiff, space, nanoxml	IDP	NO	EXAM, Relative Improvement
M10	Predict vulnerable classes and methods in Java projects [326]	SVM, LoR	Apache Tomcat, Apache CXF, Stanford SecuriBench	DC, IDP, FC	NO	Recall, Precision, F1, AUC
	Propose and evaluate a general framework for vulnerability severity classification [327]	KNN, RF, DT	Windows 10, QuickTime, Oracle Business suite, etc.	IDP	NO	Precision, Recall, F1, Accuracy, AUC
	Conduct a comparative study on the performance of machine learning-based vulnerability detection [134]	LSTM, GRU, CNN	the National Vulnerability Database (NVD), Software Assurance Reference Dataset (SARD)	DC	NO	Precision, Recall, F1
	Assess the generalizability of code2vec token embeddings [138]	Seq2Seq	SWaT, WADI	DC	NO	Precision, Recall, F1, BLEU

task. More specifically, standard evaluation benchmarks are missing, and as a result, different researchers evaluated their methods on different datasets. Moreover, there is no standard evaluation metric(s), and as a result, different evaluation metrics were adopted. Worse still, none of these studies reported the hyper-parameters tested, which makes it difficult to reproduce the results. Data cleaning and imbalanced data preprocessing techniques are widely adopted in these studies.

What are the current trends of the competing ML and DL techniques applied to these three SE tasks? For defect prediction, the two ML studies focused on investigating or improving the effectiveness of state-of-the-art ML learners on two popular types of defects: just-in-time defects and effort-aware defects. For instance, Just-In-Time Software Defect Prediction (JIT-SDP) is concerned with predicting whether software changes are defect-inducing or clean [321]. To address this task, Tabassum *et al.* [321] proposed an ensemble model (Oversampling Online Bagging), which tackles class imbalance evolution in an online JIT-SDP scenario taking verification latency into account. We found that one DL study began to use a different data type (image) for defect prediction and exploited the advantage of a CNN in image classification. More specifically, Chen *et al.* [103] proposed an end-to-end DL framework that can directly get prediction results for programs without utilizing feature extraction tools. They first visualized programs as images, then applied a self-attention mechanism to extract image features and used transfer learning to reduce the difference in sample distributions between projects, and finally, fed the image files into a pre-trained, deep learning model for defect prediction.

For defect detection and localization, the two ML studies highlighted the advantages of two different categories of fault localization techniques: value-based techniques [323] and code-based techniques [324]. Specifically, code-based techniques measure the statistical associations/correlations between the occurrence of observable software failures and the coverage of individual program elements that are potential fault locations, such as statements, basic blocks, and subprograms. By contrast, value-based techniques focus on the values of a program's variables, which carry relevant information that is often neglected when conditional branches are omitted or incorrect. On the other hand, the DL models can help localize the fault in the program and generate patches to fix defects. Specifically, Zhang *et al.* [118] utilized a DL generator (SEQ2SEQ) to learn patterns from a large amount of historical exception handling code, which can localize potential exceptions in the source code and generate code to handle the exceptions.

For software quality prediction, vulnerability prediction appears to be an active research topic in software security since both ML studies and one of the DL studies focused on it. Sultana *et al.* [327] concluded the positive effect of the identified class-level metrics, and Chen *et al.* [326] confirmed the effectiveness of feature selection on vulnerability prediction. Zheng *et al.* [134] concluded that DL models can achieve better performance in vulnerability prediction than canonical ML models. The other DL study [138] investigated a different quality attribute, the generalizability of the token embeddings learned by code2vec for downstream SE tasks.

4.3.5 Replicability and Reproducibility Assessment

According to the ACM policy on artifact review and badging [339], replicability refers to the ability of an independent group to obtain the same result using the author's own artifacts. Likewise, reproducibility is the act of obtaining the same result using generally the same methods rather than the original artifacts. Reproducibility is clearly the ultimate goal, but replicability is an intermediate step to promote practices.

Why are replicability and reproducibility of studies essential to SE? Replicability and reproducibility are essential to identifying the quality and testing the credibility of the original studies, which in turn can increase the confidence that we can have in the results and allow us to distinguish reliable and unreliable results [269]. More importantly, the replicability and reproducibility of ML/DL applications have a great impact on generalizing and applying research results to different domains. As mentioned in Section 1, ML/DL has become a popular way to represent data in SE due to the great performance in classification, regression and generation tasks. A lack of reproducibility and replicability can be detrimental to the SE research community.

How many studies addressing ML/DL applications were replicated and reproduced in SE? Following the extraction process stated in Section 3.4, we found that 236 of the 1,428 papers (17%) claimed that they either replicated the same methodologies of other studies in our collection as baselines and used the same dataset to test their performance [189] or reproduced the methodologies on a different dataset to test the generalizability of their findings [340]. Tracing back to the studies being replicated or reproduced, we found that 189 studies were being replicated and 41 studies were being reproduced (24 studies were being replicated and reproduced) in these 236 papers. Following the *narrative synthesis* stated in Section 3.6 and based on the quality checklist in Table 4, we found two types of relations between replicability/reproducibility and the level of detail provided by ML/DL-related SE studies: direct and indirect.

What and how much information should a ML/DL study provide to simplify replicability and reproducibility in SE? According to Fu and Menzies [314], it is hard to replicate or reproduce ML/DL applications in SE research due to the nondisclosure of datasets and source code. Therefore, for the direct relations, we checked whether a study provided the full replication package, including training data and source code,[8] which is the easiest way for other researchers to replicate the original experiments. Training data comprises the data consumed by ML/DL algorithms after preprocessing a raw dataset. If the data is not accessible, there may be a mismatch between the regenerated data and the originally preprocessed data [103]. On the other hand, the full package of source code consists of at least two parts, end-to-end scripting (e.g., data preprocessing, statistical analyses) and the implementation of the ML/DL-based approach (e.g., model construction, parameter tuning methods, training algorithms) [341], which guarantee the compliance with the same ML/DL workflow when replicating. If the source code is not accessible, it would leave follow-up studies no choice but to re-implement the entire approach from scratch by relying on the description of the approach in the original study, which may not contain all original implementation details [103]. A further analysis is described in Appendix A.5, available in the online supplemental material.

Recommendation. According to the results from Tables A.2, A.3 and A.5 (see Appendix A), available in the online supplemental material, "Rep" studies have a better rate than "Non-Rep" studies for providing almost all the details in the checklist (except QA8), which indicates that this information indeed benefits the replicability and reproducibility of ML/DL studies in SE. On the other hand, though we separately assessed the studies with and without available replication packages, providing both the replication packages and detailed descriptions is recommended. While replication packages could make the replication and reproduction of ML/DL studies much easier, detailed descriptions of the ML/DL workflow activities could promote the understanding of the proposed ML/DL methodologies by other researchers, which would be helpful for reproducing the ML/DL studies in more generalized and applicable scenarios. Therefore, based on the patterns found in the analysis of direct and indirect relations (see Appendix A.5), available in the online supplemental material, we summarize some actionable implications against the threats to replicability and reproducibility, as described below:

- *Sharing the replication package and providing usage instructions.* It is good for long-term maintenance to share the implementation and training data in an open-access platform (e.g., GitHub, Zenodo), which

[8] According to the ACM policy, an artifact is available if a DOI or a link to the data or source code repository, along with a unique identifier for the object, is provided.

can reduce the risk of deprecated links. We found that such cases usually occurred when a dataset was posted on the server of a company or an educational constitution. To address the privacy concerns of proprietary or industry data, some state-of-the-art algorithms are available to prevent the disclosure of sensitive metric values [342], such as CLIFF&-MORPH and ManualDown. Besides, it would be better to provide some instructions (e.g., readme) on (1) how to deploy the proposed ML/DL model on different operating systems, and (2) which files should be used for training or testing.

- *Sharing the tools or methods for data extraction and preprocessing*. For state-of-the-art tools, it is necessary to provide the access link or the referenced studies. Open-source and free-access tools are highly recommended, such as BeautifulSoup (a Python parser for HTML and XML files) [343], Scrapy (a crawler to obtain web pages) [344], SZZ (an approach to identify bug-introducing commits) [345], Stanford CoreNLP (a set of NLP tools) [346], and Porter Stemming (an algorithm for term normalization) [347]. For self-designed methods, researchers are encouraged to provide the pseudo code with detailed explanations in the paper and include the implementation in the replication package. In addition, for data cleaning, noise filtering criteria and related statistics are needed in order for other researchers to verify and replicate. These include statistics on how the data size changes by following each filtering criterion.
- *Fully specifying hyper-parameter optimization for ML/DL*. ML/DL papers should specify (1) which hyper-parameters are to be tuned (and if so, using which method) and which ones are set to their default values, and (2) training details, such as the loss function and the weight update algorithm.
- *Performing an error analysis for ML/DL*. We recommend that researchers locate and understand the underlying causes of the errors, typically following the four steps described in Section 4.3.3.

4.4 Understanding ML/DL Technique Selection (RQ3)

It is important to ensure that others understand why a ML/DL technique is selected for a specific SE problem. Lacking rationales and tradeoff analysis among SE studies would adversely affect the generalizability and applicability of the ML/DL models. Many ML/DL techniques are chosen based on heuristics or experiences. For a given problem, a model can be chosen because (1) it performed better in the past (discussed in *Theme 1*), (2) the comparative results are better, or (3) the trials show success. For the second scenario, according to QA12 in Table A.5, 1,157 (975 + 182) studies compared their proposed approaches with other baseline approaches. We discovered that the majority of these baseline methods are also ML/DL-based (different ML classifiers or DL architectures), though some other non-ML/DL baseline methods/tools were also adopted. For instance, Wang *et al.* [348] compared six different ML classifiers (i.e., RF, Decision Table, DT, NB, LoR, and SVM) to assess the automated patch correctness and results showed that RF was the most effective model since it achieved the highest recall while still maintaining a relatively high precision for two experimental settings. For the third scenario, Ye *et al.* [72] trained task-specific word embeddings to estimate semantic similarities between documents and empirical evaluations showed that the embeddings led to improvements in a previously explored bug localization task and the newly defined task of linking API documents to computer programming questions. To further address **RQ3**, we first used the *thematic synthesis* method by *vivo coding* (described in Section 3.6) to collect summarized rationales (vivo codes) from all 1,428 papers with regard to the selection of ML/DL models. Then based on these vivo codes, we manually identified five patterns (themes) and grouped the vivo codes into similar categories. The result analysis of these five patterns is shown below.

We found that 516 of the 1,428 papers (36%) provided explicit rationales for their proposed ML/DL models, including 346 ML studies and 170 DL studies, respectively. We then grouped the collected vivo codes into five themes, as shown in Table 13. We observed that the rationales for the chosen ML models spanned over all five categories, while almost all rationales for the chosen DL models fell into two themes: "Better Performance" and "Simple Task/Data." As mentioned earlier, due to the complex internal representation of DL, it is understandable why "Better Interpretability" and "Simple Implementation/Model" were not the two reasons for SE researchers to select DL models. However, it was somewhat surprising that only one paper explicitly mentioned that "Robustness to Data Problems" (CNN-based image classification techniques can effectively remove the non-code and noisy-code frames [200]) was one of the reasons for selecting DL models. Next, we will discuss essential findings upon these five themes.

Better Performance. The rationale "Better Performance" is described in a similar fashion for selecting both ML and DL models — that is, "previous studies have demonstrated the great performance of the chosen ML/DL model in similar SE tasks." As shown in Table 13, we noticed that 319 (236+83) of the 516 studies (62%) reporting rationales considered performance as the top criterion for selecting ML/DL algorithms. This result shows that *Theme 1* is dominant in selecting the appropriate ML/DL models for SE tasks. However, using "Better Performance" as the top prioritized rationale of selecting the appropriate ML/DL models for specific SE tasks may potentially have a negative impact: Researchers may blindly trust the reported good performance of specific ML/DL techniques while ignoring their downsides, such as the long computation time and the higher modeling complexity, thus missing a trade-off analysis among techniques [158], [314], [349], [350]. The above-mentioned two downsides are correlated to *Theme 4* and *Theme 5*.

Robustness to Data Problems. Due to the unique characteristics and capability, different ML and DL models can be resilient to diverse data problems, which may still achieve good performance in imperfect datasets without any preprocessing work. For ML, we identified five different kinds of data problems which the selected ML models from 36 studies were resilient to, as shown in Table 14. First, *Noisy and missing data* is the most common problem when

TABLE 13
Rationale Categorization for ML and DL Studies

ID	Theme	Comments	Example	# ML	# DL
1	Better Performance	Proven in prior studies compared with other ML/DL models.	*"We selected random forest since this algorithm has been used in many previous empirical studies and tends to have good predictive power."*	236	83
2	Robustness to Data Problems	Apply robust learners which are resilient to diverse data problems (e.g., noise, missing data).	*"We use Random Forest to take into account the specific characteristics of merge data, such as being imbalanced, in our classifiers."*	36	1
3	Simple Task/ Data	The complexity of the underlying task/data.	*"The advantages of using a neural network is to automatically extract useful features of link artifacts. We can train the network to encode the artifacts into real-valued vectors that capture relevant features. This relieves us from the arduous and brittle task of manual feature extraction, which requires (i) expert domain knowledge and (ii) maintenance in case of changes in Android icc or the used static analysis"*	77	108
4	Better Interpretability	Explainable model.	*"We chose to use a decision tree classifier for this study since it offers an explainable model. This is very advantageous because we can use these models to understand what attributes affect whether or not a bug will be re-opened."*	13	0
5	Simple Implementation/ Model	Easy to implement or simpler structure of the model.	*"Naive Bayes was the best choice because of the ease in which its training can be updated on-the-fly, improving its performance as it adjusts to its user."*	26	1

"Theme" shows the summarized themes and "Comments" lists the guidelines to group the similar vivo codes. Sample rationales are shown in "Example," and "# ML" and "# DL" show the total number (includes the overlap that a rationale could be grouped into multiple themes) of relevant rationales per theme for ML and DL, respectively.

obtaining data directly from raw datasets in software repositories since these datasets are rarely clean and complete [274]. In addition to the aforementioned data preprocessing techniques in Section 4.3, another solution to mitigate the noisy and missing data issue is to apply robust learners, which are characterized as being less influenced by imperfect data, such as decision tree learners. DTs are widely applied in SE studies because of their robustness against noise. They have the ability to identify irrelevant attributes, as well as detect discriminating, missing or empty attributes [363]. Second, *class imbalance* refers to the samples of one class significantly outnumber the samples of others, which appears frequently in classification tasks, such as *Defect Prediction*. Third, *overfitting* is a common problem during the model training process, which was discussed in Section 4.3.2. Ensemble learning algorithms are often selected by researchers due to their robustness to class imbalance and overfitting problems. Robustness is guaranteed by averaging the classification performance of multiple classifiers, which leads to the elimination of uncorrelated errors and, thus, enhances overall classification performance[353]. Fourth, *small datasets* can always be well addressed by SVMs [364] because SVMs are based on a solid theoretical foundation and can adapt to datasets with relatively small-scale samples. Finally, for *unlabeled data*, unsupervised or semi-supervised algorithms are selected because they either do not need the prior labeled data (unsupervised) or require only a small set of labeled data (semi-supervised). As shown in Table 13, only one DL study explicitly addressed the robustness to data problems. This implies that researchers might need to pay more attention to the robustness of different DL model architectures in SE.

Simple Task/Data. As mentioned earlier, there are indeed circumstances in which one may prefer DL to ML (and vice versa). The oft-cited reasons for using ML include task/data simplicity. However, when the underlying data points can be fit, for instance, by a linear function, one may prefer to use a SVM with a linear kernel to prevent overfitting the training data (e.g., [360]). In contrast, when the underlying data exhibit complex patterns and/or require a deeper, possibly semantic understanding, then DL is certainly the preferred choice (e.g., [260]). In addition, if the task is too complex to enable the design of hand-crafted features, then the ability of DL to learn task-specific feature representations would make DL the ideal choice (e.g, [214]).

Better Interpretability. Interpretability (also called explanability) can be achieved at two levels: (1) Global — using interpretable ML techniques or intrinsic model-specific techniques (e.g., *SkopeRules* and *RuleMatrix* [365], *Duplex Output algorithm* [289]) so that the entire predictions and recommendations processes are transparent and comprehensible; and (2) local — using model-agnostic techniques (e.g., LIME [366]) to make the prediction results more interpretable [367]. For global interpretability, as shown in Table 13, all 13 ML studies selected DTs (or tree-based algorithms, e.g., Classification and Regression Tree) considering the interpretability of the model. DT boundaries are parallel to the dimensions of the input space and expressible in terms of linear conditions over input variables, which makes DT boundaries understandable by researchers [368]. None of 170 DL studies reported rationales related to global interpretability because DL is inherently harder to interpret given their internal mechanisms are a "black box" to SE researchers. Note that the states of a RNN are in the form of numerical vectors which

TABLE 14
Summary of Data Problems Extracted from ML Studies

Data Problem	Tasks	Selected ML Models	Example References
Noisy or missing data	R2,R6,I9,A1,M7,P1,M12,P2,T4,M10	DT, RF, SVM, NN	[282], [351], [352]
Class imbalance	A1,M21,M22	Ensemble (e.g., RF, Bagging), NN	[353], [354], [355]
Small dataset	I3,A2,P1,M2,P7,I6	Regression Tree, SVM, Reinforcement	[84], [356], [357]
Overfitting to training set	M21,M10,M3,A1,I11	SVM, Ensemble (e.g, RF, AdaBoost, XGBoost)	[358], [359], [360]
Unlabeled data	P1,M5,P4	Semi-supervised, Unsupervised	[282], [361], [362]

The Task-IDs *are in Table 5, and "Selected ML Models" denotes some typical ML models chosen by ML studies to address each data problem.*

have multiple values and are hard to interpret [369]. Furthermore, the intrinsic logic of the convolutional layers of a CNN is less interpretable, which prevents understanding the attention of a CNN at different levels and scales [370]. The difficulty in the interpretability of neural network representations, such as CNNs and RNNs, has always been a limiting factor for the applicability and generalizability of DL applications in SE. Although no DL studies provided rationales related to global interpretability, with the increasing attention paid to the interpretability of both ML and DL models, balancing the trade-off between performance and interpretability is now becoming a hot topic in SE [288], [371]. This is mainly being addressed in *SE for ML/DL*, which is beyond the scope of this study. Rather than understanding and reasoning about neural networks directly through software testing and verification techniques (e.g., mutation testing [372], concolic testing [373]), recent research aims to extract simpler models from neural networks such that they are human-interpretable [369].

For local interpretability, model-agnostic techniques can provide an explanation for each prediction (i.e., an instance to be explained) [367], which can let users understand why the prediction is made by ML/DL models. Such techniques were first introduced and empirically evaluated in SE in 2020 by Jiarpakdee *et al.* [374], which focused on generating instance explanations for defect prediction models. We encourage SE researchers to improve state-of-the-art ML and DL models to not only provide insights or generate recommendations but also be able to explain how these insights and recommendations are generated by the ML/DL models, in order to help other researchers or practitioners understand how the models arrive at a decision and why they outperform or underperform in specific scenarios.

Simple Implementation/Model. Based on the 26 ML studies and one DL study in Table 13, we observed two patterns of describing rationales related to simple implementation. First, one common kind of rationales observed in all 27 ML/DL studies is "available off-the-shelf implementations" [375] or "available replication packages from other studies" [376]. Second, for the studies that built a self-designed version, the simplicity of the implementation for ML and DL models depends on various factors. From the point of view of feature engineering, as mentioned in Sections 4.2.5 and 4.3.1, a DL model might be easy to implement because it obviates the need for feature engineering, while all canonical ML models can only utilize manual feature engineering, which is labor-intensive and may need to be performed for each new task or dataset [377]. However, DL models typically require larger amounts of time and computational resources to train [314] than many ML models (e.g., LiR, LoR).

The other kind of rationales (observed in the 26 ML studies) is that a chosen ML model has a much simpler structure than a DL model, making the learning process understandable and easily explainable. Simply put, the complexity of a learning algorithm will affect the generalizability of the learning model. According to Occam's Razor [378], machine learning models with less complexity are preferred as they are expected to generalize better. In other words, if two models have the same performance on a training dataset, then the simpler model should be chosen because it is expected to generalize better when used to make predictions on new data [379]. Among the 27 ML/DL studies in our collection, four studies did not provide any comparative analysis, and the remaining 23 studies provided the comparison analysis and experiments among ML or non-ML models in terms of complexity. Given that DL is generally more complex than canonical ML classifiers, "ML versus DL" is the most apparent tradeoff analysis. However, we observed no such tradeoff analysis in these 23 studies, which indicates little consideration of Occam's Razor [378] in SE — a complexity comparison as well as tradeoff analysis on various ML/DL models for a specific SE task. Recently, controversial studies have emerged that warn us to use ML/DL cautiously and encourage the exploration of simple methods as part of the rationale of model selection. For instance, Liu *et al.* [156] proposed a nearest neighbor algorithm that did not require any training to generate short commit messages, which not only was much faster and simpler but also performed better than the Neural Machine Translation approach by Jiang *et al.* [158]. Xu *et al.* [350], [380] and Fu and Menzies [314], [349] debated the effectiveness of DL versus SVM in predicting the relatedness between Stack Overflow knowledge units. Eventually, they agreed in part that while SVM based approaches offered slightly better performance, they were slower than DL-based approaches on larger datasets. These studies shed light and provide excellent examples on how the complexity tradeoff analysis can be performed to guide the selection of appropriate ML/DL models for SE tasks.

For these five themes, we have provided answers to the questions of "when to use ML/DL," and "which ML/DL to use." A natural question is "when *not* to use ML/DL." Recall that ML/DL allows us to automatically acquire knowledge from data. Hence, one can easily conceive the scenarios in which one should not apply ML/DL to a given SE task.

First, the amount of data available for training — particularly annotated data — is inadequate for training an accurate model. As mentioned before, while annotated data can be obtained easily for some SE tasks, the same is not true for other SE tasks. Consider, for instance, process management tasks, where the associated software project data (e.g., the attributes characterizing software projects and the successful software process models [381]) is relatively small and has to be collected over a long period of time. This implies that it could take a long time to collect a reasonable amount of data for these tasks. The difficulties involved in data collection could be a reason for SE researchers to consider employing non-ML/DL approaches.

Second, the knowledge needed to properly address a given SE task is absent from data. This typically occurs when expert knowledge is required to address the task. Root cause analysis is a good example of a task that needs to be addressed with expert knowledge: It is usually hard to extract meaningful features from ground-truth root-causes (e.g., log data), and it requires a huge amount of manual effort to transfer adequate domain knowledge to a feature matrix. For instance, in order to categorize the root causes of failed tests from log data through ML clustering and classification algorithms, Kahles *et al.* [382] had to conduct further interviews with the testing engineers and map their expert knowledge into distinctive ground-truth root-cause categories.

5 Discussion

5.1 Summary of Findings

In this section, we will summarize the findings from the three research questions.

RQ1. As far as the application of ML to SE tasks is concerned, we have observed two important trends over the last six years: (1) a larger variety of SE artifacts has been effectively analyzed using different ML techniques; and (2) word embedding techniques are being integrated into different ML-based applications. As for the application of DL to SE tasks, we have observed that (1) classification results could be improved by replacing or combining the hand-crafted features that are typically used in ML with representation learning (by DL); (2) results are continuously improved by enhancing DL models; (3) the generalizability to different presentation styles (unseen projects) could be improved by pre-trained models such as BERT; (4) through the application of CNNs, SE researchers have made significant progress on identifying and extracting elements embedded in multimedia; and (5) with the help of Seq2Seq deep generation models, code and text based generation tasks in SE have been tackled more effectively than before. In addition, we have found that (1) CNN or RNN based architectures are widely adopted in DL models because they are adept at handling images and text, which are the two major types of data in SE research; (2) two types of CNNs and RNNs, namely Siamese and Tree-based models, have been specifically tailored to fit SE tasks. Finally, we have identified novel ML/DL applications in SE, such as screencast analysis and the use of biometrics, as well as novel ML/DL models that were specifically developed for the SE domain, such as pre-trained models of source code.

RQ2. For data preparation, ML and DL share commonalities in data source and data extraction. As far as data preprocessing is concerned, while we discussed the similarities in preprocessing text, code and image for ML and DL, we identified that imbalanced data preprocessing techniques are more frequently mentioned in DL studies than ML studies in requirements engineering. In addition, there is a crucial difference between the two with regard to feature engineering: Canonical ML approaches have a significant time-sink in manual feature engineering techniques to improve the data representation, whereas DL approaches obviate the need for manual feature engineering and allow task-specific data representations to be learned as part of the model training process. Besides, we observed that there were an increasing number of studies that employ automated feature engineering in the last two years, and that automated feature engineering has been a preferred choice recently when using *code*. As for model training, we found two common ways that SE studies implemented their proposed ML and DL models: (1) using an off-the-shelf toolkit package, or (2) creating self-designed versions. However, we discovered that DL studies typically involve building models from scratch or modifying existing models while off-the-shelf packages are more readily available for canonical ML algorithms. As far as hyper-parameter optimization is concerned, we discovered two common ways employed by ML and DL studies in SE for parameter tuning: (1) use state-of-the-art hyper-parameter optimization techniques, and (2) create self-designed algorithms or strategies to tune specific hyper-parameters. Both ways are adopted in ML studies, while DL studies tend to use self-designed scripts. Though DL models typically have more hyper-parameters than ML models, we did not observe the correlations among the SE task, the type of hyper-parameter optimization and the type of DL architecture. With respect to evaluation, for both ML and DL, we discussed nine commonly used evaluation metrics against prediction performance and three non-traditional, task-specific evaluation metrics, namely robustness, effectiveness and explainability. We noticed that there was a lack of a standard evaluation methodology/framework for ML/DL studies when applied to each SE task, and that it was still not a common practice for SE researchers to share their data and ML/DL implementations.

Finally, we discovered that it was not uncommon to encounter replicability and reproducibility issues in SE because of the prevalence of inadequate descriptions. Specifically, we found that ML/DL studies in SE need to provide detailed descriptions of (1) the tools or methods for data extraction and preprocessing in data preparation, (2) the hyper-parameter optimization procedure, and (3) the error analysis in evaluation.

RQ3. We found that heuristics and past experiences, especially "Better Performance", have been adopted as the top rationale of selecting the appropriate ML/DL models for a specific SE task by SE researchers, and that little consideration has been given to model complexity (Occam's Razor) in this decision process. In addition, we identified five data problems to which the selected ML models were resilient. We also acknowledged the improvements in interpreting neural network representations and the attempts made by complex and black-box models to explain their decisions.

5.2 Actionable Implications

Given the above findings, next we will propose some actionable items for researchers who conduct research related to the synergy between SE and AI, ML/DL tool builders for SE research/practices, and educators.

To promote the generalizability and applicability of ML/DL approaches to SE tasks, researchers should take into consideration more factors rather than blindly trust heuristics and prior experiences during *the model selection process.* Specifically, based on the guidelines (Section 4.2.5) summarized in **RQ1** and the findings from **RQ3**, it is necessary for researchers to perform a comprehensive trade-off analysis among the following factors: Occam's Razor (model complexity), task complexity, task types (e.g., classification, generation), interpretability, and dataset quality.

To ensure the replicability and reproducibility of ML/DL studies in SE, researchers should share a replication package and provide sufficient details when describing the processes of data preparation, model training, and evaluation. In addition, we recommend that the organizers of SE conferences consider adopting a practice that is now fairly standard in AI conferences, which is to ask the authors to fill out a reproducibility *checklist* when submitting a paper. The checklist is typically composed of a set of questions which ensures that the authors have provided sufficient information for replicability/reproducibility, such as "Did you report the number of parameters in the models used, the total computational budget (e.g., GPU hours), and computing infrastructure used?" and "Did you discuss the experimental setup, including hyperparameter search and best-found hyperparameter values?"[9] Reviewers are then explicitly asked to take into account the information provided in the checklist when reviewing a submission.

To narrow the gap between academics and real-world practices, tool builders should make the ML/DL techniques more explainable and actionable. According to a qualitative survey of practitioners' needs in defect prediction models [383], the explainability and actionability of software analytics are as important as the predictions. Most software practitioners do not understand the reason behind the predictions from software analytics (explainability) and do not know what to actually do or what to avoid doing to improve the quality of the software system (actionability) [384]. This leads to a lack of trust and transparency, hindering the adoption of ML/DL techniques in practice. According to **RQ3**, there is still much work required to improve both the global and local levels of interpretability (explainability) for ML/DL techniques in SE. To make ML/DL more explainable, three steps are recommended to tool builders: (1) analyze the domain for better understanding the SE task, social contexts, and stakeholders, especially figuring out the "explain to whom?" (e.g., developers); (2) elicit the requirements to understand practitioners' needs, including the goals (e.g., gain deeper insights) and "what to explain" (e.g., why a file is predicted as defective); and (3) design the solution for explanation, figuring out the scopes (local or global), the types of ML/DL models (described in Section 2) and the (model-specific or model-agnostic) techniques. A lack of actionable guidance for ML/DL remains an extremely challenging problem in SE. Tantithamthavorn *et al.* [383] generated two types of actionable guidance for defect prediction models by using a rule-based model-agnostic technique: (1) What developers should do to mitigate the risk of having defects and (2) what developers should not do to avoid increasing the risk of having defects.

To address SE concerns in AI courses, relevant course materials or books (e.g., ML/DL for SE and SE for ML/DL) should be provided to educators who teach the applications of AI in SE. AI education typically focuses on algorithms and techniques or applying these techniques in artificial settings (e.g., fixed datasets and Jupyter notebooks), and are narrowly focused on optimizing model accuracy [385]. However, there are some discrepancies between the SE process and the ML workflow, so knowledge of how to integrate the ML workflow into the SE process should be discussed in a book or taught in a specific curriculum. For instance, according to **RQ2**, our review identified a lack of a standard evaluation methodology/framework for ML/DL applications in each SE task. Hence, more formal and practical tutorials of evaluating ML/DL applications in SE could be provided, which can standardize the evaluation process and let researchers easily compare model performance among different studies for the same SE task. More generally, the field of ML/DL for SE has been growing so rapidly in recent years that it has been difficult for SE students and practitioners to keep abreast of the research progress. Someone who wants to understand this area of research may not even know where to begin, and this SLR could provide a useful starting point. To better organize the research results in this field, the SE community may consider building a website that enumerates each SE task. Specifically, for each SE task, there can be a dedicated webpage that enumerates the relevant resources, including the links to papers published on that task, the publicly available annotated datasets, the publicly available implementations of systems developed for that task, as well as a leaderboard that tracks the best results achieved to date on each dataset. We note that this requires community-wide effort, but having a website like this could have a lasting impact on the SE community, as it would make it easier for SE researchers — particularly those who are new to the field — to access the resources relevant to each SE task. Lastly, our SLR represents an important first attempt on analyzing a sub-discipline (*ML/DL for SE*) in SE. We expect the insights to be updated continuously by incorporating emerging ML/DL studies in the future.

5.3 Future Work

SE for ML/DL. In the last two years, the widespread adoption of deep neural networks in software systems has fueled the need for software engineering practices specific to DL systems [386] and the number of studies to investigate *SE for ML/DL* is rapidly increasing — typically for testing and debugging ML/DL systems [387], [388], [389]. Compared to traditional software systems, ML/DL systems are relatively less deterministic and more statistics-orientated [389] due to their fundamentally different nature and construction. In the future work, we would conduct a deep analysis of *SE for ML/DL*, where SE techniques can be used to help guide the creation of useful ML/DL software. We are also interested in investigating the replicability and reproducibility of those papers related to *SE for ML/DL*. Lastly, we plan to explore the recent studies that have tried to tackle the interpretability [369], [371], fairness [390], and robustness [391] of DL, and anticipate that future research could bring additional insights to how improved DL model performance can be turned into more effective and efficient SE practices, and what changes in SE practices would be useful to optimize the proposed DL-based approach.

Transferring ML/DL Research Results to Real-World Applications. Technology transfer demands significant efforts to adapt a research prototype to a product-quality tool that addresses the needs of real scenarios and which is to be integrated into a

[9] See https://aclrollingreview.org/responsibleNLPresearch/ for a sample checklist.

mainstream product or development process [392]. It also requires close cooperation and collaboration between industry and academia throughout the entire research process. Later on, we would like to collaborate with industry partners to create a road-map to improve the state-of-the-art ML/DL-related SE research and facilitate the transfer of research results into real-world applications. The potential road-map also aims to stimulate the future directions for research on the synergy between SE and ML/DL, and to build a healthy eco environment for collaboration among SE/AI researchers and industrial practitioners through observing the limitations from this road-map.

6 Limitations

One limitation of our study is the potential bias in determining the categories of SE tasks for the collected studies. To better explore the insights from the classification, the categories' granularity should be neither too coarse nor too fine. For those questionable studies, we first identified the keywords, which indicate related tasks, in the abstract or related work according to the authors' claim to determine their temporary categories. Then, we compared the objectives and contributions with other studies. We decided whether these temporary categories should be merged with the existing SE tasks or kept as new ones. The advice from the SE experts also mitigated this threat to study validity.

Another limitation might be the possibility of missing ML/DL-related studies during the search and selection phases. Although it was impossible to identify and retrieve all relevant publications considering the many ML/DL-related SE studies, our search strategy integrated manual and automated searches. This could keep the number of missing studies as small as possible.

7 Related Work

Some empirical and case studies of SE and ML emerged at the beginning of the 21st century. Di Stefano and Menzies [393] suggested academic application guidelines and conducted a case study on a reuse dataset using three different machine learners. Zhang *et al.* [2] grouped around 60 existing ML studies in SE at that time into a limited number of SE tasks in which the majority of them are software maintenance and project management tasks. Then they discussed some general steps regarding applying machine learning methods to SE tasks and provided a guideline on how to select the type of learning methods for a given task. One common purpose of these studies was to stimulate more research interest (which was scarce at that time) in the areas of ML and SE. In the end, this new research materialized, and was verified by our SLR: The number of canonical ML and emerging ML (e.g., DL) studies in SE is thriving in the past decade (2009-2020), with thousands of papers being published on nearly 80 SE tasks, and the process of applying ML techniques to SE tasks has been regulated to three specific stages: data preparation, model training and evaluation.

Mahmood *et al.* [269] found low replicability[10] in defect prediction and potential low quality studies in defect prediction. They investigated what characteristics of a defect prediction study make it likely to be replicated and provided practical steps to incentivize and standardize aspects of replication. In contrast, we aim to extend this study to the entire SE domain, investigating not only what but also how the identified factors could make the original studies more replicable and reproducible.

Many SLRs, surveys and comparative studies [183], [394], [395], [396], [397], [398], [399], [400] have focused on investigating the use of ML/DL in software defect prediction. Wen *et al.*'s SLR [401] explored ML-based software development effort estimation models from four aspects: type of ML technique, estimation accuracy, model comparison, and estimation context. Fontana *et al.* [402] performed the large-scale experiment of applying 16 ML algorithms to code smells and concluded that the application of ML to detect code smells can provide a high accuracy. Compared to our SLR, the findings from the above studies are limited to several specific SE tasks, and performance comparisons (rather than a comprehensive evaluation) are often the main purpose.

8 Conclusion

This paper presented a systematic literature review that comprehensively investigated and evaluated 1,428 state-of-the-art ML/DL-related SE studies to address three research questions that are of interest to the SE community. Through an elaborated investigation and analysis, we provided a comprehensive review of the current progress and generalizability of ML/DL for SE, which can be summarized as follows. First, we observed two changes that ML brings to SE after 2014. Second, we identified three improvements by DL classification models over ML classification models and two unique DL contributions to SE tasks that ML techniques were not capable of tackling. Third, though having some commonalities in data preparation, model training and evaluation, ML and DL have different feature representations, different hyper-parameters to be tuned, and different ways to be implemented when applied to SE tasks. Finally, we discovered five main reasons why SE researchers select a ML/DL technique for a specific SE task.

References

[1] A. E. Hassan and T. Xie, "Software intelligence: The future of mining software engineering data," in *Proc. FSE/SDP Workshop Future Softw. Eng. Res.*, 2010, pp. 161–166.

[2] D. Zhang and J. J. Tsai, "Machine learning and software engineering," *Softw. Qual. J.*, vol. 11, no. 2, pp. 87–119, Jun. 2003.

[3] A. E. Hassan and T. Xie, "Mining software engineering data," in *Proc. IEEE/ACM 32nd Int. Conf. Softw. Eng.*, 2010, pp. 503–504.

[4] Q. Niyaz, W. Sun, and A. Y. Javaid, "A deep learning based DDoS detection system in software-defined networking (SDN)," 2016, *arXiv:1611.07400*.

[5] D. Charte, F. Charte, S. García, M. J. del Jesus, and F. Herrera, "A practical tutorial on autoencoders for nonlinear feature fusion: Taxonomy, models, software and guidelines," *Informat. Fusion*, vol. 44, pp. 78–96, 2018. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1566253517307844

[6] S. Martínez-Fernández *et al.*, "Software engineering for AI-based systems: A survey," 2021, *arXiv:2105.01984*.

[7] H. Niu, I. Keivanloo, and Y. Zou, "Learning to rank code examples for code search engines," *Empirical Softw. Eng.*, vol. 22, no. 1, pp. 259–291, 2017.

[10] In their study, they used the term replication and reproduction to refer to replicability and reproducibility respectively.

[8] Y. Kim, S. Mun, S. Yoo, and M. Kim, "Precise learn-to-rank fault localization using dynamic and static features of target programs," *ACM Trans. Softw. Eng. Methodol.*, vol. 28, no. 4, Oct. 2019. [Online]. Available: https://doi.org/10.1145/3345628

[9] Y. Tian, D. Wijedasa, D. Lo, and C. Le Goues, "Learning to rank for bug report assignee recommendation," in *Proc. IEEE 24th Int. Conf. Prog. Comprehension*, 2016, pp. 1–10.

[10] A. Bertolino, A. Guerriero, B. Miranda, R. Pietrantuono, and S. Russo, "Learning-to-rank versus ranking-to-learn: Strategies for regression testing in continuous integration," in *Proc. IEEE/ACM 42nd Int. Conf. Softw. Eng.*, 2020, pp. 1–12.

[11] G. Zhao, D. A. da Costa, and Y. Zou, "Improving the pull requests review process using learning-to-rank algorithms," *Empirical Softw. Eng.*, vol. 24, no. 4, pp. 2140–2170, 2019.

[12] A. Perini, F. Ricca, and A. Susi, "Tool-supported requirements prioritization: Comparing the AHP and CBRank methods," *Informat. Softw. Technol.*, vol. 51, no. 6, pp. 1021–1032, 2009. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950584908001717

[13] X. Ye, R. Bunescu, and C. Liu, "Mapping bug reports to relevant files: A ranking model, a fine-grained benchmark, and feature evaluation," *IEEE Trans. Softw. Eng.*, vol. 42, no. 4, pp. 379–402, Apr. 2016.

[14] W. Sun, X. Yan, and A. A. Khan, "Generative ranking based sequential recommendation in software crowdsourcing," in *Proc. Eval. Assessment Softw. Eng.*, 2020, pp. 419–426.

[15] L. Song, L. L. Minku, and X. Yao, "Software effort interval prediction via Bayesian inference and synthetic bootstrap resampling," *ACM Trans. Softw. Eng. Methodol.*, vol. 28, no. 1, pp. 1–46, Jan. 2019.

[16] X. Yu, J. Liu, Z. Yang, X. Jia, Q. Ling, and S. Ye, "Learning from imbalanced data for predicting the number of software defects," in *Proc. IEEE 28th Int. Symp. Softw. Rel. Eng.*, 2017, pp. 78–89.

[17] H. Zhang, L. Gong, and S. Versteeg, "Predicting bug-fixing time: An empirical study of commercial software projects," in *Proc. 35th Int. Conf. Softw. Eng.*, 2013, pp. 1042–1051.

[18] H. Wang, L. Wang, Q. Yu, Z. Zheng, A. Bouguettaya, and M. R. Lyu, "Online reliability prediction via motifs-based dynamic Bayesian networks for service-oriented systems," *IEEE Trans. Softw. Eng.*, vol. 43, no. 6, pp. 556–579, Jun. 2017.

[19] S. Romansky, N. C. Borle, S. Chowdhury, A. Hindle, and R. Greiner, "Deep green: Modelling time-series of software energy consumption," in *Proc. IEEE Int. Conf. Softw. Maintenance Evol.*, 2017, pp. 273–283.

[20] Y. Wan *et al.*, "Improving automatic source code summarization via deep reinforcement learning," in *Proc. 33rd ACM/IEEE Int. Conf. Automated Softw. Eng.*, 2018, pp. 397–407.

[21] P. Yin and G. Neubig, "A syntactic neural model for general-purpose code generation," in *Proc. 55th Annu. Meeting Assoc. Comput. Linguistics*, 2017, pp. 440–450. [Online]. Available: https://www.aclweb.org/anthology/P17–1041

[22] Y. Liang and K. Q. Zhu, "Automatic generation of text descriptive comments for code blocks," in *Proc. 32nd AAAI Conf. Artif. Intell.*, 2018, pp. 5229–5236.

[23] A. L. Samuel, *Some Studies in Machine Learning Using the Game of Checkers. II—Recent Progress*, D. N. L. Levy, Ed., Berlin, Germany: Springer, 1988.

[24] T. M. Mitchell *et al.*, "Machine learning. 1997," New York, NY, USA: McGraw-Hill, 1997, pp. 870–877.

[25] C. M. Bishop, *Pattern Recognition and Machine Learning*. Berlin, Germany: Springer, 2006.

[26] Jason Brownlee, "Difference between classification and regression in machine learning," 2021, [Accessed: Aug. 08, 2021]. [Online]. Available: https://machinelearningmastery.com/classification-versus-regression-in-machine-learning/

[27] Y. Zou, T. Ye, Y. Lu, J. Mylopoulos, and L. Zhang, "Learning to rank for question-oriented software text retrieval (t)," in *Proc. IEEE/ACM 30th Int. Conf. Automated Softw. Eng.*, 2015, pp. 1–11.

[28] T. G. Dietterich, "Ensemble methods in machine learning," in *Proc. Int. Workshop Mult. Classifier Syst.*, 2000, pp. 1–15.

[29] E. Kocaguneli, T. Menzies, and J. W. Keung, "On the value of ensemble effort estimation," *IEEE Trans. Softw. Eng.*, vol. 38, no. 6, pp. 1403–1416, Nov./Dec. 2012.

[30] F. Thung, X.-B. D. Le, and D. Lo, "Active semi-supervised defect categorization," in *Proc. IEEE 23rd Int. Conf. Prog. Comprehension*, 2015, pp. 60–70.

[31] D. A. Cohn, L. E. Atlas, and R. E. Ladner, "Improving generalization with active learning," *Mach. Learn.*, vol. 15, no. 2, pp. 201–221, 1994.

[32] M. Li, H. Zhang, R. Wu, and Z.-H. Zhou, "Sample-based software defect prediction with active and semi-supervised learning," *Automated Softw. Eng.*, vol. 19, no. 2, pp. 201–230, 2012.

[33] P. Jamshidi, N. Siegmund, M. Velez, C. Kästner, A. Patel, and Y. Agarwal, "Transfer learning for performance modeling of configurable systems: An exploratory analysis," in *Proc. IEEE/ACM 32nd Int. Conf. Automated Softw. Eng.*, 2017, pp. 497–508.

[34] R. Krishna, T. Menzies, and W. Fu, "Too much automation? The bellwether effect and its implications for transfer learning," in *Proc. IEEE/ACM 31st Int. Conf. Automated Softw. Eng.*, 2016, pp. 122–131.

[35] F. Rahman, D. Posnett, and P. Devanbu, "Recalling the "imprecision" of cross-project defect prediction," in *Proc. ACM SIGSOFT 20th Int. Symp. Found. Softw. Eng.*, 2012, pp. 1–11.

[36] K.-X. Xue, L. Su, Y.-F. Jia, and K.-Y. Cai, "A neural network approach to forecasting computing-resource exhaustion with workload," in *Proc. 9th Int. Conf. Qual. Softw.*, 2009, pp. 315–324.

[37] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, no. 7553, 2015, Art. no. 436.

[38] A. N. Lam, A. T. Nguyen, H. A. Nguyen, and T. N. Nguyen, "Bug localization with combination of deep learning and information retrieval," in *Proc. IEEE/ACM 25th Int. Conf. Prog. Comprehension*, 2017, pp. 218–229.

[39] S. Wang, T. Liu, and L. Tan, "Automatically learning semantic features for defect prediction," in *Proc. IEEE/ACM 38th Int. Conf. Softw. Eng.*, 2016, pp. 297–308.

[40] C. Chen, T. Su, G. Meng, Z. Xing, and Y. Liu, "From UI design image to GUI skeleton: A neural machine translator to bootstrap mobile GUI implementation," in *Proc. 40th Int. Conf. Softw. Eng.*, 2018, pp. 665–676.

[41] J. Guo, J. Cheng, and J. Cleland-Huang, "Semantically enhanced software traceability using deep learning techniques," in *Proc. IEEE/ACM 39th Int. Conf. Softw. Eng.*, 2017, pp. 3–14.

[42] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Comput.*, vol. 9, no. 8, pp. 1735–1780, 1997.

[43] M. Wen, R. Wu, and S.-C. Cheung, "How well do change sequences predict defects? sequence learning from software changes," *IEEE Trans. Softw. Eng.*, vol. 46, no. 11, pp. 1155–1175, Nov. 2020.

[44] K. Cho *et al.*, "Learning phrase representations using RNN encoder-decoder for statistical machine translation," 2014, *arXiv:1406.1078*.

[45] Y. Wu *et al.*, "SCDetector: Software functional clone detection based on semantic tokens analysis," in *Proc. 35th IEEE/ACM Int. Conf. Automated Softw. Eng.*, 2020, pp. 821–833.

[46] B. Kitchenham and S. Charters, "Guidelines for performing systematic literature reviews in software engineering (version 2.3)," Keele Univ. Univ. Durham, Durham, UK, Tech. Rep., 2007.

[47] H. Zhang, M. A. Babar, and P. Tell, "Identifying relevant studies in software engineering," *Informat. Softw. Technol.*, vol. 53, no. 6, pp. 625–637, 2011. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0950584910002260

[48] R. Parloff, "Why deep learning is suddenly changing your life," 2016. [Online]. Available: http://fortune.com/ai-artificial-intelligence-deep-machine-learning/

[49] T. Xie, S. Thummalapenta, D. Lo, and C. Liu, "Data mining for software engineering," *Comput.*, vol. 42, no. 8, pp. 55–62, Aug. 2009.

[50] D. Gonzalez, T. Zimmermann, and N. Nagappan, "The state of the ML-universe: 10 years of artificial intelligence & machine learning software development on github," in *Proc. 17th Int. Conf. Mining Softw. Repositories*, 2020, pp. 431–442. [Online]. Available: https://doi.org/10.1145/3379597.3387473

[51] E. d. S. Maldonado, E. Shihab, and N. Tsantalis, "Using natural language processing to automatically detect self-admitted technical debt," *IEEE Trans. Softw. Eng.*, vol. 43, no. 11, pp. 1044–1062, Nov. 2017.

[52] X. Xia, D. Lo, S. J. Pan, N. Nagappan, and X. Wang, "HYDRA: Massively compositional model for cross-project defect prediction," *IEEE Trans. Softw. Eng.*, vol. 42, no. 10, pp. 977–998, Oct. 2016.

[53] M. Mirakhorli and J. Cleland-Huang, "Detecting, tracing, and monitoring architectural tactics in code," *IEEE Trans. Softw. Eng.*, vol. 42, no. 3, pp. 205–220, Mar. 2016.

[54] Ieee standard glossary of software engineering terminology, *IEEE Std 610.12–1990*, pp. 1–84, Dec. 1990.

[55] X. Xie, J. W. Ho, C. Murphy, G. Kaiser, B. Xu, and T. Y. Chen, "Testing and validating machine learning classifiers by metamorphic testing," *J. Syst. Softw.*, vol. 84, no. 4, pp. 544–558, 2011. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0164121210003213

[56] M. Unterkalmsteiner, T. Gorschek, A. K. M. M. Islam, C. K. Cheng, R. B. Permadi, and R. Feldt, "Evaluation and measurement of software process improvement—A systematic literature review," *IEEE Trans. Softw. Eng.*, vol. 38, no. 2, pp. 398–424, Mar./Apr. 2012.

[57] J. Cohen, "A coefficient of agreement for nominal scales," *Educ. Psychol. Meas.*, vol. 20, no. 1, pp. 37–46, 1960.

[58] J. R. Landis and G. G. Koch, "The measurement of observer agreement for categorical data," *Biometrics*, vol. 33, pp. 159–174, 1977.

[59] B. Kitchenham, "Procedures for performing systematic reviews," Keele, UK, Keele University, vol. 33, no. 2004, pp. 1–26, 2004.

[60] L. Yang *et al.*, "Quality assessment in systematic literature reviews: A software engineering perspective," *Informat. Softw. Technol.*, vol. 130, 2021, Art. no. 106397. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950584920301610

[61] X. Huang, H. Zhang, X. Zhou, M. Ali Babar, and S. Yang, "Synthesizing qualitative research in software engineering: A critical review," in *Proc. IEEE/ACM 40th Int. Conf. Softw. Eng.*, 2018, pp. 1207–1218.

[62] J. M. González-Barahona and G. Robles, "On the reproducibility of empirical software engineering studies based on data retrieved from development repositories," *Empir. Softw. Eng.*, vol. 17, no. 1, pp. 75–89, 2012.

[63] S. Amershi *et al.*, "Software engineering for machine learning: A case study," in *Proc. 41st Int. Conf. Softw. Eng. Softw. Eng. Pract.*, 2019, pp. 291–300. [Online]. Available: https://doi.org/10.1109/ICSE-SEIP.2019.00042

[64] J. Saldana, *Fundamentals of Qualitative Research*. New York, NY, USA: Oxford Univ. Press, 2011.

[65] S. Wang, "Supplemental data for TSE paper," Apr. 2022. [Online]. Available: https://doi.org/10.5281/zenodo.5977109

[66] M. White, C. Vendome, M. Linares-Vásquez, and D. Poshyvanyk, "Toward deep learning software repositories," in *Proc. 12th Work. Conf. Mining Softw. Repositories*, 2015, pp. 334–345. [Online]. Available: http://dl.acm.org/citation.cfm?id=2820518.2820559

[67] L. Bao, Z. Xing, X. Xia, D. Lo, and A. E. Hassan, "Inference of development activities from interaction with uninstrumented applications," *Empir. Softw. Eng.*, vol. 23, no. 3, pp. 1313–1351, Jun. 2018. [Online]. Available: https://doi.org/10.1007/s10664–017-9547-8

[68] D. Girardi, N. Novielli, D. Fucci, and F. Lanubile, "Recognizing developers' emotions while programming," in *Proc. IEEE/ACM 42nd Int. Conf. Softw. Eng.*, 2020, pp. 666–677. [Online]. Available: https://doi.org/10.1145/3377811.3380374

[69] K. Blincoe, G. Valetto, and D. Damian, "Facilitating coordination between software developers: A study and techniques for timely and efficient recommendations," *IEEE Trans. Softw. Eng.*, vol. 41, no. 10, pp. 969–985, Oct. 2015.

[70] L. Bao, Z. Xing, X. Xia, D. Lo, and S. Li, "Who will leave the company?: A large-scale industry study of developer turnover by mining monthly work report," in *Proc. IEEE/ACM 14th Int. Conf. Mining Softw. Repositories*, 2017, pp. 170–181.

[71] M. Bhat, K. Shumaiev, K. Koch, U. Hohenstein, A. Biesdorf, and F. Matthes, "An expert recommendation system for design decision making: Who should be involved in making a design decision?," in *Proc. IEEE Int. Conf. Softw. Archit.*, 2018, pp. 85–8509.

[72] X. Ye, H. Shen, X. Ma, R. Bunescu, and C. Liu, "From word embeddings to document similarities for improved information retrieval in software engineering," in *Proc. 38th Int. Conf. Softw. Eng.*, 2016, pp. 404–415. [Online]. Available: https://doi.org/10.1145/2884781.2884862

[73] J. Wang, M. Li, S. Wang, T. Menzies, and Q. Wang, "Images don't lie: Duplicate crowdtesting reports detection with screenshot information," *Informat. Softw. Technol.*, vol. 110, pp. 139–155, 2019. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950584919300503

[74] Y. Wahba, N. H. Madhavji, and J. Steinbacher, *Evaluating the Effectiveness of Static Word Embeddings on the Classification of IT Support Tickets*. Armonk, NY, USA: IBM Corp., 2020, pp. 198–206.

[75] Z. Liu, X. Xia, D. Lo, and J. Grundy, "Automatic, highly accurate app permission recommendation," *Automated Softw. Eng.*, vol. 26, no. 2, pp. 241–274, 2019.

[76] T. Mikolov, I. Sutskever, K. Chen, G. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," in *Proc. Int. Conf. Neural Inf. Process. Syst.*, 2013, pp. 3111–3119.

[77] J. Pennington, R. Socher, and C. D. Manning, "GloVe: Global vectors for word representation," in *Proc. Conf. Empir. Methods Natural Lang. Process.*, 2014, pp. 1532–1543.

[78] D. Ye, Z. Xing, C. Y. Foo, J. Li, and N. Kapre, "Learning to extract API mentions from informal natural language discussions," in *Proc. IEEE Int. Conf. Softw. Maintenance Evol.*, 2016, pp. 389–399.

[79] Z. Han, X. Li, H. Liu, Z. Xing, and Z. Feng, "Deepweak: Reasoning common software weaknesses via knowledge graph embedding," in *Proc. IEEE 25th Int. Conf. Softw. Anal., Evol. Reengineering*, 2018, pp. 456–466.

[80] S. Reddy, C. Lemieux, R. Padhye, and K. Sen, "Quickly generating diverse valid test inputs with reinforcement learning," in *Proc. IEEE/ACM 42nd Int. Conf. Softw. Eng.*, 2020, pp. 1410–1421.

[81] S. S. Emam and J. Miller, "Test case prioritization using extended digraphs," *ACM Trans. Softw. Eng. Methodol.*, vol. 25, no. 1, pp. 1–41, Dec. 2015. [Online]. Available: https://doi.org/10.1145/2789209

[82] S. Carino and J. H. Andrews, "Dynamically testing GUIs using ant colony optimization (t)," in *Proc. IEEE/ACM 30th Int. Conf. Automated Softw. Eng.*, 2015, pp. 138–148.

[83] A. Barriga, R. Heldal, L. Iovino, M. Marthinsen, and A. Rutle, "An extensible framework for customizable model repair," in *Proc. IEEE/ACM 23rd Int. Conf. Model Driven Eng. Lang. Syst.*, 2020, pp. 24–34. [Online]. Available: https://doi.org/10.1145/3365438.3410957

[84] Z. Wu *et al.*, "REINAM: Reinforcement learning for input-grammar inference," in *Proc. 27th ACM Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, 2019, pp. 488–498. [Online]. Available: https://doi.org/10.1145/3338906.3338958

[85] S. Hönel, M. Ericsson, W. Löwe, and A. Wingkvist, "Using source code density to improve the accuracy of automatic commit classification into maintenance activities," *J. Syst. Softw.*, vol. 168, 2020, Art. no. 110673. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121220301291

[86] X. Chen *et al.*, "A systemic framework for crowdsourced test report quality assessment," *Empir. Softw. Eng.*, vol. 25, no. 2, pp. 1382–1418, 2020.

[87] Z. Zhang, H. Sun, and H. Zhang, "Developer recommendation for topcoder through a meta-learning based policy model," *Empir. Softw. Eng.*, vol. 25, no. 1, pp. 859–889, 2020.

[88] T. Hey, J. Keim, A. Koziolek, and W. F. Tichy, "Norbert: Transfer learning for requirements classification," in *Proc. IEEE 28th Int. Requirements Eng. Conf.*, 2020, pp. 169–179.

[89] A. Sainani, P. R. Anish, V. Joshi, and S. Ghaisas, "Extracting and classifying requirements from software engineering contracts," in *Proc. IEEE 28th Int. Requirements Eng. Conf.*, 2020, pp. 147–157.

[90] L. Shi, M. Xing, M. Li, Y. Wang, S. Li, and Q. Wang, "Detection of hidden feature requests from massive chat messages via deep siamese network," in *Proc. IEEE/ACM 42nd Int. Conf. Softw. Eng.*, 2020, pp. 641–653.

[91] M. Li, L. Shi, Y. Yang, and Q. Wang, "A deep multitask learning approach for requirements discovery and annotation from open forum," in *Proc. IEEE/ACM 35th Int. Conf. Automated Softw. Eng.*, 2020, pp. 336–348.

[92] S. Fakhoury, V. Arnaoudova, C. Noiseux, F. Khomh, and G. Antoniol, "Keep it simple: Is deep learning good for linguistic smell detection?," in *Proc. IEEE 25th Int. Conf. Softw. Anal., Evol. ReEng.*, 2018, pp. 602–611.

[93] M. Hadj-Kacem and N. Bouassida, "Deep representation learning for code smells detection using variational auto-encoder," in *Proc. Int. Joint Conf. Neural Netw.*, 2019, pp. 1–8.

[94] F. Zampetti, A. Serebrenik, and M. Di Penta, "Automatically learning patterns for self-admitted technical debt removal," in *Proc. IEEE 27th Int. Conf. Softw. Anal., Evol. Reeng.*, 2020, pp. 355–366.

[95] Y. Chen, C. M. Poskitt, J. Sun, S. Adepu, and F. Zhang, "Learning-guided network fuzzing for testing cyber-physical system defences," in *Proc. IEEE/ACM 34th Int. Conf. Automated Softw. Eng.*, 2019, pp. 962–973.

[96] T. T. Chekam, M. Papadakis, T. F. Bissyandé, Y. Le Traon, and K. Sen, "Selecting fault revealing mutants," *Empir. Softw. Eng.*, vol. 25, no. 1, pp. 434–487, 2020.

[97] D. Mao, L. Chen, and L. Zhang, "An extensive study on cross-project predictive mutation testing," in *Proc. IEEE 12th Conf. Softw. Testing Validation Verification*, 2019, pp. 160–171.

[98] Y. Zheng *et al.*, "Wuji: Automatic online combat game testing using evolutionary deep reinforcement learning," in *Proc. 34th IEEE/ACM Int. Conf. Automated Softw. Eng.*, 2019, pp. 772–784.

[99] M. Tufano, C. Watson, G. Bavota, M. Di Penta, M. White, and D. Poshyvanyk, "Learning how to mutate source code from bug-fixes," in *Proc. IEEE Int. Conf. Softw. Maintenance Evol.*, 2019, pp. 301–312.

[100] M. Tan, L. Tan, S. Dara, and C. Mayeux, "Online defect prediction for imbalanced data," in *Proc. IEEE/ACM 37th Int. Conf. Softw. Eng.*, 2015, pp. 99–108.

[101] F. Zhang, A. E. Hassan, S. McIntosh, and Y. Zou, "The use of summation to aggregate software metrics hinders the performance of defect prediction models," *IEEE Trans. Softw. Eng.*, vol. 43, no. 5, pp. 476–491, May 2017.

[102] J. Nam, S. J. Pan, and S. Kim, "Transfer defect learning," in *Proc. 35th Int. Conf. Softw. Eng.*, 2013, pp. 382–391.

[103] J. Chen *et al.*, "Software visualization and deep transfer learning for effective software defect prediction," in *Proc. ACM/IEEE 42nd Int. Conf. Softw. Eng.*, 2020, pp. 578–589. [Online]. Available: https://doi.org/10.1145/3377811.3380389

[104] S. Wang, T. Liu, J. Nam, and L. Tan, "Deep semantic feature learning for software defect prediction," *IEEE Trans. Softw. Eng.*, vol. 46, no. 12, pp. 1267–1293, Dec. 2020.

[105] Z. Xu *et al.*, "LDFR: Learning deep feature representation for software defect prediction," *J. of Syst. Softw.*, vol. 158, 2019, Art. no. 110402. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121219301761

[106] K. Zhu, " Within-project and cross-project just-in-time defect prediction based on denoising autoencoder and convolutional neural network," *IET Softw.*, vol. 14, pp. 185–195, Jun. 2020. [Online]. Available: https://digital-library.theiet.org/content/journals/10.1049/iet-sen.2019.0278

[107] T. Zhou, X. Sun, X. Xia, B. Li, and X. Chen, "Improving defect prediction with deep forest," *Informat. Softw. Technol.*, vol. 114, pp. 204–216, 2019. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950584919301466

[108] J. Xu, F. Wang, and J. Ai, "Defect prediction with semantics and context features of codes based on graph representation learning," *IEEE Trans. Rel.*, vol. 70, no. 2, pp. 613–625, Jun. 2021.

[109] K. Heo, H. Oh, and K. Yi, "Machine-learning-guided selectively unsound static analysis," in *Proc. IEEE/ACM 39th Int. Conf. Softw. Eng.*, 2017, pp. 519–529.

[110] R. Natella, D. Cotroneo, J. A. Duraes, and H. S. Madeira, "On fault representativeness of software fault injection," *IEEE Trans. Softw. Eng.*, vol. 39, no. 1, pp. 80–96, Jan. 2013.

[111] D. Kim, Y. Tao, S. Kim, and A. Zeller, "Where should we fix this bug? a two-phase recommendation model," *IEEE Trans. Softw. Eng.*, vol. 39, no. 11, pp. 1597–1610, Nov. 2013.

[112] D. Mu *et al.*, "Renn: Efficient reverse execution with neural-network-assisted alias analysis," in *Proc. IEEE/ACM 34th Int. Conf. Automated Softw. Eng.*, 2019, pp. 924–935.

[113] Z. Zhang, Y. Lei, X. Mao, and P. Li, "CNN-FL: An effective approach for localizing faults using convolutional neural networks," in *Proc. IEEE 26th Int. Conf. Softw. Anal., Evol. Reengineering*, 2019, pp. 445–455.

[114] R. Kapur and B. Sodhi, "A defect estimator for source code: Linking defect reports with programming constructs usage metrics," *ACM Trans. Softw. Eng. Methodol.*, vol. 29, no. 2, pp. 1–35, Apr. 2020. [Online]. Available: https://doi.org/10.1145/3384517

[115] Y. Lin, J. Sun, L. Tran, G. Bai, H. Wang, and J. Dong, "Break the dead end of dynamic slicing: Localizing data and control omission bug," in *Proc. IEEE/ACM 33rd Int. Conf. Automated Softw. Eng.*, 2018, pp. 509–519. [Online]. Available: https://doi.org/10.1145/3238147.3238163

[116] Y. Li, S. Wang, T. N. Nguyen, and S. Van Nguyen, "Improving bug detection via context-based code representation learning and attention-based neural networks," *Proc. ACM Prog.. Lang.*, vol. 3, no. OOPSLA, pp. 1–30, Oct. 2019. [Online]. Available: https://doi.org/10.1145/3360588

[117] J. Zhang, R. Xie, W. Ye, Y. Zhang, and S. Zhang, *Exploiting Code Knowl. Graph for Bug Localization via Bi-Directional Attention*. New York, NY, USA: Association for Computing Machinery, 2020, pp. 219–229. [Online]. Available: https://doi.org/10.1145/3387904.3389281

[118] J. Zhang, X. Wang, H. Zhang, H. Sun, Y. Pu, and X. Liu, "Learning to handle exceptions," in *Proc. IEEE/ACM 35th Int. Conf. Automated Softw. Eng.*, 2020, pp. 29–41.

[119] H. Liang, L. Sun, M. Wang, and Y. Yang, "Deep learning with customized abstract syntax tree for bug localization," *IEEE Access*, vol. 7, pp. 116 309–116 320, 2019.

[120] J. Chen, H. Ma, and L. Zhang, *Enhanced Compiler Bug Isolation via Memoized Search*. New York, NY, USA: Association for Computing Machinery, 2020, pp. 78–89. [Online]. Available: https://doi.org/10.1145/3324884.3416570

[121] Y. Xiao, J. Keung, K. E. Bennin, and Q. Mi, "Improving bug localization with word embedding and enhanced convolutional neural networks," *Informat. Softw. Technol.*, vol. 105, pp. 17–29, 2019. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950584918301654

[122] Z. Kurtanović and W. Maalej, "Mining user rationale from software reviews," in *Proc. IEEE 25th Int. Requirements Eng. Conf.*, 2017, pp. 61–70.

[123] B. Lin, F. Zampetti, G. Bavota, M. Di Penta, M. Lanza, and R. Oliveto, "Sentiment analysis for software engineering: How far can we go?," in *Proc. 40th Int. Conf. Softw. Eng.*, 2018, pp. 94–104. [Online]. Available: https://doi.org/10.1145/3180155.3180195

[124] Z. Qian, B. Shen, W. Mo, and Y. Chen, "Satiindicator: Leveraging user reviews to evaluate user satisfaction of sourceforge projects," in *Proc. IEEE 40th Annu. Comput. Softw. Appl. Conf.*, 2016, pp. 93–102.

[125] C. Gao, J. Zeng, X. Xia, D. Lo, M. R. Lyu, and I. King, "Automating app review response generation," in *Proc. 34th IEEE/ACM Int. Conf. Automated Softw. Eng.*, 2019, pp. 163–175.

[126] Y. Gu, K. Yang, S. Fu, S. Chen, X. Li, and I. Marsic, "Multimodal affective analysis using hierarchical attention strategy with word-level alignment," in *Proc. 56th Annu. Meeting Assoc. Comput. Linguistics*, 2018, pp. 2225–2235. [Online]. Available: https://www.aclweb.org/anthology/P18–1207

[127] F. Pecorelli, F. Palomba, D. Di Nucci, and A. De Lucia, "Comparing heuristic and machine learning approaches for metric-based code smell detection," in *Proc. IEEE/ACM 27th Int. Conf. Prog. Comprehension*, 2019, pp. 93–104.

[128] M. White, M. Tufano, C. Vendome, and D. Poshyvanyk, "Deep learning code fragments for code clone detection," in *Proc. 31st IEEE/ACM Int. Conf. Automated Softw. Eng.*, 2016, pp. 87–98. [Online]. Available: http://doi.acm.org/10.1145/2970276.2970326

[129] B. Liu *et al.*, "DIFF: Cross-version binary code similarity detection with dnn," in *Proc. IEEE/ACM 33rd Int. Conf. Automated Softw. Eng.*, 2018, pp. 667–678. [Online]. Available: https://doi.org/10.1145/3238147.3238199

[130] V. Saini, F. Farmahinifarahani, Y. Lu, P. Baldi, and C. V. Lopes, "Oreo: Detection of clones in the twilight zone," in *Proc. 26th ACM Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, 2018, Art. no. 354365. [Online]. Available: https://doi.org/10.1145/3236024.3236026

[131] M. Tufano, C. Watson, G. Bavota, M. Di Penta, M. White, and D. Poshyvanyk, "Deep learning similarities from different representations of source code," in *Proc. IEEE/ACM 15th Int. Conf. Mining Softw. Repositories*, 2018, pp. 542–553.

[132] W. Wang, G. Li, B. Ma, X. Xia, and Z. Jin, "Detecting code clones with graph neural network and flow-augmented abstract syntax tree," in *Proc. IEEE 27th Int. Conf. Softw. Anal., Evol. Reengineering*, 2020, pp. 261–271.

[133] R. Scandariato, J. Walden, A. Hovsepyan, and W. Joosen, "Predicting vulnerable software components via text mining," *IEEE Trans. Softw. Eng.*, vol. 40, no. 10, pp. 993–1006, Oct. 2014.

[134] W. Zheng *et al.*, "The impact factors on the performance of machine learning-based vulnerability detection: A comparative study," *J. Syst. Softw.*, vol. 168, 2020, Art. no. 110659. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121220301229

[135] Z. Han, X. Li, Z. Xing, H. Liu, and Z. Feng, "Learning to predict severity of software vulnerability using only vulnerability description," in *Proc. IEEE Int. Conf. Softw. Maintenance Evol.*, 2017, pp. 125–136.

[136] Z. Chen, H. Wang, C. Xu, X. Ma, and C. Cao, "Vision: Evaluating scenario suitableness for DNN models by mirror synthesis," in *Proc. 26th Asia-Pacific Softw. Eng. Conf.*, 2019, pp. 78–85.

[137] M. J. Mashhadi and H. Hemmati, "Hybrid deep neural networks to infer state models of black-box systems," in *Proc. IEEE/ACM 35th Int. Conf. Automated Softw. Eng.*, 2020, pp. 299–311.

[138] H. J. Kang, T. F. Bissyand, and D. Lo, "Assessing the generalizability of code2vec token embeddings," in *Proc. IEEE/ACM 34th Int. Conf. Automated Softw. Eng.*, 2019, pp. 1–12.

[139] T. Le *et al.*, "Maximal divergence sequential autoencoder for binary software vulnerability detection," in *Proc. Int. Conf. Learn. Representations*, 2019, p. 15. [Online]. Available: https://openreview.net/forum?id=ByloIiCqYQ

[140] G. Tang *et al.*, "A comparative study of neural network techniques for automatic software vulnerability detection," in *Proc. Int. Symp. Theor. Aspects Softw. Eng.*, 2020, pp. 1–8.

[141] Y. Fan, X. Xia, D. Lo, and A. E. Hassan, "Chaff from the wheat: Characterizing and determining valid bug reports," *IEEE Trans. Softw. Eng.*, vol. 46, no. 5, pp. 495–525, May 2020.

[142] X. Ye, F. Fang, J. Wu, R. Bunescu, and C. Liu, "Bug report classification using LSTM architecture for more accurate software defect locating," in *Proc. IEEE 17th Int. Conf. Mach. Learn. Appl.*, 2018, pp. 1438–1445.

[143] J. He, L. Xu, M. Yan, X. Xia, and Y. Lei, "Duplicate bug report detection using dual-channel convolutional neural networks," in *Proc. 28th Int. Conf. Prog. Comprehension*, 2020, Art. no. 117127. [Online]. Available: https://doi.org/10.1145/3387904.3389263

[144] B. Soleimani Neysiani, S. M. Babamir, and M. Aritsugi, "Efficient feature extraction model for validation performance improvement of duplicate bug report detection in software bug triage systems," *Informat. Softw. Technol.*, vol. 126, 2020, Art. no. 106344. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950584920301117

[145] U. Koc, S. Wei, J. S. Foster, M. Carpuat, and A. A. Porter, "An empirical assessment of machine learning approaches for triaging reports of a java static analysis tool," in *Proc. IEEE 12th Conf. Softw. Testing, Validation Verification*, 2019, pp. 288–299.

[146] O. Chaparro *et al.*, "Assessing the quality of the steps to reproduce in bug reports," in *Proc. 27th ACM Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, 2019, Art. no. 8696. [Online]. Available: https://doi.org/10.1145/3338906.3338947

[147] I. M. Rodrigues, D. Aloise, E. R. Fernandes, and M. Dagenais, *A Soft Alignment Model for Bug Deduplication*. New York, NY, USA: Association for Computing Machinery, 2020, Art. no. 4353. [Online]. Available: https://doi.org/10.1145/3379597.3387470

[148] J. Anvik and G. C. Murphy, "Reducing the effort of bug report triage: Recommenders for development-oriented decisions," *ACM Trans. Softw. Eng. Methodol.*, vol. 20, no. 3, pp. 1–35, Aug. 2011. [Online]. Available: https://doi.org/10.1145/2000791.2000794

[149] S. F. A. Zaidi, F. M. Awan, M. Lee, H. Woo, and C.-G. Lee, "Applying convolutional neural networks with different word representation techniques to recommend bug fixers," *IEEE Access*, vol. 8, pp. 213 729–213 747, 2020.

[150] W. Zhang, "Efficient bug triage for industrial environments," in *Proc. IEEE Int. Conf. Softw. Maintenance Evol.*, 2020, pp. 727–735.

[151] C. Vendome, M. Linares-Vásquez, G. Bavota, M. Di Penta, D. German, and D. Poshyvanyk, "Machine learning-based detection of open source license exceptions," in *Proc. IEEE/ACM 39th Int. Conf. Softw. Eng.*, 2017, pp. 118–129.

[152] X. Zhang *et al.*, "Robust log-based anomaly detection on unstable log data," in *Proc. 27th ACM Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, 2019, pp. 807–817. [Online]. Available: https://doi.org/10.1145/3338906.3338931

[153] Z. Liu, C. Chen, J. Wang, Y. Huang, J. Hu, and Q. Wang, "Owl eyes: Spotting ui display issues via visual understanding," in *Proc. IEEE/ACM 35th Int. Conf. Automated Softw. Eng.*, 2020, pp. 398–409.

[154] R. Yan, X. Xiao, G. Hu, S. Peng, and Y. Jiang, "New deep learning method to detect code injection attacks on hybrid applications," *J. Syst. Softw.*, vol. 137, pp. 67–77, 2018. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0164121217302571

[155] H. Xia *et al.*, "How android developers handle evolution-induced API compatibility issues: A large-scale study," in *Proc. IEEE/ACM 42nd Int. Conf. Softw. Eng.*, 2020, pp. 886–898.

[156] Z. Liu, X. Xia, A. E. Hassan, D. Lo, Z. Xing, and X. Wang, "Neural-machine-translation-based commit message generation: How far are we?," in *Proc. IEEE/ACM 33rd Int. Conf. Automated Softw. Eng.*, 2018, pp. 373–384. [Online]. Available: http://doi.acm.org/10.1145/3238147.3238190

[157] S. Iyer, I. Konstas, A. Cheung, and L. Zettlemoyer, "Summarizing source code using a neural attention model," in *Proc. 54th Annu. Meeting Assoc. Comput. Linguistics*, 2016, pp. 2073–2083. [Online]. Available: https://www.aclweb.org/anthology/P16–1195

[158] S. Jiang, A. Armaly, and C. McMillan, "Automatically generating commit messages from diffs using neural machine translation," in *Proc. IEEE/ACM 32nd Int. Conf. Automated Softw. Eng.*, 2017, pp. 135–146. [Online]. Available: http://dl.acm.org/citation.cfm?id=3155562.3155583

[159] A. LeClair, S. Haque, L. Wu, and C. McMillan, *Improved Code Summarization via a Graph Neural Network*. New York, NY, USA: Association for Computing Machinery, 2020, pp. 184–195. [Online]. Available: https://doi.org/10.1145/3387904.3389268

[160] W. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang, "A transformer-based approach for source code summarization," in *Proc. 58th Annu. Meeting Assoc. Comput. Linguistics*, 2020, pp. 4998–5007. [Online]. Available: https://aclanthology.org/2020.acl-main.449

[161] Z. Zhou, H. Yu, and G. Fan, "Effective approaches to combining lexical and syntactical information for code summarization," *Softw. Pract. Experience*, vol. 50, no. 12, pp. 2313–2336, 2020. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.2893

[162] M. Allamanis, H. Peng, and C. Sutton, "A convolutional attention network for extreme summarization of source code," in *Proc. 33rd Int. Conf. Mach. Learn.*, 2016, pp. 2091–2100. [Online]. Available: http://proceedings.mlr.press/v48/allamanis16.html

[163] P. Bielik, V. Raychev, and M. Vechev, "PHOG: Probabilistic model for code," in *Proc. 33rd Int. Conf. Mach. Learn.*, 2016, pp. 2933–2942. [Online]. Available: https://proceedings.mlr.press/v48/bielik16.html

[164] S. Han, D. R. Wallace, and R. C. Miller, "Code completion from abbreviated input," in *Proc. IEEE/ACM Int. Conf. Automated Softw. Eng.*, 2009, pp. 332–343.

[165] V. J. Hellendoorn, S. Proksch, H. C. Gall, and A. Bacchelli, "When code completion fails: A case study on real-world completions," in *Proc. IEEE/ACM 41st Int. Conf. Softw. Eng.*, 2019, pp. 960–970.

[166] F. Liu, G. Li, Y. Zhao, and Z. Jin, *Multi-Task Learning Based Pre-Trained Language Model for Code Completion*. New York, NY, USA: Association for Computing Machinery, 2020, pp. 473–485. [Online]. Available: https://doi.org/10.1145/3324884.3416591

[167] W. Ling *et al.*, "Latent predictor networks for code generation," in *Proc. 54th Annu. Meeting Assoc. Comput. Linguistics*, 2016, pp. 599–609. [Online]. Available: https://www.aclweb.org/anthology/P16–1057

[168] Z. Sun, Q. Zhu, L. Mou, Y. Xiong, G. Li, and L. Zhang, "A grammar-based structural CNN decoder for code generation," in *Proc. AAAI Conf. Artif. Intell.*, 2019, pp. 7055–7062. [Online]. Available: https://ojs.aaai.org/index.php/AAAI/article/view/4686

[169] J. Lacomis *et al.*, "Dire: A neural approach to decompiled identifier naming," in *Proc. IEEE/ACM 34th Int. Conf. Automated Softw. Eng.*, 2019, pp. 628–639.

[170] Y. Jia, M. B. Cohen, M. Harman, and J. Petke, "Learning combinatorial interaction test generation strategies using hyperheuristic search," in *Proc. IEEE/ACM 37th IEEE Int. Conf. Softw. Eng.*, 2015, pp. 540–550.

[171] P. Godefroid, H. Peleg, and R. Singh, "Learn&fuzz: Machine learning for input fuzzing," in *Proc. IEEE/ACM 32nd Int. Conf. Automated Softw. Eng.*, 2017, pp. 50–59.

[172] X. Liu, X. Li, R. Prajapati, and D. Wu, "DeepFuzz: Automatic generation of syntax valid C programs for fuzz testing," in *Proc. AAAI Conf. Artif. Intell.*, 2019, pp. 1044–1051. [Online]. Available: https://ojs.aaai.org/index.php/AAAI/article/view/3895

[173] M. Liu, K. Li, and T. Chen, "DeepSQLi: Deep semantic learning for testing SQL injection," in *Proc. 29th ACM SIGSOFT Int. Symp. Softw. Testing Anal.*, 2020, pp. 286–297. [Online]. Available: https://doi.org/10.1145/3395363.3397375

[174] Z. Li, H. Zhao, J. Shi, Y. Huang, and J. Xiong, "An intelligent fuzzing data generation method based on deep adversarial learning," *IEEE Access*, vol. 7, pp. 49 327–49 340, 2019.

[175] T. Ahmad, A. Ashraf, D. Truscan, A. Domi, and I. Porres, "Using deep reinforcement learning for exploratory performance testing of software systems with multi-dimensional input spaces," *IEEE Access*, vol. 8, pp. 195 000–195 020, 2020.

[176] S. Bhatia, P. Kohli, and R. Singh, "Neuro-symbolic program corrector for introductory programming assignments," in *Proc. IEEE/ACM 40th Int. Conf. Softw. Eng.*, 2018, pp. 60–70.

[177] Z. Chen, S. Kommrusch, M. Tufano, L.-N. Pouchet, D. Poshyvanyk, and M. Monperrus, "Sequencer: Sequence-to-sequence learning for end-to-end program repair," *IEEE Trans. Softw. Eng.*, vol. 47, no. 9, pp. 1943–1959, Sep. 2021.

[178] L. Wu, F. Li, Y. Wu, and T. Zheng, *GGF: A Graph-Based Method for Program. Lang. Syntax Error Correction.* New York, NY, USA: Association for Computing Machinery, 2020, pp. 139–148. [Online]. Available: https://doi.org/10.1145/3387904.3389252

[179] Y. Li, S. Wang, and T. N. Nguyen, "DLFix: Context-based code transformation learning for automated program repair," in *Proc. IEEE/ACM 42nd Int. Conf. Softw. Eng.*, 2020, pp. 602–614. [Online]. Available: https://doi.org/10.1145/3377811.3380345

[180] M. White, M. Tufano, M. Martínez, M. Monperrus, and D. Poshyvanyk, "Sorting and transforming program repair ingredients via deep learning code similarities," in *Proc. IEEE 26th Int. Conf. Softw. Anal., Evol. ReEng.*, 2019, pp. 479–490.

[181] H. Tian *et al.*, "Evaluating representation learning of code changes for predicting patch correctness in program repair," in *Proc. IEEE/ACM 35th Int. Conf. Automated Softw. Eng.*, 2020, pp. 981–992.

[182] T. McCabe, "A complexity measure," *IEEE Trans. Softw. Eng.*, vol. SE-2, no. 4, pp. 308–320, Dec. 1976.

[183] B. Ghotra, S. McIntosh, and A. E. Hassan, "Revisiting the impact of classification techniques on the performance of defect prediction models," in *Proc. 37th Int. Conf. Softw. Eng.*, 2015, pp. 789–800. [Online]. Available: http://dl.acm.org/citation.cfm?id=2818754.2818850

[184] X. Yang, D. Lo, X. Xia, Y. Zhang, and J. Sun, "Deep learning for just-in-time defect prediction," in *Proc. IEEE Int. Conf. Softw. Qual., Rel. Secur.*, 2015, pp. 17–26.

[185] J. Li, P. He, J. Zhu, and M. R. Lyu, "Software defect prediction via convolutional neural network," in *Proc. IEEE Int. Conf. Softw. Qual., Rel. Secur.*, 2017, pp. 318–328.

[186] J. Li, P. He, J. Zhu, and M. R. Lyu, "Software defect prediction via convolutional neural network," in *Proc. IEEE Int. Conf. Softw. Qual., Rel. Secur.*, 2017, pp. 318–328.

[187] L. Li, H. Feng, W. Zhuang, N. Meng, and B. Ryder, "Cclearner: A deep learning-based clone detection approach," in *Proc. IEEE Int. Conf. Softw. Maintenance Evol.*, 2017, pp. 249–260.

[188] A. Viet Phan, M. Le Nguyen, and L. Thu Bui, "Convolutional neural networks over control flow graphs for software defect prediction," in *Proc. IEEE 29th Int. Conf. Tools Artif. Intell.*, 2017, pp. 45–52.

[189] Y. Xiao, J. Keung, Q. Mi, and K. E. Bennin, "Bug localization with semantic and structural features using convolutional neural network and cascade forest," in *Proc. 22nd Int. Conf. Eval. Assessment Softw. Eng.*, 2018, pp. 101–111. [Online]. Available: https://doi.org/10.1145/3210459.3210469

[190] Y. Xiao, J. Keung, Q. Mi, and K. E. Bennin, "Improving bug localization with an enhanced convolutional neural network," in *Proc. 24th Asia-Pacific Softw. Eng. Conf.*, 2017, pp. 338–347.

[191] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," 2013, *arXiv: 1301.3781*.

[192] P. Bojanowski, E. Grave, A. Joulin, and T. Mikolov, "Enriching word vectors with subword information," *Trans. Assoc. Comput. Linguistics*, vol. 5, pp. 135–146, 2017.

[193] X. Wang, J. Liu, L. Li, X. Chen, X. Liu, and H. Wu, "Detecting and explaining self-admitted technical debts with attention-based neural networks," in *Proc. IEEE/ACM 35th Int. Conf. Automated Softw. Eng.*, 2020, pp. 871–882.

[194] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "Code2vec: Learning distributed representations of code," *Proc. ACM Prog. Lang.*, vol. 3, no. POPL, pp. 1–29, Jan. 2019. [Online]. Available: https://doi.org/10.1145/3290353

[195] Q. Le and T. Mikolov, "Distributed representations of sentences and documents," in *Proc. 31st Int. Conf. Mach. Learn.*, 2014, pp. 1188–1196. [Online]. Available: http://proceedings.mlr.press/v32/le14.html

[196] T. Hoang, H. J. Kang, D. Lo, and J. Lawall, "Cc2vec: Distributed representations of code changes," in *Proc. IEEE/ACM 42nd Int. Conf. Softw. Eng.*, 2020, pp. 518–529. [Online]. Available: https://doi.org/10.1145/3377811.3380361

[197] Y. Wang, L. Shi, M. Li, Q. Wang, and Y. Yang, "A deep context-wise method for coreference detection in natural language requirements," in *Proc. IEEE 28th Int. Requirements Eng. Conf.*, 2020, pp. 180–191.

[198] M. Alahmadi, A. Khormi, B. Parajuli, J. Hassel, S. Haiduc, and P. Kumar, "Code localization in programming screencasts," *Empir. Softw. Eng.*, vol. 25, no. 2, pp. 1536–1572, 2020.

[199] J. Ott, A. Atchison, P. Harnack, A. Bergh, and E. Linstead, "A deep learning approach to identifying source code in images and video," in *Proc. IEEE/ACM 15th Int. Conf. Mining Softw. Repositories*, 2018, pp. 376–386.

[200] L. Bao, Z. Xing, X. Xia, D. Lo, M. Wu, and X. Yang, "Psc2code: Denoising code extraction from programming screencasts," *ACM Trans. Softw. Eng. Methodol.*, vol. 29, no. 3, pp. 1–38, Jun. 2020. [Online]. Available: https://doi.org/10.1145/3392093

[201] J. Chen *et al.*, "Object detection for graphical user interface: Old fashioned or deep learning or a combination?," in *Proc. 28th ACM Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, 2020, pp. 1202–1214. [Online]. Available: https://doi.org/10.1145/3368089.3409691

[202] C. Bernal-Cárdenas, N. Cooper, K. Moran, O. Chaparro, A. Marcus, and D. Poshyvanyk, "Translating video recordings of mobile app usages into replayable scenarios," in *Proc. IEEE/ACM 42nd Int. Conf. Softw. Eng.*, 2020, pp. 309–321. [Online]. Available: https://doi.org/10.1145/3377811.3380328

[203] T. D. White, G. Fraser, and G. J. Brown, "Improving random GUI testing with image-based widget detection," in *Proc. 28th ACM SIGSOFT Int. Symp. Softw. Testing Anal.*, 2019, pp. 307–317. [Online]. Available: https://doi.org/10.1145/3293882.3330551

[204] D. Zhao *et al.*, "Seenomaly: Vision-based linting of GUI animation effects against design-don't guidelines," in *Proc. IEEE/ACM 42nd Int. Conf. Softw. Eng.*, 2020, pp. 1286–1297.

[205] S. Ren, K. He, R. Girshick, and J. Sun, "Faster R-CNN: Towards real-time object detection with region proposal networks," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 39, no. 6, pp. 1137–1149, Jun. 2017.

[206] S. Lai, L. Xu, K. Liu, and J. Zhao, "Recurrent convolutional neural networks for text classification," 2015. [Online]. Available: https://www.aaai.org/ocs/index.php/AAAI/AAAI15/paper/view/9745/9552

[207] X. Fu, H. Cai, W. Li, and L. Li, "SEADS: Scalable and cost-effective dynamic dependence analysis of distributed systems via reinforcement learning," *ACM Trans. Softw. Eng. Methodol.*, vol. 30, no. 1, pp. 1–45, Dec. 2021. [Online]. Available: https://doi.org/10.1145/3379345

[208] T. Lutellier, H. V. Pham, L. Pang, Y. Li, M. Wei, and L. Tan, "Coconut: Combining context-aware neural translation models using ensemble for program repair," in *Proc. 29th ACM SIGSOFT Int. Symp. Softw. Testing Anal.*, 2020, pp. 101–114. [Online]. Available: https://doi.org/10.1145/3395363.3397369

[209] V. Saini *et al.*, "Towards automating precision studies of clone detectors," in *Proc. IEEE/ACM 41st Int. Conf. Softw. Eng.*, 2019, pp. 49–59.

[210] J. Deshmukh, K. M. Annervaz, S. Podder, S. Sengupta, and N. Dubash, "Towards accurate duplicate bug retrieval using deep learning techniques," in *Proc. IEEE Int. Conf. Softw. Maintenance Evol.*, 2017, pp. 115–124.

[211] D. Ma, Y. Bai, Z. Xing, L. Sun, and X. Li, "A knowledge graph-based sensitive feature selection for android malware classification," in *Proc. 27th Asia-Pacific Softw. Eng. Conf.*, 2020, pp. 188–197.

[212] T. Zhang, Q. Du, J. Xu, J. Li, and X. Li, "Software defect prediction and localization with attention-based models and ensemble learning," in *Proc. 27th Asia-Pacific Softw. Eng. Conf.*, 2020, pp. 81–90.

[213] H. Yu, W. Lam, L. Chen, G. Li, T. Xie, and Q. Wang, "Neural detection of semantic code clones via tree-based convolution," in *Proc. IEEE/ACM 27th Int. Conf. Prog. Comprehension*, 2019, pp. 70–80.

[214] J. Zhao, A. Albarghouthi, V. Rastogi, S. Jha, and D. Octeau, "Neural-augmented static analysis of android communication," in *Proc. 26th ACM Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, 2018, pp. 342–353. [Online]. Available: https://doi.org/10.1145/3236024.3236066

[215] Y. Shu, Y. Sui, H. Zhang, and G. Xu, "Perf-AL: Performance prediction for configurable software through adversarial learning," in *Proc. IEEE/ACM 14th Int. Symp. Empir. Softw. Eng. Meas.*, 2020, pp. 1–11. [Online]. Available: https://doi.org/10.1145/3382494.3410677

[216] J. Harer *et al.*, "Learning to repair software vulnerabilities with generative adversarial networks," in *Proc. Adv. Neural Informat. Process. Syst.*, 2018, pp. 7944–7954. [Online]. Available: https://proceedings.neurips.cc/paper/2018/file/68abef8ee1ac9b664a90b0bbaff4f770-Paper.pdf

[217] D. Zhao, Z. Xing, C. Chen, X. Xia, and G. Li, "ActionNet: Vision-based workflow action recognition from programming screencasts," in *Proc. IEEE/ACM 41st Int. Conf. Softw. Eng.*, 2019, pp. 350–361.

[218] L. Bao, Z. Xing, X. Xia, and D. Lo, "VT-Revolution: Interactive programming video tutorial authoring and watching system," *IEEE Trans. Softw. Eng.*, vol. 45, no. 8, pp. 823–838, Aug. 2019.

[219] T. Fritz, A. Begel, S. C. Müller, S. Yigit-Elliott, and M. Züger, "Using psycho-physiological measures to assess task difficulty in software development," in *Proc. 36th Int. Conf. Softw. Eng.*, 2014, pp. 402–413. [Online]. Available: https://doi.org/10.1145/2568225.2568266

[220] S. C. Müller and T. Fritz, "Using (bio)metrics to predict code quality online," in *Proc. IEEE/ACM 38th Int. Conf. Softw. Eng.*, 2016, pp. 452–463.

[221] D. Fucci, D. Girardi, N. Novielli, L. Quaranta, and F. Lanubile, "A replication study on code comprehension and expertise using lightweight biometric sensors," in *Proc. IEEE/ACM 27th Int. Conf. Prog. Comprehension*, 2019, pp. 311–322.

[222] D. Girardi, A. Ferrari, N. Novielli, P. Spoletini, D. Fucci, and T. Huichapa, "The way it makes you feel predicting users' engagement during interviews with biofeedback and supervised learning," in *Proc. IEEE 28th Int. Requirements Eng. Conf.*, 2020, pp. 32–43.

[223] A. Svyatkovskiy, S. K. Deng, S. Fu, and N. Sundaresan, *IntelliCode Compose: Code Generation Using Transformer*. New York, NY, USA: Association for Computing Machinery, 2020, pp. 1433–1443. [Online]. Available: https://doi.org/10.1145/3368089.3417058

[224] X. Lu, Y. Cao, Z. Chen, and X. Liu, "A first look at emoji usage on github: An empirical study," 2018, *arXiv:1812.04863*.

[225] Z. Chen, Y. Cao, X. Lu, Q. Mei, and X. Liu, "SEntiMoji: An emoji-powered learning approach for sentiment analysis in software engineering," in *Proc. 27th ACM Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, 2019, pp. 841–852. [Online]. Available: https://doi.org/10.1145/3338906.3338977

[226] K. Clark, M.-T. Luong, Q. V. Le, and C. D. Manning, "Electra: Pre-training text encoders as discriminators rather than generators," 2020, *arXiv:2003.10555*.

[227] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," in *Proc. Conf. North Amer. Chapter Assoc. Comput. Linguistics Hum. Lang. Technol.*, 2019, pp. 4171–4186. [Online]. Available: https://aclanthology.org/N19–1423

[228] J. Devlin, M.-W. Chang, and K. L. an Kristina Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," in *Proc. Conf. North Amer. Chapter Assoc. Comput. Linguistics Hum. Lang. Technol.*, 2019, pp. 4171–4186.

[229] K. Ethayarajh, "How contextual are contextualized word representations? comparing the geometry of BERT, ELMO, and GPT-2 embeddings," 2019, *arXiv:1909.00512*.

[230] N. S. Rao, N. Imam, J. Hanley, and S. Oral, "Wide-area lustre file system using LNet routers," in *Proc. IEEE Annu. Int. Syst. Conf.*, 2018, pp. 1–6.

[231] Y. Liu *et al.*, "RoBERTa: A robustly optimized BERT pretraining approach," 2019, *arXiv:1907.11692*.

[232] C. Raffel *et al.*, "Exploring the limits of transfer learning with a unified text-to-text transformer," *J. Mach. Learn. Res.*, vol. 21, no. 140, pp. 1–67, 2020. [Online]. Available: http://jmlr.org/papers/v21/20–074.html

[233] M. Lewis *et al.*, "BART: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension," 2019, *arXiv:1910.13461*.

[234] R.-M. Karampatsis and C. Sutton, "SCELMo: Source code embeddings from language models," 2020, *arXiv:2004.13214*.

[235] J. Zhang, H. Hong, Y. Zhang, Y. Wan, Y. Liu, and Y. Sui, "Disentangled code representation learning for multiple programming languages," in *Proc. Findings Assoc. Comput. Linguistics*, 2021, pp. 4454–4466.

[236] A. Kanade, P. Maniatis, G. Balakrishnan, and K. Shi, "Learning and evaluating contextual embedding of source code," in *Proc. Int. Conf. Mach. Learn.*, 2020, pp. 5110–5121.

[237] L. Buratti *et al.*, "Exploring software naturalness through neural language models," 2020, *arXiv:2006.12641*.

[238] N. T. de Sousa and W. Hasselbring, "JavaBERT: Training a transformer-based model for the java programming language," 2021, *arXiv:2110.10404*.

[239] F. Liu, G. Li, Y. Zhao, and Z. Jin, "Multi-task learning based pre-trained language model for code completion," in *Proc. IEEE/ACM 35th Int. Conf. Automated Softw. Eng.*, 2020, pp. 473–485.

[240] Z. Feng *et al.*, "CodeBERT: A pre-trained model for programming and natural languages," in *Proc. Conf. Empir. Methods Natural Lang. Process. Findings*, 2020, pp. 1536–1547.

[241] D. Peng, S. Zheng, Y. Li, G. Ke, D. He, and T.-Y. Liu, "How could neural networks understand programs?," in *Proc. Int. Conf. Mach. Learn.*, 2021, pp. 8476–8486.

[242] D. Guo *et al.*, "GraphcodeBERT: Pre-training code representations with data flow," in *Proc. Int. Conf. Learn. Representations*, 2021, p. 18.

[243] X. Wang *et al.*, "SyncoBERT: Syntax-guided multi-modal contrastive pre-training for code representation," 2021, *arXiv:2108.04556*.

[244] A. Svyatkovskiy, S. K. Deng, S. Fu, and N. Sundaresan, "IntelliCode compose: Code generation using transformer," in *Proc. 28th ACM Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, 2020, pp. 1433–1443.

[245] B. Roziere, M.-A. Lachaux, M. Szafraniec, and G. Lample, "DOBF: A deobfuscation pre-training objective for programming languages," 2021, *arXiv:2102.07492*.

[246] D. Drain, C. Wu, A. Svyatkovskiy, and N. Sundaresan, "Generating bug-fixes using pretrained transformers," in *Proc. 5th ACM SIGPLAN Int. Symp. Mach. Program.*, 2021, pp. 1–8.

[247] A. Mastropaolo *et al.*, "Studying the usage of text-to-text transfer transformer to support code-related tasks," in *Proc. Int. Conf. Softw. Eng.*, 2021, pp. 336–347.

[248] W. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang, "Unified pre-training for program understanding and generation," in *Proc. Conf. North Amer. Chapter Assoc. Comput. Linguistics - Hum. Lang. Technol.*, 2021, pp. 2655–2668.

[249] L. Phan *et al.*, "Cotext: Multi-task learning with code-text transformer," 2021, *arXiv:2105.08645*.

[250] W. Qi *et al.*, "ProphetNet-x: Large-scale pre-training models for english, chinese, multi-lingual, dialog, and code generation," 2021, *arXiv:2104.08006*.

[251] Y. Wang, W. Wang, S. Joty, and S. C. Hoi, "Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation," in *Proc. Conf. Empir. Methods Natural Lang. Process.*, 2021, pp. 8696–8708.

[252] X. Jiang, Z. Zheng, C. Lyu, L. Li, and L. Lyu, "TreeBERT: A tree-based pre-trained model for programming language," in *Proc. Uncertainty Artif. Intell.*, 2021, pp. 54–63.

[253] C. Niu, C. Li, V. Ng, J. Ge, L. Huang, and B. Luo, "SPT-code: Sequence-to-sequence pre-training for learning source code representations," 2022, *arXiv:2201.01549*.

[254] S. Ao, T. Zhou, G. Long, Q. Lu, L. Zhu, and J. Jiang, "CO-PILOT: Collaborative planning and reinforcement learning on sub-task curriculum," in *Proc. Adv. Neural Informat. Process. Syst.*, 2021, pp. 10444–10456.

[255] S.-T. Shi, M. Li, D. Lo, F. Thung, and X. Huo, "Automatic code review by learning the revision of source code," in *Proc. AAAI Conf. Artif. Intell.*, pp. 4910–4917, 2019. [Online]. Available: https://ojs.aaai.org/index.php/AAAI/article/view/4420

[256] R. Cai, Z. Liang, B. Xu, Z. Li, Y. Hao, and Y. Chen, "TAG : Type auxiliary guiding for code comment generation," in *Proc. 58th Annu. Meeting Assoc. Comput. Linguistics*, 2020, pp. 291–301. [Online]. Available: https://aclanthology.org/2020.acl-main.27

[257] E. Dinella, H. Dai, Z. Li, M. Naik, L. Song, and K. Wang, "Hoppity: Learning graph transformations to detect and fix bugs in programs," in *Proc. Int. Conf. Learn. Representations*, 2020, p. 17. [Online]. Available: https://openreview.net/forum?id=SJeqs6EFvB

[258] J. Zhang, L. Zhang, M. Harman, D. Hao, Y. Jia, and L. Zhang, "Predictive mutation testing," *IEEE Trans. Softw. Eng.*, vol. 45, no. 9, pp. 898–918, Sep. 2019.

[259] M. Golagha, A. Pretschner, and L. C. Briand, "Can we predict the quality of spectrum-based fault localization?," in *Proc. IEEE 13th Int. Conf. Softw. Testing, Validation Verification*, 2020, pp. 4–15.

[260] A. LeClair, Z. Eberhart, and C. McMillan, "Adapting neural text classification for improved software categorization," in *Proc. IEEE Int. Conf. Softw. Maintenance Evol.*, 2018, pp. 461–472.

[261] Y. Zou, B. Ban, Y. Xue, and Y. Xu, "CCGraph: A PDG-based code clone detector with approximate graph matching," in *Proc. 35th IEEE/ACM Int. Conf. Automated Softw. Eng.*, 2020, pp. 931–942.

[262] K. S. Tai, R. Socher, and C. D. Manning, “Improved semantic representations from tree-structured long short-term memory networks,” in *Proc. 53rd Annu. Meeting Assoc. Comput. Linguistics 7th Int. Joint Conf. Natural Lang. Process.*, 2015, pp. 1556–1566. [Online]. Available: https://www.aclweb.org/anthology/P15–1150

[263] J. Bruna, W. Zaremba, A. Szlam, and Y. LeCun, “Spectral networks and locally connected networks on graphs,” in *Proc. 2nd Int. Conf. Learn. Representations*, 2014, p. 14.

[264] H. Liu, Y. Yu, S. Li, Y. Guo, D. Wang, and X. Mao, “BugSum: Deep context understanding for bug report summarization,” in *Proc. 28th Int. Conf. Prog. Comprehension*, 2020, pp. 94–105. [Online]. Available: https://doi.org/10.1145/3387904.3389272

[265] I. Sutskever, O. Vinyals, and Q. V. Le, “Sequence to sequence learning with neural networks,” in *Proc. Adv. Neural Informat. Process. Syst.*, 2014, pp. 3104–3112.

[266] S. Jiang, A. Armaly, and C. McMillan, “Neural machine translation by jointly learning to align and translate,” in *Proc. 3rd Int. Conf. Learn. Representations*, 2015, p. 15.

[267] J. Cambronero, H. Li, S. Kim, K. Sen, and S. Chandra, “When deep learning met code search,” in *Proc. 27th ACM Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, 2019, pp. 964–974. [Online]. Available: https://doi.org/10.1145/3338906.3340458

[268] L. Shi *et al.*, “Learning to extract transaction function from requirements: An industrial case on financial software,” in *Proc. 28th ACM Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, 2020, pp. 1444–1454. [Online]. Available: https://doi.org/10.1145/3368089.3417053

[269] Z. Mahmood, D. Bowes, T. Hall, P. C. Lane, and J. Petrić, “Reproducibility and replicability of software defect prediction studies,” *Informat. Softw. Technol.*, vol. 99, pp. 148–163, 2018. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0950584917304202

[270] S. Akbarinasaji, B. Caglayan, and A. Bener, “Predicting bug-fixing time: A replication study using an open source software project,” *J. Syst. Softw.*, vol. 136, pp. 173–186, 2018. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121217300365

[271] W. Zhang, L. Ziqiang,W. Qing, and L. Juan, “Finelocator: A novel approach to method-level fine-grained bug localization by query expansion,” *Informat. Softw. Technol.*, vol. 110, pp. 121–135, 2019. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950584919300436

[272] M. Habayeb, S. S. Murtaza, A. Miranskyy, and A. B. Bener, “On the use of hidden Markov model to predict the time to fix bugs,” *IEEE Trans. Softw. Eng.*, vol. 44, no. 12, pp. 1224–1244, Dec. 2018.

[273] C. Ni, X. Xia, D. Lo, X. Chen, and Q. Gu, “Revisiting supervised and unsupervised methods for effort-aware cross-project defect prediction,” *IEEE Trans. Softw. Eng.*, vol. 48, no. 3, pp. 786–802, Mar. 2020.

[274] S. García, J. Luengo, and F. Herrera, “Tutorial on practical tips of the most influential data preprocessing algorithms in data mining,” *Knowl.-Based Syst.*, vol. 98, pp. 1–29, 2016. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0950705115004785

[275] D. Bowes, T. Hall, and J. Petrić, “Software defect prediction: Do different classifiers find the same defects?,” *Softw. Qual. J.*, vol. 26, no. 2, pp. 525–552, 2018.

[276] D. Gray, D. Bowes, N. Davey, Y. Sun, and B. Christianson, “Reflections on the NASA MDP data sets,” *IET Softw.*, vol. 6, no. 6, pp. 549–558, 2012.

[277] Z.-W. Zhang, X.-Y. Jing, and F. Wu, “Low-rank representation for semi-supervised software defect prediction,” *IET Softw.*, vol. 12, no. 6, pp. 527–535, 2018.

[278] K. K. Sabor, M. Hamdaqa, and A. Hamou-Lhadj, “Automatic prediction of the severity of bugs using stack traces and categorical features,” *Informat. Softw. Technol.*, vol. 123, 2020, Art. no. 106205. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950584919302137

[279] S. Wold, K. Esbensen, and P. Geladi, “Principal component analysis,” *Chemometrics Intell. Lab. Syst.*, vol. 2, no. 1–3, pp. 37–52, 1987.

[280] Wikipedia contributors, “Feature scaling — Wikipedia, the free encyclopedia,” 2021, Accessed: Aug. 08, 2021. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Feature_scaling&oldid=1031633529

[281] F. Zhang, Q. Zheng, Y. Zou, and A. E. Hassan, “Cross-project defect prediction using a connectivity-based unsupervised classifier,” in *Proc. IEEE/ACM 38th Int. Conf. Softw. Eng.*, 2016, pp. 309–320.

[282] X.-Y. Jing, F. Qi, F. Wu, and B. Xu, “Missing data imputation based on low-rank recovery and semi-supervised regression for software effort estimation,” in *Proc. IEEE/ACM 38th Int. Conf. Softw. Eng.*, 2016, pp. 607–618.

[283] W. Zhang, Y. Yang, and Q. Wang, “Using Bayesian regression and EM algorithm with missing handling for software effort prediction,” *Informat. Softw. Technol.*, vol. 58, pp. 58–70, 2015. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S095058491400216X

[284] N. Mittas, E. Papatheocharous, L. Angelis, and A. S. Andreou, “Integrating non-parametric models with linear components for producing software cost estimations,” *J. Syst. Softw.*, vol. 99, pp. 120–134, 2015. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121214002088

[285] A. Mockus, “Missing data in software engineering,” *Guide to Advanced Empirical Software Engineering*. Berlin, Germany: Springer, 2008, pp. 185–200.

[286] Y. Zhou, B. Xu, H. Leung, and L. Chen, “An in-depth study of the potentially confounding effect of class size in fault prediction,” *ACM Trans. Softw. Eng. Methodol.*, vol. 23, no. 1, pp. 1–51, Feb. 2014. [Online]. Available: https://doi.org/10.1145/2556777

[287] Y. Yang *et al.*, “Effort-aware just-in-time defect prediction: Simple unsupervised models could be better than supervised models,” in *Proc. 24th ACM SIGSOFT Int. Symp. on Found. Softw. Eng.*, 2016, pp. 157–168. [Online]. Available: https://doi.org/10.1145/2950290.2950353

[288] J. Jiarpakdee, C. Tantithamthavorn, and C. Treude, “The impact of automated feature selection techniques on the interpretation of defect models,” *Empir. Softw. Eng.*, vol. 25, no. 5, pp. 3590–3638, 2020.

[289] S. Mensah, J. Keung, M. F. Bosu, and K. E. Bennin, “Duplex output software effort estimation model with self-guided interpretation,” *Informat. Softw. Technol.*, vol. 94, pp. 1–13, 2018. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950584916304025

[290] J. Nam, W. Fu, S. Kim, T. Menzies, and L. Tan, “Heterogeneous defect prediction,” *IEEE Trans. Softw. Eng.*, vol. 44, no. 9, pp. 874–896, Sep. 2018.

[291] P. R. Anish *et al.*, “Probing for requirements knowledge to stimulate architectural thinking,” in *Proc. 38th Int. Conf. Softw. Eng.*, 2016, pp. 843–854. [Online]. Available: https://doi.org/10.1145/2884781.2884801

[292] R. Malhotra and M. Khanna, “An empirical study for software change prediction using imbalanced data,” *Empir. Softw. Eng.*, vol. 22, no. 6, pp. 2806–2851, 2017.

[293] P. Phannachitta, “On an optimal analogy-based software effort estimation,” *Informat. Softw. Technol.*, vol. 125, 2020, Art. no. 106330. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950584920300872

[294] G. Carrozza, D. Cotroneo, R. Natella, R. Pietrantuono, and S. Russo, “Analysis and prediction of mandelbugs in an industrial software system,” in *Proc. IEEE 6th Int. Conf. Softw. Testing, Verification Validation*, 2013, pp. 262–271.

[295] J. Zhang, X. Wang, H. Zhang, H. Sun, K. Wang, and X. Liu, “A novel neural source code representation based on abstract syntax tree,” in *Proc. IEEE/ACM 41st Int. Conf. Softw. Eng.*, 2019, pp. 783–794.

[296] A. S. Andreou and S. P. Chatzis, “Software defect prediction using doubly stochastic poisson processes driven by stochastic belief networks,” *J. Syst. Softw.*, vol. 122, pp. 72–82, 2016. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121216301601

[297] C. López-Martín, Y. Villuendas-Rey, M. Azzeh, A. Bou Nassif, and S. Banitaan, “Transformed k-nearest neighborhood output distance minimization for predicting the defect density of software projects,” *J. Syst. Softw.*, vol. 167, 2020, Art. no. 110592. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121220300728

[298] S. Mensah, J. Keung, S. G. MacDonell, M. F. Bosu, and K. E. Bennin, “Investigating the significance of bellwether effect to improve software effort estimation,” in *Proc. IEEE Int. Conf. Softw. Qual., Rel. Secur.*, 2017, pp. 340–351.

[299] X. Gu, H. Zhang, D. Zhang, and S. Kim, “Deep API learning,” in *Proc. 24th ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 2016, pp. 631–642. [Online]. Available: https://doi.org/10.1145/2950290.2950334

[300] K. Moran, C. Bernal-Cárdenas, M. Curcio, R. Bonett, and D. Poshyvanyk, "Machine learning-based prototyping of graphical user interfaces for mobile apps," *IEEE Trans. Softw. Eng.*, vol. 46, no. 2, pp. 196–221, Feb. 2020.

[301] M. H. Osman, M. R. Chaudron, and P. V. D. Putten, "An analysis of machine learning algorithms for condensing reverse engineered class diagrams," in *Proc. IEEE Int. Conf. Softw. Maintenance*, 2013, pp. 140–149.

[302] Y. Ma, G. Luo, X. Zeng, and A. Chen, "Transfer learning for cross-company software defect prediction," *Informat. Softw. Technol.*, vol. 54, no. 3, pp. 248–256, 2012. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950584911001996

[303] I. H. Witten and E. Frank, "Data mining: Practical machine learning tools and techniques with Java implementations," *ACM SIGMOD Rec.*, vol. 31, no. 1, pp. 76–77, 2002.

[304] P. Liu, X. Zhang, M. Pistoia, Y. Zheng, M. Marques, and L. Zeng, "Automatic text input generation for mobile testing," in *Proc. IEEE/ACM 39th Int. Conf. Softw. Eng.*, 2017, pp. 643–653.

[305] J. Bergstra and Y. Bengio, "Random search for hyper-parameter optimization," *J. Mach. Learn. Res.*, vol. 13, no. Feb, pp. 281–305, 2012.

[306] W. Fu, T. Menzies, and X. Shen, "Tuning for software analytics: Is it really necessary?," *Informat. Softw. Technol.*, vol. 76, pp. 135–146, 2016. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0950584916300738

[307] C. Stanik, L. Montgomery, D. Martens, D. Fucci, and W. Maalej, "A simple NLP-based approach to support onboarding and retention in open source communities," in *Proc. IEEE Int. Conf. Softw. Maintenance Evol.*, 2018, pp. 172–182.

[308] G. Uddin and F. Khomh, "Automatic summarization of API reviews," in *Proc. IEEE/ACM 32nd Int. Conf. Automated Softw. Eng.*, 2017, pp. 159–170.

[309] J. Shimagaki, Y. Kamei, N. Ubayashi, and A. Hindle, "Automatic topic classification of test cases using text mining at an Android smartphone vendor," in *Proc. IEEE/ACM 12th Int. Symp. Empir. Softw. Eng. Meas.*, 2018, pp. 1–10. [Online]. Available: https://doi.org/10.1145/3239235.3268927

[310] A. L. Oliveira, P. L. Braga, R. M. Lima, and M. L. Cornélio, "GA-based method for feature selection and parameters optimization for machine learning regression applied to software effort estimation," *Informat. Softw. Technol.*, vol. 52, no. 11, pp. 1155–1166, 2010. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950584910000984

[311] C. Theisen and L. Williams, "Better together: Comparing vulnerability prediction models," *Informat. Softw. Technol.*, vol. 119, 2020, Art. no. 106204. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950584919302125

[312] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, and K. Matsumoto, "The impact of automated parameter optimization on defect prediction models," *IEEE Trans. Softw. Eng.*, vol. 45, no. 7, pp. 683–711, Jul. 2019.

[313] H. Ha and H. Zhang, "DeepPerf: Performance prediction for configurable software with deep sparse neural network," in *Proc. IEEE/ACM 41st Int. Conf. Softw. Eng.*, 2019, pp. 1095–1106.

[314] W. Fu and T. Menzies, "Easy over hard: A case study on deep learning," in *Proc. 11th Joint Meeting Found. Softw. Eng.*, 2017, pp. 49–60. [Online]. Available: http://doi.acm.org/10.1145/3106237.3106256

[315] Y. Tian and Y. Zhang, "A comprehensive survey on regularization strategies in machine learning," *Informat. Fusion*, vol. 80, pp. 146–166, 2022. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S156625352100230X

[316] D. Pizzolotto and K. Inoue, "Identifying compiler and optimization options from binary code using deep learning approaches," in *Proc. IEEE Int. Conf. Softw. Maintenance Evol.*, 2020, pp. 232–242.

[317] Y. Wan *et al.*, "Multi-modal attention network learning for semantic source code retrieval," in *Proc. IEEE/ACM 34th Int. Conf. Automated Softw. Eng.*, 2019, pp. 13–25.

[318] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting," *J. Mach. Learn. Res.*, vol. 15, no. 56, pp. 1929–1958, 2014. [Online]. Available: http://jmlr.org/papers/v15/srivastava14a.html

[319] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," in *Proc. 13th Int. Conf. Artif. Intell. Statist.*, 2010, pp. 249–256. [Online]. Available: https://proceedings.mlr.press/v9/glorot10a.html

[320] L. Duong, H. Afshar, D. Estival, G. Pink, P. R. Cohen, and M. Johnson, "Multilingual semantic parsing and code-switching," in *Proc. 21st Conf. Comput. Natural Lang. Learn.*, 2017, pp. 379–389.

[321] S. Tabassum, L. L. Minku, D. Feng, G. G. Cabral, and L. Song, "An investigation of cross-project learning in online just-in-time software defect prediction," in *Proc. IEEE/ACM 42nd Int. Conf. Softw. Eng.*, 2020, pp. 554–565.

[322] K. Zhu, N. Zhang, S. Ying, and D. Zhu, "Within-project and cross-project just-in-time defect prediction based on denoising autoencoder and convolutional neural network," *IET Softw.*, vol. 14, no. 3, pp. 185–195, 2020.

[323] A. Podgurski and Y. Küçük, "Counterfault: Value-based fault localization by modeling and predicting counterfactual outcomes," in *Proc. IEEE Int. Conf. Softw. Maintenance Evol.*, 2020, pp. 382–393.

[324] P. Ma, H. Cheng, J. Zhang, and J. Xuan, "Can this fault be detected: A study on fault detection via automated test generation," *J. Syst. Softw.*, vol. 170, 2020, Art. no. 110769. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121220301862

[325] Z. Zhang, Y. Lei, X. Mao, M. Yan, L. Xu, and J. Wen, "Improving deep-learning-based fault localization with resampling," *J. Softw. Evol. Process*, vol. 33, no. 3, 2021, Art. no. e2312. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/smr.2312

[326] J. Chen, P. K. Kudjo, S. Mensah, S. A. Brown, and G. Akorfu, "An automatic software vulnerability classification framework using term frequency-inverse gravity moment and feature selection," *J. Syst. Softw.*, vol. 167, 2020, Art. no. 110616. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121220300947

[327] K. Z. Sultana, V. Anu, and T.-Y. Chong, "Using software metrics for predicting vulnerable classes and methods in Java projects: A machine learning approach," *J. Softw. Evol. Process*, vol. 33, no. 3, 2021, Art. no. e2303. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/smr.2303

[328] A. Agrawal, W. Fu, D. Chen, X. Shen, and T. Menzies, "How to "DODGE," complex software analytics," *IEEE Trans. Softw. Eng.*, vol. 47, no. 10, pp. 2182–2194, Oct. 2021.

[329] D. J. Hand and R. J. Till, "A simple generalisation of the area under the ROC curve for multiple class classification problems," *Mach. Learn.*, vol. 45, no. 2, pp. 171–186, 2001.

[330] M. Hossin and M. N. Sulaiman, "A review on evaluation metrics for data classification evaluations," *Int. J. Data Mining Knowl. Manage. Process*, vol. 5, no. 2, 2015, Art. no. 1.

[331] L. Minku, F. Sarro, E. Mendes, and F. Ferrucci, "How to make best use of cross-company data for web effort estimation?," in *Proc. IEEE/ACM Int. Symp. Empir. Softw. Eng. Meas.*, 2015, pp. 1–10.

[332] T. Nguyen *et al.*, "Complementing global and local contexts in representing API descriptions to improve API retrieval tasks," in *Proc. 26th ACM Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, 2018, pp. 551–562. [Online]. Available: https://doi.org/10.1145/3236024.3236036

[333] P. W. McBurney *et al.*, "Towards prioritizing documentation effort," *IEEE Trans. Softw. Eng.*, vol. 44, no. 9, pp. 897–913, Sep. 2018.

[334] B. Wei, Y. Li, G. Li, X. Xia, and Z. Jin, "Retrieve and refine: Exemplar-based neural comment generation," in *Proc. IEEE/ACM 35th Int. Conf. Automated Softw. Eng.*, 2020, pp. 349–360.

[335] W. Chan, S. Cheung, J. C. Ho, and T. Tse, "PAT: A pattern classification approach to automatic reference oracles for the testing of mesh simplification programs," *J. Syst. Softw.*, vol. 82, no. 3, pp. 422–434, 2009. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121208001817

[336] N. Yefet, U. Alon, and E. Yahav, "Adversarial examples for models of code," *Proc. ACM Prog. Lang.*, vol. 4, no. OOPSLA, pp. 1–30, Nov. 2020. [Online]. Available: https://doi.org/10.1145/3428230

[337] X. Ren, Z. Xing, X. Xia, D. Lo, X. Wang, and J. Grundy, "Neural network-based detection of self-admitted technical debt: From performance to explainability," *ACM Trans. Softw. Eng. Methodol.*, vol. 28, no. 3, pp. 1–45, Jul. 2019. [Online]. Available: https://doi.org/10.1145/3324916

[338] X. Liu, L. Huang, C. Li, and V. Ng, "Linking source code to untangled change intents," in *Proc. IEEE Int. Conf. Softw. Maintenance Evol.*, 2018, pp. 393–403.

[339] A. Result, "Artifact review and badging," 2017. [Online]. Available: https://www.acm.org/publications/policies/artifact-review-badging

[340] L. Pascarella, F. Palomba, and A. Bacchelli, "Re-evaluating method-level bug prediction," in *Proc. IEEE 25th Int. Conf. Softw. Anal., Evol. Reengineering*, 2018, pp. 592–601.

[341] K. Broman *et al.*, "Recommendations to funding agencies for supporting reproducible research," *Amer. Statist. Assoc.*, vol. 2, pp. 1–4, 2017.

[342] Y. Fan, C. Lv, X. Zhang, G. Zhou, and Y. Zhou, "The utility challenge of privacy-preserving data-sharing in cross-company defect prediction: An empirical study of the cliff amp;morph algorithm," in *Proc. IEEE Int. Conf. Softw. Maintenance Evol.*, 2017, pp. 80–90.

[343] Beautiful soup, 2022. [Online]. Available: https://www.crummy.com/software/BeautifulSoup/

[344] Scrapy, 2022. [Online]. Available: https://scrapy.org/

[345] J. undefinedliwerski, T. Zimmermann, and A. Zeller, "When do changes induce fixes?," in *Proc. Int. Workshop Mining Softw. Repositories*, 2005, pp. 1–5. [Online]. Available: https://doi.org/10.1145/1083142.1083147

[346] C. D. Manning, M. Surdeanu, J. Bauer, J. R. Finkel, S. Bethard, and D. McClosky, "The Stanford coreNLP natural language processing toolkit," in *Proc. 52nd Annu. Meeting Assoc. Comput. Linguistics Syst. Demonstrations*, 2014, pp. 55–60.

[347] M. Porter, "The Porter stemming algorithm," 2022. [Online]. Available: https://tartarus.org/martin/PorterStemmer/

[348] S. Wang *et al.*, *Automated Patch Correctness Assessment: How Far Are We?* New York, NY, USA: Association for Computing Machinery, 2020, pp. 968–980. [Online]. Available: https://doi.org/10.1145/3324884.3416590

[349] T. Menzies, S. Majumder, N. Balaji, K. Brey, and W. Fu, "500 + times faster than deep learning: (A case study exploring faster methods for text mining stackoverflow)," in *Proc. IEEE/ACM 15th Int. Conf. Mining Softw. Repositories*, 2018, pp. 554–563.

[350] B. Xu, A. Shirani, D. Lo, and M. A. Alipour, "Prediction of relatedness in stack overflow: Deep learning versus SVM: A reproducibility study," in *Proc. 12th ACM/IEEE Int. Symp. Empir. Softw. Eng. Meas.*, 2018, pp. 21:1–21:10. [Online]. Available: http://doi.acm.org/10.1145/3239235.3240503

[351] Y. Tian, M. Nagappan, D. Lo, and A. E. Hassan, "What are the characteristics of high-rated apps? A case study on free android applications," in *Proc. IEEE Int. Conf. Softw. Maintenance Evol.*, 2015, pp. 301–310.

[352] Y. Kamei, T. Fukushima, S. McIntosh, K. Yamashita, N. Ubayashi, and A. E. Hassan, "Studying just-in-time defect prediction using cross-project models," *Empir. Softw. Eng.*, vol. 21, no. 5, pp. 2072–2106, 2016.

[353] I. H. Laradji, M. Alshayeb, and L. Ghouti, "Software defect prediction using ensemble learning on selected features," *Informat. Softw. Technol.*, vol. 58, pp. 388–402, 2015. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S09505849 14001591

[354] M. Owhadi-Kareshk, S. Nadi, and J. Rubin, "Predicting merge conflicts in collaborative software development," in *Proc. IEEE/ACM Int. Symp. Empir. Softw. Eng. Meas.*, 2019, pp. 1–11.

[355] Y. Bai, Z. Xing, X. Li, Z. Feng, and D. Ma, "Unsuccessful story about few shot malware family classification and siamese network to the rescue," in *Proc. IEEE/ACM 42nd Int. Conf. Softw. Eng.*, 2020, pp. 1560–1571.

[356] Y. Nagashima and Y. He, "Pamper: Proof method recommendation system for Isabelle/HOL," in *Proc. IEEE/ACM 33rd Int. Conf. Automated Softw. Eng.*, 2018, pp. 362–372. [Online]. Available: https://doi.org/10.1145/3238147.3238210

[357] L. Song, L. L. Minku, and X. Yao, "A novel automated approach for software effort estimation based on data augmentation," in *Proc. 26th ACM Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, 2018, pp. 468–479. [Online]. Available: https://doi.org/10.1145/3236024.3236052

[358] N. Medeiros, N. Ivaki, P. Costa, and M. Vieira, "Software metrics as indicators of security vulnerabilities," in *Proc. IEEE 28th Int. Symp. on Softw. Rel. Eng.*, 2017, pp. 216–227.

[359] J. L. Dargan, J. S. Wasek, and E. Campos-Nanez, "Systems performance prediction using requirements quality attributes classification," *Requirements Eng.*, vol. 21, no. 4, pp. 553–572, 2016.

[360] Z. Xu, C. Wen, and S. Qin, "Type learning for binaries and its applications," *IEEE Trans. Rel.*, vol. 68, no. 3, pp. 893–912, Sep. 2019.

[361] X. Larrucea and I. Santamaría, "Correlations study and clustering from SPI experiences in small settings," *J. Softw. Evol. Process*, vol. 31, no. 1, 2019, Art. no. e1989. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/smr.1989

[362] M. Yan, X. Zhang, C. Liu, L. Xu, M. Yang, and D. Yang, "Automated change-prone class prediction on unlabeled dataset using unsupervised method," *Informat. Softw. Technol.*, vol. 92, pp. 1–16, 2017. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S095058491630163X

[363] E. Parra, C. Dimou, J. Llorens, V. Moreno, and A. Fraga, "A methodology for the classification of quality of requirements using machine learning techniques," *Informat. Softw. Technol.*, vol. 67, pp. 180–195, 2015. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950584915001299

[364] Y. Liu, L. Liu, H. Liu, and X. Wang, "Analyzing reviews guided by app descriptions for the software development and evolution," *J. Softw. Evol. Process*, vol. 30, no. 12, 2018, Art. no. e2112. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/smr.2112

[365] F. Dalpiaz, D. Dell'Anna, F. B. Aydemir, and S. Çevikol, "Requirements classification with interpretable machine learning and dependency parsing," in *Proc. IEEE 27th Int. Requirements Eng. Conf.*, 2019, pp. 142–152.

[366] M. T. Ribeiro, S. Singh, and C. Guestrin, ""Why should i trust you?": Explaining the predictions of any classifier," in *Proc. 22nd ACM SIGKDD Int. Conf. Knowl. Discov. Data Mining*, 2016, pp. 1135–1144. [Online]. Available: https://doi.org/10.1145/2939672.2939778

[367] C. Tantithamthavorn and J. Jiarpakdee, "Monash University," 2021, retrieved 2021–05-17. [Online]. Available: http://xai4se.github.io/

[368] R. Ben Abdessalem, S. Nejati, L. C. Briand, and T. Stifter, "Testing vision-based control systems using learnable evolutionary algorithms," in *Proc. IEEE/ACM 40th Int. Conf. Softw. Eng.*, 2018, pp. 1016–1026.

[369] G. Dong *et al.*, "Towards interpreting recurrent neural networks through probabilistic abstraction," in *Proc. IEEE/ACM 35th Int. Conf. Automated Softw. Eng.*, 2020, pp. 499–510.

[370] B. Wang, R. Ma, J. Kuang, and Y. Zhang, "How decisions are made in brains: Unpack "black box" of CNN with Ms. Pac-Man video game," *IEEE Access*, vol. 8, pp. 142 446–142 458, 2020.

[371] T. Mori and N. Uchihira, "Balancing the trade-off between accuracy and interpretability in software defect prediction," *Empir. Softw. Eng.*, vol. 24, no. 2, pp. 779–825, 2019.

[372] L. Ma *et al.*, "DeepMutation: Mutation testing of deep learning systems," in *Proc. IEEE 29th Int. Symp. Softw. Rel. Eng.*, 2018, pp. 100–111.

[373] Y. Sun, M. Wu, W. Ruan, X. Huang, M. Kwiatkowska, and D. Kroening, "Concolic testing for deep neural networks," in *Proc. IEEE/ACM 33rd Int. Conf. Automated Softw. Eng.*, 2018, pp. 109–119. [Online]. Available: http://doi.acm.org/10.1145/3238147.3238172

[374] J. Jiarpakdee, C. K. Tantithamthavorn, H. K. Dam, and J. Grundy, "An empirical study of model-agnostic techniques for defect prediction models," *IEEE Trans. Softw. Eng.*, vol. 48, no. 1, pp. 166–185, Jan. 2022.

[375] X. Wang, Y. Dang, L. Zhang, D. Zhang, E. Lan, and H. Mei, "Predicting consistency-maintenance requirement of code clonesat copy-and-paste time," *IEEE Trans. Softw. Eng.*, vol. 40, no. 8, pp. 773–794, Aug. 2014.

[376] S. Stapleton *et al.*, "A human study of comprehension and code summarization," in *Proc. 28th Int. Conf. Prog. Comprehension*, 2020, pp. 2–13. [Online]. Available: https://doi.org/10.1145/3387904.3389258

[377] Y. Bengio, A. Courville, and P. Vincent, "Representation learning: A review and new perspectives," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 35, no. 8, pp. 1798–1828, Aug. 2013.

[378] A. Blumer, A. Ehrenfeucht, D. Haussler, and M. K. Warmuth, "Occam's razor," *Informat. Process. Lett.*, vol. 24, no. 6, pp. 377–380, 1987. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0020019087901141

[379] P. Domingos, "The role of Occam's razor in knowledge discovery," *Data Mining Knowl. Discov.*, vol. 3, no. 4, pp. 409–425, 1999.

[380] B. Xu, D. Ye, Z. Xing, X. Xia, G. Chen, and S. Li, "Predicting semantically linkable knowledge in developer online forums via convolutional neural network," in *Proc. IEEE/ACM 31st Int. Conf. Automated Softw. Eng.*, 2016, pp. 51–62. [Online]. Available: http://doi.acm.org/10.1145/2970276.2970357

[381] Q. Song *et al.*, "A machine learning based software process model recommendation method," *J. Syst. Softw.*, vol. 118, pp. 85–100, 2016. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121216300425

[382] J. Kahles, J. Törrönen, T. Huuhtanen, and A. Jung, "Automating root cause analysis via machine learning in agile software testing environments," in *Proc. IEEE 12th Conf. Softw. Testing, Validation Verification*, 2019, pp. 379–390.

[383] C. Tantithamthavorn, J. Jiarpakdee, and J. Grundy, "Actionable analytics: Stop telling me what it is; please tell me what to do," *IEEE Softw.*, vol. 38, no. 4, pp. 115–120, Jul./Aug. 2021.

[384] D. Chen, W. Fu, R. Krishna, and T. Menzies, "Applications of psychological science for actionable analytics," in *Proc. 26th ACM Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, 2018, pp. 456–467. [Online]. Available: https://doi.org/10.1145/3236024.3236050

[385] C. Kästner and E. Kang, "Teaching software engineering for AL-enabled systems," in *Proc. IEEE/ACM 42nd Int. Conf. Softw. Eng. Softw. Eng. Educ. Training*, 2020, pp. 45–48.

[386] M. J. Islam, R. Pan, G. Nguyen, and H. Rajan, "Repairing deep neural networks: Fix patterns and challenges," in *Proc. IEEE/ACM 42nd Int. Conf. Softw. Eng.*, 2020, pp. 1135–1146.

[387] S. Gerasimou, H. F. Eniser, A. Sen, and A. Cakan, "Importance-driven deep learning system testing," in *Proc. IEEE/ACM 42nd Int. Conf. Softw. Eng.*, 2020, pp. 702–713.

[388] R. Zhang, W. Xiao, H. Zhang, Y. Liu, H. Lin, and M. Yang, "An empirical study on program failures of deep learning jobs," in *Proc. IEEE/ACM 42nd Int. Conf. Softw. Eng.*, 2020, pp. 1159–1170.

[389] J. M. Zhang, M. Harman, L. Ma, and Y. Liu, "Machine learning testing: Survey, landscapes and horizons," *IEEE Trans. Softw. Eng.*, vol. 48, no. 1, pp. 1–36, Jan. 2022.

[390] S. Udeshi, P. Arora, and S. Chattopadhyay, "Automated directed fairness testing," in *Proc. IEEE/ACM 33rd Int. Conf. Automated Softw. Eng.*, 2018, pp. 98–108. [Online]. Available: https://doi.org/10.1145/3238147.3238165

[391] Y. Feng, Q. Shi, X. Gao, J. Wan, C. Fang, and Z. Chen, "DeepGini: Prioritizing massive tests to enhance the robustness of deep neural networks," in *Proc. 29th ACM SIGSOFT Int. Symp. Softw. Testing Anal.*, 2020, pp. 177–188. [Online]. Available: https://doi.org/10.1145/3395363.3397357

[392] Y. Dang, D. Zhang, S. Ge, R. Huang, C. Chu, and T. Xie, "Transferring code-clone detection and analysis to practice," in *Proc. IEEE/ACM 39th Int. Conf. Softw. Eng.: Softw. Eng. Pract. Track*, 2017, pp. 53–62.

[393] J. S. Di Stefano and T. Menzies, "Machine learning for software engineering: Case studies in software reuse," in *Proc. IEEE 14th Int. Conf. Tools Artif. Intell.*, 2002, pp. 246–251.

[394] Y. Zhou *et al.*, "How far we have progressed in the journey? An examination of cross-project defect prediction," *ACM Trans. Softw. Eng. Methodol.*, vol. 27, no. 1, pp. 1:1–1:51, Apr. 2018. [Online]. Available: http://doi.acm.org/10.1145/3183339

[395] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, and K. Matsumoto, "An empirical comparison of model validation techniques for defect prediction models," *IEEE Trans. Softw. Eng.*, vol. 43, no. 1, pp. 1–18, Jan. 2017.

[396] S. Herbold, A. Trautsch, and J. Grabowski, "Global versus local models for cross-project defect prediction," *Empir. Softw. Eng.*, vol. 22, no. 4, pp. 1866–1902, Aug. 2017. [Online]. Available: https://doi.org/10.1007/s10664–016-9468-y

[397] M. Yan, Y. Fang, D. Lo, X. Xia, and X. Zhang, "File-level defect prediction: Unsupervised versus supervised models," in *Proc. IEEE/ACM Int. Symp. Empir. Softw. Eng. Meas.*, 2017, pp. 344–353.

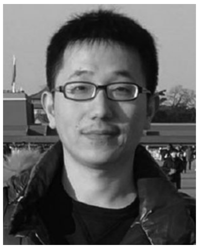
[398] T. Hall, S. Beecham, D. Bowes, D. Gray, and S. Counsell, "A systematic literature review on fault prediction performance in software engineering," *IEEE Trans. Softw. Eng.*, vol. 38, no. 6, pp. 1276–1304, Nov./Dec. 2012.

[399] M. Shepperd, D. Bowes, and T. Hall, "Researcher bias: The use of machine learning in software defect prediction," *IEEE Trans. Softw. Eng.*, vol. 40, no. 6, pp. 603–616, Jun. 2014.

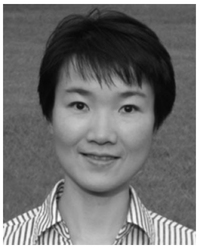
[400] E. Arisholm, L. C. Briand, and E. B. Johannessen, "A systematic and comprehensive investigation of methods to build and evaluate fault prediction models," *J. Syst. Softw.*, vol. 83, no. 1, pp. 2–17, 2010. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0164121209001605

[401] J. Wen, S. Li, Z. Lin, Y. Hu, and C. Huang, "Systematic literature review of machine learning based software development effort estimation models," *Informat. Softw. Technol.*, vol. 54, no. 1, pp. 41–59, 2012. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0950584911001832

[402] F. Arcelli Fontana, M. V. Mäntylä, M. Zanoni, and A. Marino, "Comparing and experimenting machine learning techniques for code smell detection," *Empir. Softw. Eng.*, vol. 21, no. 3, pp. 1143–1191, Jun. 2016. [Online]. Available: https://doi.org/10.1007/s10664–015-9378-4



Simin Wang is currently working toward the PhD degree in computer science with the Southern Methodist University (SMU), Dallas, TX, USA. His research focuses on synergy of machine/deep learning, natural language processing and software engineering. He is advised by Prof. LiGuo Huang.



Liguo Huang received the MS and PhD degrees from the Computer Science Department and Center for Systems and Software Engineering (CSSE), the University of Southern California (USC). She is an Associate Professor with Computer Science Department, the Southern Methodist University (SMU), Dallas, TX, USA. Her primary research centers around synergy of machine/deep learning, natural language processing and software engineering, software quality assurance, process modeling and improvement, stakeholder/value-based software engineering.

Amiao Gao is currently working toward the PhD degree in computer science with the Southern Methodist University (SMU), Dallas, TX, USA. Her research focuses on machine learning and deep learning in software engineering. She is advised by Prof. LiGuo Huang.

Jidong Ge received the PhD degree in computer science from Nanjing University in 2007. He is an Associate Professor with Software Institute, Nanjing University. He is also a member of the State Key Laboratory for Novel Software Technology. His current research interests include NLP and intelligent software engineering.

Tengfei Zhang received the ME degree in software engineering from Nanjing University, China, in 2020. He is currently a software engineer with Huawei, and was working toward the master's degree under supervision by Professor Jidong Ge. His research interests include software engineering and machine learning.

Haitao Feng received the ME Degree in software engineering from Nanjing University, China, in 2020. He is currently a software engineer with Xiaomi, and was working toward the master's degree under supervision by Professor Jidong Ge. His research interests include software engineering and machine learning.

Ishna Satyarth is currently working toward the PhD degree in computer science with the Lyle School of Engineering, Southern Methodist University. Her research focuses on machine learning and software engineering application. She is advised by Prof. LiGuo Huang.

Ming Li is currently a professor with Nanjing University. His major research interests include machine learning and data mining, especially on software mining. He has served as the area chair of IJCAI, IEEE ICDM, senior PC member of the premium conferences in artificial intelligence such as AAAI. He is the founding chair of the International Workshop on Software Mining. He has been granted various awards including PAKDD Early Career Award, etc.

He Zhang is a professor with Software Engineering and the director with DevOps+ Research Laboratory, Nanjing University, China, also a principal scientist with CSIRO, Australia. He joined academia after many years working in software industry. He undertakes research in software engineering, in particular software process, software architecture, DevOps, software security, empirical and evidence-based software engineering. He has published more than 160 peer-reviewed papers in prestigious international conferences and journals.

Vincent Ng received the BS degree from Carnegie Mellon University and the PhD degree from Cornell University. He is a professor with Computer Science and a member of the Human Language Technology Research Institute, the University of Texas at Dallas. His primary research is in the areas of natural language processing and AI-based software engineering.